Efficient Dataflow Analysis of Logic Programs

SAUMYA K. DEBRAY

The University of Arizona, Tucson, Arizona

Abstract. We investigate a framework for efficient dataflow analyses of logic programs. A number of problems arise in this context: aliasing effects can make analysis computationally expensive for sequential logic programming languages; synchronization issues can complicate the analysis of parallel logic programming languages; and finiteness restrictions to guarantee termination can limit the expressive power of such analyses. Our main result is to give a simple characterization of a family of flow analyses where these issues can be ignored without compromising soundness. This results in algorithms that are simple to verify and implement, and efficient in execution. Based on this approach, we describe an efficient algorithm for flow analysis of sequential logic programs, extend this approach to handle parallel executions, and finally describe how infinite chains in the analysis domain can be accommodated without compromising termination.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.4 [**Programming Languages**]: Processes—*compilers, optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Program Analysis, PROLOG

1. Introduction

Despite the numerous attractive features offered by logic programming languages, they can often be dismayingly inefficient in execution. This has given rise to a great deal of research in the analysis and optimization of logic programs (see Section 8). This work has addressed some of the issues raised by the presence of features, such as unification and nondeterminism, that are not found in traditional languages. However, there are a number of significant problems that appear to not have been addressed adequately in much of this work: the computational issues raised by aliasing, and by synchronization considerations for parallel logic programming languages; and issues of expressiveness of flow analysis systems arising out of finiteness constraints that are imposed to guarantee termination of analyses. The purpose of this paper is to address these issues by developing a framework for a class of

Preliminary versions of parts of this work have appeared in *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, Jan. 1988; and *Proceedings of the Fifth International Conference on Logic Programming*, Seattle, Aug. 1988.

This work was supported in part by the National Science Foundation under grant number CCR-8702939.

Author's address: Department of Computer Science, The University of Arizona, Tucson, AZ 85721; email: debray@cs.arizona.edu.

dataflow analysis problems commonly encountered in logic programming.

In the analysis of logic programs, unification and the presence of "logical variables" can give rise to aliasing, and dependencies between variables, whose effects can be difficult to predict, making the task of validating such analyses a nontrivial one (see [17; 34]). Moreover, in order to handle aliasing effects correctly, it is necessary in general to maintain information regarding dependencies between variables—a task that can seriously affect the efficiency of the analysis algorithms (for example, a number of flow analysis problems for traditional languages become intractable in the presence of recursion and aliasing [51]). Ideally, we would like to carry out our analyses ignoring aliasing effects, in the interests of efficiency, and still be guaranteed soundness. This raises the question of characterizing the class of flow analysis problems for logic programs for which aliasing effects can safely be ignored. Analysis algorithms for such problems can be greatly simplified, resulting in significant gains in efficiency. We give a simple characterization of the class of flow analysis problems for which aliasing effects can safely be ignored, and develop a general framework for such analyses. Our strategy in doing this is to develop a framework for the flow analyses of general logic programs that ignores aliasing effects, and then describing the conditions under which the analyses are sound; the soundness criterion then serves to characterize the class of flow analyses that can be safely carried out without worrying about aliasing. We show that aliasing effects can safely be ignored as long as the analysis domains satisfy a simple *substitution-closure* property.

In the analysis of parallel logic programs, a similar problem arises with regard to synchronization. In most AND-parallel logic programming models, processes communicate via shared variables [10; 11; 28; 37; 59; 63]. It is possible to devise analysis algorithms for such languages given a significant amount of information about the synchronization primitives of the language [8; 23; 62]. The problem with this is that such analyses become language-specific, making it difficult to generalize them across languages and execution models. On the other hand, if few assumptions are made about the execution model, it becomes difficult to predict the variable bindings seen by a process at any point in the execution without making further assumptions about the runtime system, e.g. the scheduler. Ideally, we would like to carry out our analyses ignoring such issues of synchronization, resulting in simpler and more efficient algorithms, and still be guaranteed soundness. This raises the question of characterizing the class of flow analysis problems for logic programs for which synchronization issues can be safely ignored. As with aliasing, we address this by initially ignoring synchronization issues, then describing the conditions under which such analyses are sound. It turns out that this soundness criterion is exactly the same as that for the aliasing case: the analysis domains have to be substitution-closed.

Finally, there is the issue of the expressive power of a flow analysis system. Since static analyses are expected to be uniformly terminating, finiteness constraints are usually imposed on analysis domains. For example, they are required to be of finite height, or satisfy the finite chain property. However, the *a priori* imposition of such finiteness constraints can result in a loss of expressive power and precision. We describe an approach that enables us to work with analysis domains containing infinite chains, and yet be guaranteed termination. In considering soundness criteria for such analyses, it turns out that substitution-closure is a necessary property.

Based on these results, we argue that substitution-closed analyses constitute an important family of dataflow analysis problems for logic programs. The utility of such analyses is illustrated with a number of example applications.

2. Preliminaries

Most logic programming languages are based on a subset of the first order predicate calculus known as Horn clause logic. Such a language has a countably infinite set of variables, and countable sets of function and predicate symbols, these sets being mutually disjoint. Without loss of generality, we assume that with each function symbol f and each predicate symbol p is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are said to be n-ary symbols, and written f/n and p/n respectively. A 0-ary function symbol is referred to as a constant.

A term in such a language is either a variable, or 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. A literal is either an atom $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, or the negation of an atom; in the first case the literal is said to be positive, in the second case it is negative. A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. The positive literal in a definite clause is its *head*, and the remaining literals constitute its *body*. A predicate definition consists of a finite number of definite clauses, all whose heads have the same predicate symbol; a goal is a set of negative literals. A logic program consists of a finite set of predicate definitions. For the purposes of analysis, it is assumed that we are given a module of the form $\langle P, \mathtt{EXPORTS}(P)\rangle$, where P is a set of predicate definitions and $\mathtt{EXPORTS}(P)$ specifies the predicates in P that are exported, i.e. that may be called from the outside. $\mathtt{EXPORTS}(P)$ is a set of pairs $\langle p, cp\rangle$, specifying that a predicate p may be called from the outside with arguments described by cp. There may be more than one entry for a predicate if it can be called in different ways.

In this paper, we adhere to the syntax of Edinburgh Prolog and write a definite clause as

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

read declaratively as "*p if q_1 and ... and q_n*". Names of variables begin with upper case letters, while names of non-variable (i.e. function and predicate) symbols begin with lower case letters. To simplify some aspects of the discussion that follows, we assume that each argument in the head of a clause is a variable: this does not lose any generality, since a clause

$$p(t_1, \ldots, t_n) \;:-\; Body$$

where $t_1, \ldots, t_n$ are arbitrary terms, can always be transformed to satisfy this assumption by rewriting it as

$$p(X_1, \ldots, X_n) \;:-\; X_1 = t_1, \ldots, X_n = t_n, Body$$

where $X_1, \ldots, X_n$ are distinct variables not appearing in the original clause.

A substitution is a mapping from variables to terms that is the identity mapping at all but finitely many points. A substitution σ_1 is said to be *more general* than a substitution σ_2 if there is a substitution θ such that $\sigma_2 = \theta \circ \sigma_1$. Two terms t_1 and t_2 are said to be *unifiable* if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; in this case, σ is said to be a *unifier* for the terms. If two terms t_1 and t_2 have a unifier, then they have a *most general unifier* that is unique up to variable renaming. To denote the term obtained as a result of unifying two given terms, we define the function *unify* as follows: given two terms t_1 and t_2,

$$\mathit{unify}(t_1, t_2) \quad = \quad \begin{cases} \theta(t_1) & \text{if } \mathrm{t}_1 \text{ and } \mathrm{t}_2 \text{ are unifiable with most general unifier } \theta \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

Note that because most general unifiers are unique only upto variable renaming, this function is not well-defined unless terms are considered modulo renaming. In the discussion that follows, therefore, we will not distinguish between alphabetic variants of a term, unless explicitly mentioned.

The operational behavior of logic programs can be described by means of *SLD-derivations*. An SLD-derivation for a goal G with respect to a program P is a sequence of goals $G_0, \ldots, G_i, G_{i+1}, \ldots$ such that $G_0 = G$, and if $G_i =$ '$a_1, \ldots, a_n$', then $G_{i+1} = \theta(a_1, \ldots, a_{i-1}, b_1, \ldots, b_m, a_{i+1}, \ldots, a_n)$ such that $1 \leq i \leq n$; b :− $b_1, \ldots, b_m$ is an alphabetic variant of a clause in P and has no variable in common with any of the goals $G_0, \ldots, G_i$; and θ is the most general unifier of a_i and b. The goal G_{i+1} is said to be obtained from G_i by means of a *resolution* step, and a_i is said to be the *resolved atom.* Intuitively, each resolution step corresponds to a procedure call. Let $G_0, \ldots, G_n$ be an SLD-derivation for a goal G in a program P, and let θ_i be the unifier obtained when resolving the goal G_i to obtain G_{i+1}, $0 \leq i < n$; if this derivation is finite and maximal, i.e. one in which it is not possible to resolve the goal G_n with any of the clauses in P, then this corresponds to a terminating computation for G: in this case, if G_n is the empty goal then the computation is said to *succeed* with answer substitution θ, where θ is the substitution obtained by restricting the substitution $\theta_n \circ \cdots \circ \theta_0$ to the variables occurring in G. If G_n is not the empty goal, then the computation is said to *fail.* If the derivation is infinite, then the computation does not terminate.

Let $p(\bar{t})$ be the resolved atom in some SLD-derivation of a goal G in a program P, then we say that $p(\bar{t})$ is a *call* that arises in the computation of G in the program. If the goal $p(\bar{t})$ can succeed with answer substitution θ, then we also say that it can succeed with its arguments bound to $\theta(\bar{t})$.

We assume that the predicates in the program are static, i.e. do not have code created and executed dynamically at runtime, e.g. via Prolog primitives such as *call*, *assert* or *retract.* Somewhat more limited analyses can be performed for dynamic programs using the techniques described in [19].

3. A Flow Analysis Framework for Logic Programs

This section develops a framework for flow analysis of logic programs. We begin by discussing some fundamental notions and results in Section 3.1. This is followed, in Section 3.2, by a discussion of *abstraction structures*, which define the abstract domain for any analysis. Section 3.3 then considers how unification, the fundamental primitive operation of logic programming languages, can be abstracted.

3.1 Comparing Instantiations of Sets of Terms

During the execution of a logic program, terms become progressively more instantiated. The notion of a term being "less general" than another is quite straightforward when dealing with individual terms: a term t_1 is less general than another term t_2, written $t_1 \sqsubseteq t_2$, if t_1 is a substitution instance of t_2. The ordering $\sqsubseteq$ is called the *subsumption order* on terms, and is a partial order modulo variable renaming. However, the analyses we consider associate variables with sets of terms, so it becomes necessary to "lift" this order to sets of terms. Define unification over sets of terms, denoted by $\mathsf{s\_unify}$, as follows:

Definition 3.1. Given sets of terms T_1 and T_2, $\mathsf{s\_unify}(T_1, T_2)$ is the least set of terms T such that $\mathit{unify}(t_1, t_2)$ is in T for each pair of unifiable terms $t_1 \in T_1$ and $t_2 \in T_2$. ▮

Lemma 3.1. (Plotkin [54], Reynolds [56]) *The set of all terms of a first order language, augmented by a distinguished symbol* $-$ *such that* $- \sqsubseteq t$ *for any term* t*, forms a complete lattice when ordered by* $\sqsubseteq$*. For any two unifiable terms* t_1 *and* t_2*,* $\mathit{unify}(t_1, t_2) = t_1 \sqcap t_2$*, where* $\sqcap$ *is the meet operation of this lattice.* □

Given two terms t_1 and t_2, t_2 is more general than t_1 if and only if $t_1 \sqcap t_2 = t_1$, i.e. if and only if $\mathit{unify}(t_1, t_2) = t_1$. We define the instantiation order over sets of terms, denoted by $\trianglelefteq$, as the natural extension of this:

Definition 3.2. Given sets of terms T_1 and T_2, T_1 is *less general* than T_2, written $T_1 \trianglelefteq T_2$, if and only if $\mathsf{s\_unify}(T_1, T_2) = T_1$. ▮

The reader may verify that $\trianglelefteq$, as defined above, is transitive. If a set of terms T is closed under unification, i.e. for any t_1 and t_2 in T, if t_1 and t_2 are unifiable then $\mathit{unify}(t_1, t_2)$ is also in T, then $\mathsf{s\_unify}(T, T) = T$. If we only consider sets of terms that are closed under unification, therefore, $\trianglelefteq$ is also reflexive, and hence a preorder. It is straightforward to show that $T_1 \trianglelefteq T_2$ and $T_2 \trianglelefteq T_1$ for any sets of terms T_1 and T_2 if and only if T_1 and T_2 are alphabetic variants. Thus, for sets of terms that are closed under unification, the relation $\trianglelefteq$ is a partial order modulo variable renaming. The meet operation for this partial order, when it exists, will be written $\triangle$. Let $\mathcal{U}(\mathsf{Term}) \subseteq \wp(\mathsf{Term})$ denote the set of sets of terms that are closed under unification, then we have:

Proposition 3.2. *For any two sets of terms* T_1 *and* T_2 *in* $\mathcal{U}(\mathsf{Term})$*,* $\mathsf{s\_unify}(T_1, T_2) = T_1 \triangle T_2$*, and is closed under unification.*

Proof Consider any two sets of terms T_1 and T_2 that are closed under unification. Then,

$$\begin{aligned}
&\mathsf{s\_unify}(\mathsf{s\_unify}(T_1, T_2), T_1)\\
&= \{t \sqcap t' \mid t \in \mathsf{s\_unify}(T_1, T_2), t' \in T_1\}\\
&= \{(t_1 \sqcap t_2) \sqcap t' \mid t_1 \in T_1, t_2 \in T_2, t' \in T_1\}\\
&\qquad\qquad \text{by definition of } \mathsf{s\_unify}(T_1, T_2)\\
&= \{(t_1 \sqcap t') \sqcap t_2 \mid t_1 \in T_1, t_2 \in T_2, t' \in T_1\}\\
&\qquad\qquad \text{since } \sqcap \text{ is associative and commutative}\\
&= \{t \sqcap t_2 \mid t \in T_1, t_2 \in T_2\}\\
&\qquad\qquad \text{where } t = t_1 \sqcap t', \text{ since } T_1 \text{ is closed under unification}
\end{aligned}$$

$= \mathsf{s\_unify}(T_1, T_2)$.

This establishes that $\mathsf{s\_unify}(T_1, T_2) \trianglelefteq T_1$. A similar argument establishes that $\mathsf{s\_unify}(T_1, T_2) \trianglelefteq T_2$. Thus, $\mathsf{s\_unify}(T_1, T_2)$ is a lower bound on T_1 and T_2 with respect to $\trianglelefteq$.

Now consider any set of terms T that is closed under unification and is also a lower bound on T_1 and T_2 with respect to $\trianglelefteq$, i.e. $T \trianglelefteq T_1$ and $T \trianglelefteq T_2$. We show that $T \trianglelefteq \mathsf{s\_unify}(T_1, T_2)$. By definition, $T \trianglelefteq T_1$ implies that $\mathsf{s\_unify}(T, T_1) = T$, i.e. $\{t \sqcap t_1 \mid t \in T, t_1 \in T_1\} = T$. Similarly, $T \trianglelefteq T_2$ implies $\{t \sqcap t_2 \mid t \in T, t_2 \in T_2\} = T$. Then, we have

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{s\_unify}(T, \mathsf{s\_unify}(T_1, T_2))} \\
\multicolumn{2}{l}{= \{t \sqcap t' \mid t \in T, t' \in \mathsf{s\_unify}(T_1, T_2)\}} \\
\multicolumn{2}{l}{= \{t \sqcap (t_1 \sqcap t_2) \mid t \in T, t_1 \in T_1, t_2 \in T_2\}} \\
 & \text{by definition of } \mathsf{s\_unify} \\
\multicolumn{2}{l}{= \{(t \sqcap t_1) \sqcap (t \sqcap t_2) \mid t \in T, t_1 \in T_1, t_2 \in T_2\}} \\
 & \text{since } \sqcap \text{ is associative, commutative and idempotent} \\
\multicolumn{2}{l}{= \{u_1 \sqcap u_2 \mid u_1 \in T, u_2 \in T\}} \\
 & \text{where } u_1 = t \sqcap t_1, u_2 = t \sqcap t_2, \text{ since } T \trianglelefteq T_1 \text{ and } T \trianglelefteq T_2 \\
= T & \text{since } T \text{ is closed under unification.}
\end{array}$$

It follows that $T \trianglelefteq \mathsf{s\_unify}(T_1, T_2)$. This establishes that $\mathsf{s\_unify}(T_1, T_2)$ is the greatest lower bound of T_1 and T_2 for any two sets of terms T_1 and T_2 that are closed under unification, i.e. $\mathsf{s\_unify}(T_1, T_2) = T_1 \bigtriangleup T_2$.

To see that $\mathsf{s\_unify}(T_1, T_2)$ is closed under unification, consider any two elements u, v in $\mathsf{s\_unify}(T_1, T_2)$. By definition, $u \in \mathsf{s\_unify}(T_1, T_2)$ implies that there are terms $u_1 \in T_1$, $u_2 \in T_2$ such that $u = u_1 \sqcap u_2$. Similarly, $v \in \mathsf{s\_unify}(T_1, T_2)$ implies that there are terms $v_1 \in T_1$, $v_2 \in T_2$ such that $v = v_1 \sqcap v_2$. Then,

$$\begin{array}{ll}
\mathit{unify}(u, v) = u \sqcap v & \text{from Lemma 3.1} \\
= (u_1 \sqcap u_2) \sqcap (v_1 \sqcap v_2) & \text{since } u = u_1 \sqcap u_2, v = v_1 \sqcap v_2 \\
= (u_1 \sqcap v_1) \sqcap (u_2 \sqcap v_2) & \text{since } \sqcap \text{ is associative and commutative} \\
= t_1 \sqcap t_2 & \text{where } t_1 = u_1 \sqcap v_1, \text{ and } t_2 = u_2 \sqcap v_2.
\end{array}$$

Since u_1 and v_1 are both in T_1, and T_1 is closed under unification, it follows that $t_1 = u_1 \sqcap v_1$ is in T_1. Similarly, $t_2 = u_2 \sqcap v_2$ is in T_2. It follows, from the definition of $\mathsf{s\_unify}$, that $\mathit{unify}(u, v) = t_1 \sqcap t_2$ is in $\mathsf{s\_unify}(T_1, T_2)$. Since this holds for any $u, v \in \mathsf{s\_unify}(T_1, T_2)$, it follows that $\mathsf{s\_unify}(T_1, T_2)$ is closed under unification. □

COROLLARY 3.3. *$\langle \mathcal{U}(\mathsf{Term}), \trianglelefteq \rangle$ is a meet-semilattice.* □

Defining the order $\trianglelefteq$ essentially involves "lifting" the partial order $\sqsubseteq$ from terms to sets of terms. It is therefore natural to consider relationships with powerdomain orderings that have been proposed in the literature (see, for example, [60]). The following proposition gives a partial connection in this regard.

PROPOSITION 3.4. *Given sets of terms $T_1, T_2 \in \mathcal{U}(\mathsf{Term})$, $T_1 \trianglelefteq T_2$ implies $(\forall t_1 \in T_1)(\exists t_2 \in T_2)[\ t_1 \sqsubseteq t_2\ ]$.*

Proof Assume that $T_1 \trianglelefteq T_2$ but there is some term $t \in T_1$ such that no term $t' \in T_2$ satisfies $t \sqsubseteq t'$. Since $T_1 \trianglelefteq T_2$, it follows that $\mathsf{s\_unify}(T_1, T_2) = T_1$, whence $t \in \mathsf{s\_unify}(T_1, T_2)$. This implies that $t = \mathit{unify}(t_1, t_2)$ for some $t_1 \in T_1, t_2 \in T_2$.

From Lemma 3.1 it follows that $t = t_1 \sqcap t_2$. This means that there is a term $t_2 \in T_2$ such that $t \sqsubseteq t_2$, which is a contradiction. It follows that if $T_1 \trianglelefteq T_2$, then for every $t_1 \in T_1$ there is a term $t_2 \in T_2$ such that $t_1 \sqsubseteq t_2$. □

The converse, however, does not hold. To see this, consider the sets of terms $T_1 = \{f(Y,Y)\}$ and $T_2 = \{f(a,X), Z\}$. It is easy to see that both T_1 and T_2 are closed under unification, and for each t_1 in T_1 there is a $t_2 \in T_2$ such that $t_1 \sqsubseteq t_2$ (the only candidate t_1 is the term $f(Y,Y)$, and the variable $Z \in T_2$ satisfies $f(Y,Y) \sqsubseteq Z$). However, $f(a,a) = \mathit{unify}(f(Y,Y), f(a,X))$ is not in T_1, so $\mathsf{s\_unify}(T_1,T_2) \neq T_1$.

3.2 Abstraction Structures

Let Term be the set of all terms of a given first order language. As a first step in the development of a general framework for the flow analysis of logic programs, we define a family of approximation domains $\mathcal{D} \subseteq \mathcal{U}(\mathsf{Term})$ whose elements are "canonical" sets of terms for compile-time analyses. It seems reasonable to require that the empty set $\emptyset$ and the set of all terms Term, representing the two extremes of information that we can have regarding a computation, should be in $\mathcal{D}$. In order that $\trianglelefteq$ be a partial order over $\mathcal{D}$, elements of $\mathcal{D}$ must be closed under unification. Analyses will typically assume that variables in a clause are uninstantiated, i.e. in their most general state, when its execution begins: for this, $\mathcal{D}$ must have a greatest element $\top_{inst}$ with respect to $\trianglelefteq$. Moreover, in order that unification of sets of terms during static analysis be well defined, it is necessary that for any two elements d_1 and d_2 in $\mathcal{D}$, their meet $d_1 \bigtriangleup d_2$ also be in $\mathcal{D}$.

A set $\mathcal{D}$ satisfying these properties is called an *instantiation set*. In referring to instantiation sets, the set of all terms Term will also be denoted by any. Then, we have the following definition:

DEFINITION 3.3. An *instantiation set* is a finite set $\mathcal{D} \subseteq \mathcal{U}(\mathsf{Term})$ satisfying the following properties:

(1) The empty set $\emptyset$ and the set of all terms any are in $\mathcal{D}$.

(2) There is a greatest element $\top_{inst}$ in $\mathcal{D}$ with respect to $\trianglelefteq$.

(3) For any two elements d_1 and d_2 in $\langle \mathcal{D}, \trianglelefteq \rangle$, their meet $d_1 \bigtriangleup d_2$ is in $\mathcal{D}$.

▮

Note that because alphabetic variants of terms are not distinguished, each element of an instantiation set is closed under variable renaming.

Strictly speaking, item (3) above is stronger than necessary, since it suffices to have a least element in $\mathcal{D}$, with respect to set inclusion, that contains $d_1 \bigtriangleup d_2$. For the purposes of this paper, we use the stronger definition given above because it simplifies the notation and proofs slightly. Our results extend in a straightforward way to the more general case.

Instantiation sets turn out to be relatively straightforward to construct:

PROPOSITION 3.5. *Any finite set of sets of terms* $\mathbf{S} \subseteq \wp(\mathsf{Term})$ *can be extended to an instantiation set* $\mathcal{D}_S$.

Proof Let var be the set of variables of the language under consideration. Given any finite set of sets of terms $\mathbf{S}$, let $\widehat{\mathbf{S}} = \mathbf{S} \cup \{\emptyset, \mathsf{any}, \mathsf{var}\}$. Given any set of terms

T, let $T^\diamond$ denote its closure under unification, and consider the sequence of sets $\mathbf{S}_{<i>}, i \geq 0$, defined as follows:

$$\mathbf{S}_{<0>} = \{S^\diamond \mid S \in \widehat{\mathbf{S}}\}; \text{ and}$$
$$\mathbf{S}_{<i>} = \{S_1 \bigtriangleup S_2 \mid S_1, S_2 \in \mathbf{S}_{<i-1>}\}, \quad i > 0.$$

Finally, define the set $\mathcal{D}_S$ as $\mathcal{D}_S = \cup_{i \geq 0} \mathbf{S}_{<i>}$. Clearly, each element of $\mathbf{S}_{<0>}$ is closed under unification, whence from Proposition 3.2, so is each element of $\mathbf{S}_{<i>}$ for each $i \geq 0$, and therefore, so is each element of $\mathcal{D}_S$. Since $\emptyset^\diamond = \emptyset$, $\mathsf{any}^\diamond = \mathsf{any}$, and $\mathsf{var}^\diamond = \mathsf{var}$, it follows that the elements $\emptyset, \mathsf{any}$, and var are in $\mathbf{S}_{<0>}$, and hence in $\mathcal{D}_S$; it is trivial to show that $\mathsf{var} \in \mathcal{D}_S$ is the greatest element of $\mathcal{D}_S$ with respect to $\trianglelefteq$. By construction, for any pair of elements $d_1, d_2 \in \mathcal{D}_S$, their meet $d_1 \bigtriangleup d_2$ is also in $\mathcal{D}_S$. Thus, to show that $\mathcal{D}_S$ is an instantiation set, it remains only to establish that it is finite. For this, we show that for any $n \geq 0$, any element $S \in \mathbf{S}_{<n>}$ can be represented as $S = \bigtriangleup\mathbf{X}$ for some $\mathbf{X} \subseteq \mathbf{S}_{<0>}$. The base case for this, with $n = 0$, is trivial. Assume that the claim holds for all $n < k$, and consider $S \in \mathbf{S}_{<k>}$. From the definition, $S = S_1 \bigtriangleup S_2$ where both S_1 and S_2 are in $\mathbf{S}_{<k-1>}$. From the inductive hypothesis, we can write $S_1 = \bigtriangleup\mathbf{A}$ and $S_2 = \bigtriangleup\mathbf{B}$, where $\mathbf{A}, \mathbf{B} \subseteq \mathbf{S}_{<0>}$. Thus, $S = (\bigtriangleup\mathbf{A}) \bigtriangleup (\bigtriangleup\mathbf{B})$. Since $\mathbf{S}_{<0>}$ is finite, both $\mathbf{A}$ and $\mathbf{B}$ must be finite: let these sets be $\mathbf{A} = \{A_1, \ldots, A_r\}$ and $\mathbf{B} = \{B_1, \ldots, B_s\}$. Then,

$$\begin{aligned} S &= \{A_1 \bigtriangleup \ldots \bigtriangleup A_r\} \bigtriangleup \{B_1 \bigtriangleup \ldots \bigtriangleup B_s\} \\ &= \{A_1 \bigtriangleup \ldots \bigtriangleup A_r \bigtriangleup B_1 \bigtriangleup \ldots \bigtriangleup B_s\}, \qquad \text{from the associativity of } \bigtriangleup \\ &= \bigtriangleup(\mathbf{A} \cup \mathbf{B}) \end{aligned}$$

and since both $\mathbf{A}$ and $\mathbf{B}$ are subsets of $\mathbf{S}_{<0>}$, so is $\mathbf{A} \cup \mathbf{B}$, so the claim holds for all $n \geq 0$. It follows that any element d of $\mathcal{D}_S$ can be represented as $d = \bigtriangleup\mathbf{X}$ for some $\mathbf{X} \subseteq \mathbf{S}_{<0>}$. This implies that the size of $\mathcal{D}_S$ cannot exceed the size of the powerset of $\mathbf{S}_{<0>}$. Since $\widehat{\mathbf{S}}$ is finite, and $\mid \mathbf{S}_{<0>} \mid = \mid \widehat{\mathbf{S}} \mid$, the powerset of $\mathbf{S}_{<0>}$ is also finite, whence $\mathcal{D}_S$ is finite. The proposition follows. □

A notion of considerable importance in the development that follows is that of *substitution-closure*:

DEFINITION 3.4. An instantiation set $\mathcal{D}$ is said to be *substitution-closed* if, for every $d \in \mathcal{D}$, if t is a term in d and σ is any substitution, then $\sigma(t)$ is also in d. ▮

Given a complete lattice $\langle L, \sqsubseteq \rangle$ and $a \in L$, the set $L(a) = \{x \in L \mid x \sqsubseteq a\}$ is called a *principal ideal* of L. Then, it is not difficult to see that the following holds:

PROPOSITION 3.6. *An instantiation set $\mathcal{D}$ is substitution-closed if and only if every nonempty element of $\mathcal{D}$ is a principal ideal of the term lattice $\langle \mathsf{Term}, \sqsubseteq \rangle$.* □

PROPOSITION 3.7. *If $\mathcal{D}$ is substitution-closed, then for any d_1 and d_2 in $\mathcal{D}$, $d_1 \trianglelefteq d_2$ if and only if $d_1 \subseteq d_2$.*

Proof [if] Let $\mathcal{D}$ be substitution-closed, and consider d_1 and d_2 in $\mathcal{D}$ such that $d_1 \subseteq d_2$. Since every element of d_1 is, trivially, an instance of itself, this implies that every element of d_1 is an instance of some element of d_2. Suppose $\mathsf{s\_unify}(d_1, d_2) \neq d_1$: in this case, either (i) there is some element t_1 in $\mathsf{s\_unify}(d_1, d_2)$ that is not in d_1; or (ii) there is some element t_1' in d_1 that is not in $\mathsf{s\_unify}(d_1, d_2)$. In case (i), it follows from the definition of $\mathsf{s\_unify}$ that t_1 must be an instance of some element of d_1, and since $\mathcal{D}$ is substitution-closed, t_1 must be in d_1, which is a contradiction. In case (ii), since $d_1 \subseteq d_2$, t_1' is also in d_2; since $unify(t_1', t_1') = t_1'$, it follows that t_1' is also in $\mathsf{s\_unify}(d_1, d_2)$, which is again a contradiction. This establishes that if $d_1 \subseteq d_2$ then $\mathsf{s\_unify}(d_1, d_2) = d_1$, i.e. $d_1 \unlhd d_2$.

[only if] Let $\mathcal{D}$ be substitution-closed, and consider $d_1, d_2 \in \mathcal{D}$ such that $d_1 \unlhd d_2$, i.e. $\mathsf{s\_unify}(d_1, d_2) = d_1$. Suppose $d_1 \not\subseteq d_2$: then, there must be some element t in d_1 that is not in d_2. But $\mathsf{s\_unify}(d_1, d_2) = d_1$, so t is in $\mathsf{s\_unify}(d_1, d_2)$, which means that t must be an instance of some element of d_2. Since $\mathcal{D}$ is substitution-closed, it follows that t must also be in d_2, which is a contradiction. Hence $d_1 \subseteq d_2$. □

Recall that a *Moore family* of subsets of a set S is a family of subsets of S that contains S and is closed under intersection [4]. Moore families are important in the context of abstract interpretation because they admit "best" descriptions for sets of values [13].

PROPOSITION 3.8. *A substitution-closed instantiation set forms a Moore family.*

Proof Let $\mathcal{D}$ be a substitution-closed instantiation set. By definition, the set of all terms is an element of $\mathcal{D}$. From Propositions 3.2 and 3.7, it follows that for any d_1 and d_2 in $\mathcal{D}$, $\mathsf{s\_unify}(d_1, d_2) = d_1 \bigtriangleup d_2 = d_1 \cap d_2$. By definition, $d_1 \bigtriangleup d_2$ is in $\mathcal{D}$, whence $\mathcal{D}$ is closed under intersection. Thus $\mathcal{D}$ forms a Moore family. □

Given any set of terms, it is necessary to specify how to find its *instantiation*, i.e. the element of $\mathcal{D}$ that "describes" it best. This is given by the *instantiation function* ι. Recall that a closure operator f on a set is one that is extensive (i.e. $x \subseteq f(x)$), monotonic (i.e. $x \subseteq y$ implies $f(x) \subseteq f(y)$) and idempotent (i.e. $f(f(x)) = f(x)$). Then, we have:

DEFINITION 3.5. An *abstraction structure* is a pair $\langle \mathcal{D}, \iota \rangle$, where $\mathcal{D} \subseteq \wp(\mathsf{Term})$ is an instantiation set, and $\iota : \wp(\mathsf{Term}) \longrightarrow \mathcal{D}$ is a closure operator on $\langle \wp(\mathsf{Term}), \subseteq \rangle$. ▮

The requirement that ι be a closure operator with respect to set inclusion follows from considerations of abstract interpretation [12]. If $\mathcal{D}$ is also closed under intersection, the instantiation function ι can be formulated in an especially simple way:

PROPOSITION 3.9. *If the instantiation set $\mathcal{D}$ is closed under intersection, then the function ι, defined by*

$$\iota(T) \quad = \quad \cap\{d \in \mathcal{D} \mid T \subseteq d\}$$

is a closure operator on $\langle \wp(\mathsf{Term}), \subseteq \rangle$. □

EXAMPLE 3.1. Consider the abstraction structure $\langle \mathcal{D}, \iota \rangle$, where $\mathcal{D} = \{\emptyset, \mathbf{int}, \mathbf{c}, \mathbf{clist}, \mathbf{list}, \mathbf{intlist}, \mathbf{nv}, \mathsf{any}\}$, where **int** represents the set of integers, **c** the set of ground terms, **list** the set of lists, **clist** the set of ground lists, **intlist** the set of lists of integers, **nv** the set of nonvariable terms and any the set of all terms. Since $\mathcal{D}$ is substitution-closed and closed under intersection, the instantiation function ι can be defined as $\iota(T) = \cap\{d \in \mathcal{D} \mid T \subseteq d\}$, and $\mathsf{s\_unify}(d_1, d_2)$ can be defined as $d_1 \cap d_2$. □

The execution of a logic program induces an association, at each program point in a clause, between the variables in the clause and the sets of terms they can be instantiated to at that point. The behavior of a program may therefore be summarized by specifying, for each such program point, the set of terms each variable in that clause can be instantiated to at when execution reaches that point. However, while the set of terms a variable can be instantiated to at any point in a program at runtime can be arbitrarily large, compile-time representations of program behavior must be finite. We therefore use the elements of instantiation sets to describe the set of terms a variable can be instantiated to at runtime.

When a clause is selected for resolution against a goal, its variables are renamed so that it is variable-disjoint with the goal. Consider a use of clause C in a computation where the variables of C have been renamed via a renaming substitution σ: we refer to this as a *σ-activation* of C. The finite set of variable names $\mathbf{V}_C$ appearing in a clause C are referred to as the *program variables* of C. The set of terms a variable can be instantiated to at any point in a program is described using instantiation states ("i-states" for short):

DEFINITION 3.6. An *instantiation state* A_C at a program point in a clause C is a mapping

$$A_C : \mathbf{V}_C \longrightarrow \mathcal{D}$$

such that for any variable v in $\mathbf{V}_C$, if $\sigma(v)$ can be bound to a term t at that program point in any σ-activation of C in any execution of the program, then $t \in A_C(v)$. ▮

Note that because of the assumption that each argument in the head of a clause is a variable, the set of variables occurring in a clause is always nonempty, whence the domain of the instantiation states of any clause is also always nonempty. 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, we drop the subscript C from the names of i-states. If the i-state at a point in a clause is A, and $A(v) = d$ for some program variable v of the clause, then d is referred to as the *instantiation* of v at that point. Since each variable in a clause C is in its least instantiated, i.e. most general, state at the beginning of execution of that clause, before its head has been unified with the arguments in the call, the corresponding "initial i-state" for C, where each variable v in $\mathbf{V}_C$ is mapped to $\top_{inst}$, is denoted by A_C^{init}. The mapping defined by an i-state A extends naturally to arbitrary terms and tuples of terms t:

(1) if t is a constant c, then $A(t) = \iota(\{c\})$;
(2) if t is a compound term $f(t_1, \ldots, t_n)$, then $A(t) = \iota(\{f(u_1, \ldots, u_n) \mid u_i \in A(t_i), 1 \leq i \leq n\})$;
(3) if t is a tuple $\langle t_1, \ldots, t_n \rangle$ then $A(t) = \langle A(t_1), \ldots, A(t_n) \rangle$.

The instantiation of a tuple of terms is referred to as its *instantiation pattern*, or "i-pattern" for short. The instantiation pattern of the arguments of a call to a predicate is referred to as the *calling pattern* for the call, while that at the return from a call is referred to as the *success pattern* for the call. An i-pattern $\langle d_1, \ldots, d_n \rangle$ *describes* a tuple of terms $\langle t_1, \ldots, t_n \rangle$ if and only if $t_i \in d_i, 1 \leq i \leq n$.

The set of i-states of a clause inherits the semilattice structure of the instantiation set $\langle \mathcal{D}, \trianglelefteq \rangle$, and is itself a meet-semilattice, where the ordering $\trianglelefteq$ on $\mathcal{D}$ is extended to i-states in the obvious way: for any two i-states A_1, A_2 for a clause C, $A_1 \trianglelefteq A_2$ if and only if $A_1(v) \trianglelefteq A_2(v)$ for every program variable of C. The set of i-states, ordered in this manner, is referred to as the *abstract domain* of the clause. The abstract domain of a program is given by the disjoint sum of the abstract domains for its clauses. For simplicity in the discussion that follows, we typically consider abstract domains for individual clauses: the extension to the abstract domain for the entire program is a straightforward construction involving the usual injection and projection operators.

3.3 Abstracting Unification

Consider a variable x occurring in a term t_1, whose instantiation in the i-state under consideration is d_1, and assume that t_1 is being unified with a term t_2 whose instantiation is d_2. Consider the effect of this unification on the instantiation of x. The instantiation of the resulting term is $d = d_1 \triangle d_2$. If $x \equiv t_1$, then the instantiation of x after unification must also be d; on the other hand, if x is a proper subterm of t_1, then it will have become instantiated to some proper subterm of the term resulting from the unification; this can be expressed using the following:

DEFINITION 3.7. Given a set of terms t, let ST be the set of all proper subterms of all elements of t, then, $sub\_inst(T)$ is defined to be the instantiation $\iota(ST)$ of ST. ▮

The instantiation "inherited" by a variable x occurring in a term during unification is given by the function *inherited_inst*, which can be defined as follows:

DEFINITION 3.8. Let t_1 be a term whose instantiation is d_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 d_2 is given by

$$\begin{aligned} inherited\_inst(x, t_1, d_1, d_2) \quad &= \quad \textbf{if } x \equiv t_1 \textbf{ then } d \textbf{ else } sub\_inst(d), \\ &\quad\quad \textbf{where } d = d_1 \triangle d_2. \end{aligned}$$

▮

Since instantiation sets are finite by definition, the function *sub_inst* can be represented as a finite table.

EXAMPLE 3.2. Consider the abstraction structure of Example 3.1. The function *sub_inst* is defined as follows:

$$\begin{aligned} sub\_inst = \{&\emptyset \mapsto \emptyset, \textbf{int} \mapsto \emptyset, \textbf{c} \mapsto \textbf{c}, \textbf{list} \mapsto \mathsf{any}, \textbf{clist} \mapsto \textbf{c}, \textbf{intlist} \mapsto \textbf{c}, \\ &\textbf{nv} \mapsto \mathsf{any}, \mathsf{any} \mapsto \mathsf{any}\}. \end{aligned}$$

Suppose a term $f(X)$ is being unified with a ground term, where the variable X is uninstantiated in the i-state A under consideration, i.e. $A(X) = \mathsf{any}$. The instantiation of the term $f(X)$ is $\textbf{nv}$. The instantiation of x after unification is given by

$$\begin{aligned} &inherited\_inst(X, f(X), \textbf{nv}, \textbf{c}) \\ &\quad = sub\_inst(\textbf{nv} \triangle \textbf{c}) \\ &\quad = sub\_inst(\textbf{c}) \\ &\quad = \textbf{c}. \end{aligned}$$

Thus, we infer that x becomes ground after the unification. □

This definition has to be engineered slightly to deal more precisely with predefined recursive types such as lists: for example, it should be possible to infer that the tail of a list is also a list. The modifications necessary to build knowledge about the structure of such recursive types into *inherited_inst* are conceptually straightforward: e.g., to deal with the element **intlist** in Example 3.1, the function *inherited_inst* can be extended as follows:

```
$inherited\_inst(x, t_1, d_1, d_2)$ = let $d = d_1 \triangle d_2$ in
    if $x \equiv t_1$ then $d$;
    else if $(is\_list(t_1) \wedge (d_1 = \mathbf{intlist} \vee d_2 = \mathbf{intlist}))$ then
        $list\_inherited\_inst(x, t_1, d_1, d_2)$;
    else $sub\_inst(d)$.
```

Here, *list_inherited_inst* is used to express knowledge about the structure of terms in **intlist**:

```
$list\_inherited\_inst(x, t_1, d_1, d_2)$ =
    let $d_1'$ = if $d_1 = \mathbf{intlist}$ then if $hd\_subterm(x, t_1)$ then int else intlist;
              else $sub\_inst(d_1)$;
        $d_2'$ = if $d_2 = \mathbf{intlist}$ then if $hd\_subterm(x, t_1)$ then int else intlist;
              else $sub\_inst(d_2)$;
    in $d_1' \triangle d_2'$.

$hd\_subterm(x, t)$ =
    if $t$ is a variable then false;
    else if $x = head(t)$ then true else $hd\_subterm(x, tail(t))$.
```

Now consider a tuple of terms $\bar{t} = \langle t_1, \ldots, t_n \rangle$ in an i-state A_0 for a clause C. Let $A_0(\bar{t})$ be $\langle d_{11}, \ldots, d_{1n} \rangle$. Consider the unification of $\bar{t}$ with another n-tuple of terms described by an i-pattern $\bar{I} \equiv \langle d_{21}, \ldots, d_{2n} \rangle$. Let x be a variable in $\mathbf{V}_C$. If x occurs in the k^{th} element t_k of $\bar{t}$, then the instantiation d' of x resulting from the unification of t_k with the term represented by the corresponding element d_{2k} of $\bar{I}$ is given by $d' = inherited\_inst(x, t_k, d_{1k}, d_{2k})$. Suppose x occurs in the m^{th} element t_m of $\bar{t}$ as well: arguing as above, the instantiation of x resulting from the unification of the m^{th} elements of the two tuples is $d'' = inherited\_inst(x, t_m, d_{1m}, d_{2m})$. The resulting instantiation of x must therefore be $d' \triangle d''$. Extending this argument to multiple occurrences of a variable is straightforward. Further, if the resulting instantiation of any variable is $\emptyset$, then this indicates that unification has failed, so the instantiation of every variable in $\mathbf{V}_C$ can be taken to be $\emptyset$. For any variable v, let $occ(v, \bar{t}) = \{j \mid v\ occurs\ in\ t_j\}$ be the indices of the elements of $\bar{t}$ in which v occurs. Then, given an i-state A, a tuple of terms $\bar{t}$ and an i-pattern $\bar{I}$, we can define a function *update_i_state* that gives the i-state resulting from unifying $\bar{t}$ with any tuple of terms described by $\bar{I}$, as follows:

DEFINITION 3.9. Let A be an i-state defined on a set of variables $\mathbf{V}$, and let $\bar{t} = \langle t_1, \ldots, t_n \rangle$ be a tuple of terms all whose variables are in $\mathbf{V}$. Let $\bar{I} = \langle d_1, \ldots, d_n \rangle$ be an i-pattern, and let A'' be the intermediate i-state defined as follows: for every variable v in $\mathbf{V}$,

$$A''(v) \quad = \quad \textbf{if } occ(v,\bar{t}) = \emptyset \textbf{ then } A(v) \\ \textbf{else } \triangle\{inherited\_inst(v, t_j, A(t_j), d_j) \mid j \in occ(v,\bar{t})\}.$$

Then, $A' = update\_i\_state(A, \bar{t}, \bar{I})$ is defined as follows: for every variable v in $\mathbf{V}$,

$$A'(v) \quad = \quad \textbf{if } A''(v) = \emptyset \text{ for any variable in } \mathbf{V} \textbf{ then } \emptyset \\ \textbf{else } A''(v).$$

▮

EXAMPLE 3.3. Consider the abstraction structure of Example 3.1, and the i-state

$$A_0 : \{M \mapsto \mathsf{any}, E \mapsto \mathsf{any}, L \mapsto \mathsf{any}, U1 \mapsto \mathsf{any}, U2 \mapsto \mathsf{any}\}.$$

Let $\bar{t} = \langle M, [E|L], [E|U1], U2\rangle$ and $\bar{I} = \langle \textbf{int}, \textbf{intlist}, \mathsf{any}, \mathsf{any}\rangle$. Then, $A_1 = update\_i\_state(A_0, \bar{t}, \bar{I})$ is obtained as follows: The variable M occurs only in the first position of $\bar{t}$, i.e. $occ(M, \bar{t}) = \{1\}$, so

$$A_1(M) = inherited\_inst(M, M, \mathsf{any}, \textbf{int}) = \mathsf{any} \triangle \textbf{int} = \textbf{int}.$$

Since $occ(E, \bar{t}) = \{2, 3\}$, $A_1(E) = \triangle\{d_1, d_2\}$, where

$d_1 = inherited\_inst(E, [E|L], \textbf{nv}, \textbf{intlist}) = \textbf{int}$; and
$d_2 = inherited\_inst(E, [E|U1], \textbf{nv}, \mathsf{any}) = \mathsf{any}$.

Thus, $A_1(E) = \textbf{int} \triangle \mathsf{any} = \textbf{int}$. Similarly, $A_1(L) = inherited\_inst(L, [E|L], \textbf{nv}, \textbf{intlist}) = \textbf{intlist}$.

Notice that in inferring the updated instantiations of E and L in this example, we have implicitly assumed that *inherited_inst* has been extended to handle list structures, as discussed above. The instantiations of the remaining variables can be worked out in a similar manner. The i-state A_1 is then obtained as

$$A_1 : \{M \mapsto \textbf{int}, E \mapsto \textbf{int}, L \mapsto \textbf{intlist}, U1 \mapsto \mathsf{any}, U2 \mapsto \mathsf{any}\}.$$

□

4. Analysis of Sequential Logic Programs

4.1 Propagating Flow Information

Given a class of queries that the user may ask of a program, not all the different calling patterns that are possible for a predicate may in fact be encountered during computations. Similarly, given a calling pattern for a predicate, only certain success patterns actually correspond to computations in the program starting with a call described by that calling pattern. With each predicate p in a program, therefore, we associate sets of admissible calling and success patterns, defined as follows:

DEFINITION 4.1. Given a predicate p in a program P, the set of *admissible calling patterns* $\texttt{CALLPAT}(p) \subseteq \mathcal{D}^n$, and the set of *admissible success patterns* $\texttt{SUCCPAT}(p) \subseteq \mathcal{D}^n \times \mathcal{D}^n$, are defined to be the smallest sets satisfying the following:

—If p is an exported predicate and I is a calling pattern for p in the class of queries specified by the user, then I is in $\texttt{CALLPAT}(p)$.

—Let q_0 be a predicate in the program, $I_c \in \texttt{CALLPAT}(q_0)$, and let there be a clause in the program of the form

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

Let the i-state at the point immediately after the literal $q_j(\bar{u}_j), 0 \le j \le n$, be A_j, where

—$A_0 = \mathit{update\_i\_state}(A^{init}, \bar{u}_0, I_c)$, where A^{init} is the initial i-state of the clause;

—$cp_i = A_{i-1}(\bar{u}_i)$ is in CALLPAT(q_i), $1 \le i \le n$; and

—if $\langle cp_i, sp_i \rangle$ is in SUCCPAT(q_i), then $A_i = \mathit{update\_i\_state}(A_{i-1}, \bar{u}_i, sp_i)$; if there is no such tuple then A_i maps each variable in the clause to $\emptyset$.

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

▮

An algorithm to compute the CALLPAT and SUCCPAT sets is given in Figure 1. The global data structures maintained by the program consist of a list WORKLIST of predicates that have to be processed; and for each predicate p in the program, tables CALLPAT(p) and SUCCPAT(p). Given a program P, WORKLIST initially contains the set of predicates appearing in EXPORTS(P). If p is an exported predicate, 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 predicates that call p, i.e. those predicates q for which there is a clause in the program of the form

$q(\ldots)$:− $\ldots$, $p(\ldots)$, $\ldots$

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 by the user for the exported predicates. Given an admissible calling pattern for a predicate, i-states are propagated across each clause for that predicate as shown in Figure 2. When all the literals in the body have been processed the success pattern for that clause is obtained by determining the instantiation of the arguments in the head in the i-state after the last literal in the body. The success pattern for the predicate is then determined from the success patterns of the clauses defining it. This is repeated until no new calling or success patterns can be obtained for any predicate, at which point the analysis terminates.

In order to avoid repeatedly computing the success pattern of a predicate for a given calling pattern, an *extension table* can be used [22; 61]. This is a memo structure that maintains, for each predicate, a set of pairs $\langle$*Call*, *RetVals*$\rangle$ where *Call* is a tuple of arguments in a call and *RetVals* is a list of solutions that have been found for that (or a subsuming) call to that predicate. At the time of a call, the extension table is first consulted to see if any solutions have already been computed for it: if any such solutions are found, these are returned directly instead of repeating the computation. If the extension table indicates that the call has been made earlier but no solutions have been returned, then the second call is suspended until solutions are returned for the first one. The extension table idea can be modified in a straightforward way to deal with calling and success patterns rather than actual calls and returns. In this way, once a success pattern has been computed for a given calling pattern for a predicate, success patterns for future

Input:. A program $\langle P, \texttt{EXPORTS}(P)\rangle$.

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

Method:. Starting with the exported predicates, iterate over the program as indicated below 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 $\texttt{CALLERS}(p)$ of predicates that call p.

(2) *Initialization:* For each n-ary predicate p defined in P, create tables $\texttt{CALLPAT}(p)$ and $\texttt{SUCCPAT}(p)$, initialized to be empty.
For each predicate p mentioned in $\texttt{EXPORTS}(P)$, add p to WORKLIST; for each $\langle p, cp\rangle$ in $\texttt{EXPORTS}(P)$, add cp to $\texttt{CALLPAT}(p)$.

(3) *Analysis*:

```
while WORKLIST not empty do
    let $p$ be an element of WORKLIST;
    WORKLIST := $WORKLIST \setminus \{p\}$;
    for each $cp \in \texttt{CALLPAT}(p)$ do
        for each clause $C$ of $p$ do
            analyse_clause($C, cp$)                /* see Figure 2 */
        od
    od
od;
```

Fig. 1. Algorithm for dataflow analysis

invocations of that predicate with the same calling pattern can be obtained via table lookup. Alternatively, *magic sets* techniques may be used to compute these sets [21; 47].

4.2 Soundness

The development above has consistently ignored the possibility of variable aliasing. We now characterize the class of dataflow analyses for which this can be safely done. First we define the notion of unification-soundness, which intuitively describes when unification simulated over an instantiation set $\mathcal{D}$ correctly reflects the possible effects of actual unification at runtime:

DEFINITION 4.2. Given an abstraction structure $\langle\mathcal{D}, \iota\rangle$, let A be any i-state whose domain is $\mathbf{V}$, $\bar{t}_1$ any n-tuple of terms all whose variables are in $\mathbf{V}$, and $\bar{t}_2$ any n-tuple of terms described by an i-pattern $\bar{I}$. Let θ be any substitution such that for every variable v in $\mathbf{V}$, $\theta(v) \in A(v)$. Then, an abstract unification procedure *update_i_state* is *unification-sound* if and only if the following holds: if $\theta(\bar{t}_1)$ and $\bar{t}_2$ are unifiable with most general unifier ψ, then $A' = \mathit{update\_i\_state}(A, \bar{t}_1, \bar{I})$ is such that for every v in $\mathbf{V}$, $\psi(\theta(v)) \in A'(v)$. ▮

LEMMA 4.1. *Given an abstraction structure $\langle\mathcal{D}, \iota\rangle$, the abstract unification procedure update_i_state is unification-sound if and only if $\mathcal{D}$ is substitution-closed.*

Proof Consider an i-state A with domain $\mathbf{V}$, and let $\bar{t}_1 = \langle u_1, \ldots, u_n\rangle$ be any tuple of terms whose variables are in $\mathbf{V}$. Let $\bar{t}_2$ be any n-tuple of terms described by $\bar{I} = \langle d_1, \ldots, d_n\rangle$, and θ any substitution such that for any $v \in \mathbf{V}$, $\theta(v) \in A(v)$. If $\theta(\bar{t}_1)$ and $\bar{t}_2$ are not unifiable, the lemma holds vacuously. Assume, therefore, that

```
function analyse_pred($p$,$cp$)            /* $p$ is the predicate to be analyzed; $cp$ is the calling pattern */
begin
    if $cp \in$ CALLPAT($p$) then return $\{sp \mid \langle cp, sp\rangle \in$ SUCCPAT($p$)$\}$;
    else
        add $cp$ to CALLPAT($p$);
        for each clause $c_i$ of $p$ do $S_i$ := analyse_clause($c_i, cp$) od;
        return $\cup_i S_i$;
    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 '$p(\bar{t})$ :- $Body$';
    $\mathbf{A}_0$ := $\{update\_i\_state(A^{init}_{cl}, \bar{t}, cp)\}$;     /* head unification */
    $\mathbf{A}_n$ := analyse_body($Body$, $\mathbf{A}_0$);
    $SP$ := $\{A(\bar{t}) \mid A \in \mathbf{A}_n\}$;                 /* success patterns for the clause */
    $NEW\_SP$ := $\{\langle cp, sp\rangle \mid sp \in SP \wedge \langle cp, sp\rangle \notin$ SUCCPAT($p$)$\}$;
    if $NEW\_SP \neq \emptyset$ then
        add $NEW\_SP$ to SUCCPAT($p$);
        add CALLERS($p$) to WORKLIST;
    fi;
    return $SP$;
end.

function analyse_body($Body$, $\mathbf{A}$)        /* $Body$ is the body of a clause $C$; $\mathbf{A}$ is a set of i-states of $C$ */
begin
    if $Body$ is empty then return $\mathbf{A}$;
    else
        let $Body$ be of the form '$q(\bar{u}), BodyTail$';
        $\mathbf{A}'$ := $\emptyset$;
        for each $A \in \mathbf{A}$ do
            $cp$ := $A(\bar{u})$;                    /* a calling pattern for $q(\bar{u})$; */
            $S$ := analyse_pred($q, cp$);         /* success patterns for $q(\bar{u})$ */
            for each $sp \in S$ do $A'$ := $A' \cup update\_i\_state(A, \bar{u}, sp)$ od
        od;
        return analyse_body($BodyTail, \mathbf{A}'$);
    fi;
end.
```

Fig. 2. The functions **analyse_pred**, **analyse_clause**, and **analyse_body**.

$\theta(\bar{t}_1)$ and $\bar{t}_2$ are unifiable with most general unifier ψ.
[if] Suppose that $\mathcal{D}$ is substitution-closed. Let x be any variable in $\mathbf{V}$, then the binding of x after unification is given by $\psi(\theta(x))$. Let $A' = \mathit{update\_i\_state}(A, \bar{t}_1, \bar{I})$. There are two possibilities:

(1) If x does not occur in $\bar{t}_1$, then $A'(x) = A(x)$. Since $\theta(x) \in A(x)$, and $\mathcal{D}$ is substitution-closed, it follows that $\sigma(\theta(x)) \in A(x)$ for any substitution σ, whence $\psi(\theta(x)) \in A'(x)$.

(2) If x occurs in $\bar{t}_1$, let $occ(x, \bar{t}_1)$ be the indices of the elements of $\bar{t}_1$ in which it occurs. From the definition of *inherited_inst*, it follows that there is some term $s_i \in \mathit{inherited\_inst}(x, u_i, A(u_i), d_i)$ such that $\psi(\theta(x)) \sqsubseteq s_i$, for each i in $occ(x, \bar{t}_1)$, where $\sqsubseteq$ is the subsumption order on terms. This implies that $\{\psi(\theta(x))\} \trianglelefteq \mathit{inherited\_inst}(x, u_i, A(u_i), d_i)$. In other words, $\{\psi(\theta(x))\}$ is a lower bound on $\mathit{inherited\_inst}(x, u_i, A(u_i), d_i)$, for each i in $occ(x, \bar{t}_1)$. Since
$$A'(x) = \triangle_{i \in occ(x,\bar{t}_1)} \mathit{inherited\_inst}(x, u_i, A(u_i), d_i)$$
is their greatest lower bound, we have
$$\{\psi(\theta(x))\} \trianglelefteq \triangle_{i \in occ(x,\bar{t}_1)} \mathit{inherited\_inst}(x, u_i, A(u_i), d_i).$$
Since $\mathcal{D}$ is substitution-closed, it follows, from Proposition 3.7, that $\{\psi(\theta(x))\} \subseteq A'(x)$, i.e. $\psi(\theta(x)) \in A'(x)$.

[only if]: Assume that $\mathcal{D}$ is not substitution-closed. This means that there is some element $d \in \mathcal{D}$, and some term $\hat{t} \in d$, such that for some substitution σ, $\hat{t}' \equiv \sigma(\hat{t}) \notin d$. Consider $\mathbf{V} = \{x, y\}$, where neither x nor y occur in $\hat{t}$; let $\bar{t}_1 = \langle x, x, y \rangle$ and $\bar{I} = \langle d', \top_{inst}, \top_{inst} \rangle$ such that $t' \in d'$, where $d' \in \mathcal{D}$. Consider the tuple $\bar{t}_2 = \langle \hat{t}', v, v \rangle$, where v is a variable not occurring in $\hat{t}$: clearly, $\bar{I}$ describes $\bar{t}_2$. Let A be an i-state such that $A(y) = d$, and let θ be the substitution $\theta = \{y \mapsto \hat{t}\}$. Consider the substitution $\sigma' : \{x \mapsto \hat{t}, v \mapsto \hat{t}\}$. The reader may easily verify that $\sigma \circ \sigma'$ is the most general unifier of the tuples $\theta(\bar{t}_1)$ and $\bar{t}_2$. Let $A' = \mathit{update\_i\_state}(A, \bar{t}_1, \bar{I})$, then

$$\begin{aligned} A'(y) &= \triangle_{i \in occ(y,\bar{t}_1)} \mathit{inherited\_inst}(y, t_i, A(t_i), d_i) \\ &= \mathit{inherited\_inst}(y, y, d, \top_{inst}) \\ &= d \triangle \top_{inst} \\ &= d. \end{aligned}$$

However, $(\sigma \circ \sigma')(\theta(y)) = \sigma(\theta(y)) = \sigma(\hat{t}) = \hat{t}'$, which is not in d. It follows that *update_i_state* is not unification-sound. □

This result can be strengthened if further restrictions are placed on the class of programs being considered, e.g. if we assume, as in the case of deductive database programs, that calls always succeed with all arguments instantiated to ground terms.

Define a *flow analysis procedure* to be any algorithm that computes the sets `CALLPAT` and `SUCCPAT` defined above, and call such an analysis procedure *complete* if, for any user-specified calling pattern, every computation that can arise from a goal described by that calling pattern is considered for analysis by the flow analysis procedure. We now show that complete flow analysis procedures are sound for approximation domains that are substitution-closed. To this end, we show that for any predicate p in a program, if p can be called with arguments C_p, then there is some calling pattern I_c in CALLPAT(p) that "describes" C_p (possibly conservatively), i.e. $\iota(C_p) \subseteq I_c$, and similarly for success patterns:

DEFINITION 4.3. A flow analysis procedure over an abstraction structure $\langle \mathcal{D}, \iota \rangle$ is *sound* if and only if, for every program P, the sets of calling and success patterns CALLPAT and SUCCPAT inferred by the analysis satisfy the following: for every exported predicate p, if $\bar{cp}$ is a calling pattern for an external query, then

(1) if $q(\bar{t})$ is a call that arises in some computation in the program starting from a query described by $p(\bar{cp})$, then there is a tuple I_c in CALLPAT(q) such that $\iota(\{\bar{t}\}) \subseteq I_c$; and
(2) if the call $q(\bar{t})$ can succeed with its arguments bound to a tuple $\bar{t}'$, then there is a pair $\langle I_c, I_s \rangle$ in SUCCPAT(q) such that $\iota(\{\bar{t}\}) \subseteq I_c$ and $\iota(\{\bar{t}'\}) \subseteq I_s$.

▮

THEOREM 4.2. *A flow analysis procedure is sound if and only if it is complete and the abstract unification procedure update_i_state is unification-sound.*

Proof [*only if*:] If the flow analysis procedure is not complete, then some executions that arise at runtime are not considered during analysis, and it is easy to see that such execution paths can give rise to calling and success patterns that do not appear in the CALLPAT and SUCCPAT tables. If *update_i_state* is not unification-sound then there exists a tuple of terms $\bar{t}$, and a tuple of terms $\bar{t}'$ described by an i-pattern $\bar{I}$, such that *update_i_state* does not correctly describe the unification of $\bar{t}$ and $\bar{t}'$. It follows that given the program consisting of the single clause '$p(\bar{t})$' and user-specified calling pattern $\bar{I}$, the SUCCPAT relation computed by the flow analysis procedure will not be sound.

[*if*:] The proof is by induction on the number of resolution steps n in the computation. Let p be a predicate in a program. Consider a call to p with arguments $\bar{t}_c$. In the base case, $n = 0$, and $\bar{t}_c$ must be a query from the user. By definition, there is a calling pattern I_c in CALLPAT(p) such that $\bar{t}_c \in I_c$, i.e. $\iota(\{\bar{t}_c\}) \subseteq I_c$. The base case for success pattern requires at least one resolution step, and occurs when there is a unit clause $p(\bar{u})$ for p such that $\bar{t}_c$ unifies with $\bar{u}$ resulting in the tuple of terms $\bar{t}_s$. In this case, the i-state after unification of the call with the head is given by $A_0 = update\_i\_state(A^{init}, \bar{u}, \iota(\{\bar{t}_c\}))$ and the success pattern I_s is $A_0(\bar{u})$. From Lemma 4.1, it follows that $\iota(\{\bar{t}_s\}) \subseteq I_s$, and that $\langle I_c, I_s \rangle$ is in SUCCPAT(p).

For the inductive step, assume that the theorem holds for $n < k$, and consider a call $p(\bar{t}_c)$ derived in k resolution steps, $k > 0$. It must be the case that the program contains a clause

$$r(\bar{u}_0) \ :- \ q_1(\bar{u}_1), \ldots, q_j(\bar{u}_j), \ldots, q_n(\bar{u}_n)$$

where q_j is p, and there is a call $r(\bar{t}_r)$ in that computation for the predicate r. Clearly, the call $r(\bar{t}_r)$ must have taken fewer than k steps of computation, whence from the induction hypothesis, there is a calling pattern I_r in CALLPAT(r) such that $\iota(\{\bar{t}_r\}) \subseteq I_r$. For each of the literals $q_i(\bar{u}_i), 1 \le i < j$, the call and return require fewer than k steps of computation, so that if the calls and corresponding returns are given by $\bar{t}_{q_i}$ and $\bar{t}'_{q_i}$, then there is some $cp_{q_i} \in$ CALLPAT(q_i) such that $\iota(\{\bar{t}_{q_i}\}) \subseteq cp_{q_i}$, and some $\langle cp_{q_i}, sp_{q_i} \rangle$ in SUCCPAT(q_i) such that $\iota(\{\bar{t}'_{q_i}\}) \subseteq sp_{q_i}$. Let

A_i be the i-state immediately before the literal $q_i(\bar{u}_i)$, $1 \leq i \leq j$, then it is an easy induction on i, using Lemma 4.1, to show that A_i gives a sound description of variable instantiations at the corresponding program point. From Lemma 4.1, it follows that there is a calling pattern cp_p in $\texttt{CALLPAT}(p)$ such that $\iota(\{\bar{t}_c\}) \subseteq cp_p$. A similar argument holds for success patterns. □

COROLLARY 4.3. *A flow analysis procedure over an abstraction structure $\langle \mathcal{D}, \iota \rangle$ is sound if and only if it is complete and $\mathcal{D}$ is substitution-closed.*

Proof From Lemma 4.1 and Theorem 4.2. □

That terminating flow analysis procedures exist follows from the fact that from the definition of *update_i_state*, the generality of variables is always nonincreasing. Since $\mathcal{D}$ is by definition finite, any variable can have only finitely many different instantiations during analysis, so each predicate can have only finitely many calling and success patterns, and it is easy to see that these can be computed in finite time.

A consequence of the substitution-closure requirement, however, is that the instantiation set can no longer have any element representing only uninstantiated variables: these must be represented by the set of all terms, any. In other words, the top element $\top_{inst}$ in $\langle \mathcal{D}, \trianglelefteq \rangle$ will necessarily be any. This has the disadvantage of losing some expressive power, e.g. we can no longer reason about dependencies between literals. This can lead to a loss of precision in some cases.

4.3 Complexity

Before discussing the worst case complexity of computing the sets **CALLPAT** and **SUCCPAT**, it is necessary to consider how much time it takes to determine whether two elements of the instantiation set $\mathcal{D}$ are "equal". Given an instantiation set $\mathcal{D}$, let $\Psi(\mathcal{D})$ denote the time required, in the worst case, to determine whether two arbitrary elements of $\mathcal{D}$ are equal. For example, if the elements of $\mathcal{D}$ are atomic constants, as in [15; 40; 44], then $\Psi(\mathcal{D}) = O(1)$; if they are trees of size at most n and equality is modulo variable renaming, as in [58], then $\Psi(\mathcal{D}) = O(n)$; if they are rational terms, i.e. terms that may contain "back edges", then $\Psi(\mathcal{D}) = O(n)$ if equality refers to isomorphism, while $\Psi(\mathcal{D}) = O(n\alpha(n))$, where α is the slow-growing pseudo-inverse of Ackermann's function, if two rational terms are considered equal if they denote the same infinite tree [33]; if they are regular expressions, and two expressions are considered equal if they denote the same language, then $\Psi(\mathcal{D})$ is likely to be exponential (since the problem of deciding whether two regular expressions do not denote the same language is PSPACE-complete [25]).

Let the *size* of a term be the total number of symbols in the term, i.e. the number of nodes in the tree representation of the term. 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. Let the number of variables in any clause be at most V. Suppose that the number of calling patterns for any predicate, and the number of success patterns for any given calling pattern, is at most K. The initial preprocessing to construct the call graph of the program and determine the sets $\texttt{CALLERS}(p)$ for each predicate p can be done in time linear in the size of the program.

First, consider the processing of a single clause for a single calling pattern for a predicate p. This involves (1) looking up $\texttt{CALLPAT}(p)$ to determine if the calling pattern has been processed already; (2) if it has, returning the associated success patterns; and (3) if it has not been processed before, propagating dataflow information across the body of the clause, evaluating the possible success patterns, installing them into SUCCPAT(p), and returning these success patterns. The cost of accessing CALLPAT(p) in step (1) requires $O(\log K)$ comparisons of tuples of size a (but note that hashing can be used to give $O(1)$ comparisons on the average). Since each such comparison costs $\Psi(\mathcal{D})$, elementwise comparisons of tuples of size a costs $a\Psi(\mathcal{D})$, and the cost of looking up each calling pattern is $O(\log K \cdot a\Psi(\mathcal{D}))$. To facilitate efficient access, SUCCPAT(p) can be maintained as a data structure that associates a collection of success patterns with each calling pattern (so that, conceptually, SUCCPAT(p) for an n-ary predicate p is a mapping from $\mathcal{D}^n$ to $\wp(\mathcal{D}^n)$), and each calling pattern in CALLPAT(p) can have an associated pointer into the corresponding success patterns in SUCCPAT(p), whence Step (2) can be performed in $O(1)$ time by simply returning this pointer.

We now consider the cost of Step (3), the propagation of dataflow information across the body of a clause. Let the size of any argument to a literal be at most s. The cost of computing the instantiation of a term from an i-state may involve computing the instantiation of each subterm, and hence in the worst case is $O(s)$. To update an i-state to reflect the unification of a clause head with a call or the return from a call, it is necessary to consider as many variables as occur in the tuple of terms involved, which can be $O(V)$. Since each literal in the body of a clause can give rise to at most K success patterns, the processing of each clause involves at most $O(lK)$ computations of i-patterns, whose cost is $O(lK \cdot as)$, and $O(lK)$ applications of *update_i_state*, whose cost is $O(lK \cdot V)$. For each body literal, the cost of actually computing and returning the success patterns corresponding to each of its calling patterns will be charged to the predicate for that literal. The cost of evaluating the body of the clause, therefore, is $O(lK(as + V))$. After the body has been evaluated, each success pattern generated for the clause must be added to SUCCPAT(p) if it is not already there: as in the case of Step (2), the relevant entries in SUCCPAT(p) can be found in $O(1)$ time, after which each success pattern can be looked up and installed (if necessary) in time logarithmic in the number of success patterns associated with the calling pattern under consideration. Since the clause can give rise to at most $O(K)$ success patterns, each of size a, the total cost of installing success patterns into SUCCPAT(p) is $O(K \log K \cdot a\Psi(\mathcal{D}))$. Thus, the total cost of processing a clause for a single calling pattern is

$$\begin{aligned} &O(K \cdot a\Psi(\mathcal{D}) + lK \cdot (as + V) + K \log K \cdot a\Psi(\mathcal{D})) \\ &= O(lK(as + V) + K \log K \cdot a\Psi(\mathcal{D})). \end{aligned}$$

By using extension tables, each calling pattern is processed exactly once, and each success pattern for a calling pattern is computed exactly once. The cost of processing a clause K calling patterns, therefore, is $O(lK^2(as + V) + K^2 \log K \cdot a\Psi(\mathcal{D}))$. If the program contains p predicate definitions, each consisting of no more than c clauses, then the total cost of processing the program is $O(pclK^2(as + V) + pcK^2 \log K \cdot a\Psi(\mathcal{D})) = O(pcla(s + V)K^2 \log K \cdot a\Psi(\mathcal{D}))$. Let $N = pcla \cdot \max(s, V)$ be the size of the program, then the time complexity of the algorithm is $O(N \cdot \Psi(\mathcal{D}) \cdot K^2 \log K)$.

Let $d = \mid \mathcal{D} \mid$ be the size of the instantiation set, then the number of calling and success patterns for any predicate, in the worst case, may be $O(d^a)$, giving a worst case time complexity of $O(N \cdot \Psi(\mathcal{D}) \cdot d^{2a} \log d^a)$. 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, predicates in a program are used with specific sets of arguments consistently instantiated in one way; indeed, the plausibility of flow analysis rests on this fact. The second reason is that the arities of predicates in a program usually do not increase as the size of the program increases. For most programs encountered in practice, therefore, the number of calling and success patterns for any predicate can usually be bounded by a (small) constant, i.e., $K = O(1)$. For such programs, or for dataflow analysis problems where the size of $\mathcal{D}$ is fixed beforehand and hence $O(1)$, the complexity of the algorithm reduces to $O(N \cdot \Psi(\mathcal{D}))$. The procedure is therefore asymptotically optimal for most programs encountered in practice.

The role played by substitution-closure is that by allowing aliasing effects to be ignored, it allows i-patterns to be computed in time $O(as)$, and *update_i_state* to be applied in time $O(V)$. This does not preclude the possibility of having analyses that do not use substitution-closed abstraction structures, but maintain sharing and dependency information between variables in a way that allows them to obtain the same asymptotic complexities for computing i-patterns and applying *update_i_state*: such analyses would attain the same overall complexity as the procedure discussed here. It follows that substitution-closure is sufficient for obtaining analysis algorithms whose time complexity is usually linear in program size, but it is not clear whether it is a necessary condition.

5. Analysis of Parallel Logic Programs

The discussion so far has focussed on sequential control strategies. Since orderings between the clauses of a predicate are ignored, pure OR-parallel execution strategies can be accommodated in this framework without any problems. The situation is different for AND-parallel programs, where the body literals in clauses are no longer totally ordered. This section extends the approach of the previous section to deal with an arbitrary partial order $\prec$, called the *control order*, on the body literals of a clause:

DEFINITION 5.1. A *control order* for a clause H :− B is a partial order $\prec$ over its literals such that $H \prec L$ for every literal L occurring in the body B. ∎

Intuitively, given literals L_1 and L_2 in the body of a clause, a control order relation $L_1 \prec L2$ can be understood as specifying that the execution of L_1 precedes that of L_2, i.e. the execution of L_2 begins after that of L_1 has finished. A control strategy for a program specifies a control ordering for each clause in the program. In general, a control strategy may associate different control orders with different calling patterns to a clause. We assume that the control strategy is specified beforehand to the flow analysis system. This is not entirely unreasonable, because compilers for parallel logic programming languages usually have some expectation of how execution may proceed. In independent AND-parallel systems [11; 28; 37], the order of execution is determined statically using a data dependency analysis of the program, while in various committed choice languages [10; 59; 63], it is possible to infer something about the relative order of execution of literals within a clause, based on annotations, guards, etc. Of course, when applied to programs of a particular language, additional knowledge about the language can be used to augment the analysis presented here and improve its precision [8; 23; 62]. We make no assumptions regarding program annotations, about whether or not the language is committed-choice, or regarding the independence of literals executing in parallel.

Because of the few assumptions made, it is possible to handle different execution models within one framework. For example, programs dealing with large databases that combine top-down and bottom-up execution can be handled, as can programs that use stream parallelism, where "producer" and "consumer" goals share variables and execute in parallel. However, this complicates matters, because in addition to aliasing, communication and synchronization between goals becomes an issue. Synchronization in parallel logic programming languages is accomplished primarily by specifying mechanisms for goals to suspend when unification attempts to bind certain kinds of uninstantiated variables. Our main result here is to show that in analyses where the instantiation set is substitution-closed, communication and synchronization issues can be ignored without compromising soundness.

Given a calling pattern for a clause, the possible success patterns for it do not depend on the order in which the body literals are executed. The admissible success pattern relation $\texttt{SUCCPAT}(p)$ for a predicate p can therefore be computed as described in the previous section, using (for example) Prolog's left-to-right ordering on literals. However, admissible calling patterns depend on the particular control order. Now consider a situation where two literals p and q execute concurrently, and both these literals precede a third literal r. If we know the i-states before p and q, we can compute their calling patterns. Since admissible success patterns are independent of the control order, we can compute their success patterns as described in Section 4, and thus the i-state that would be obtained after each of the literals p and q *if it were executing alone*. Our task, therefore, is to compose these i-states to obtain an i-state that gives a sound description of the variable bindings seen by the literal r whose execution follows those of p and q.

To deal with control orders that may not be total orders, two i-states are associated with each literal L in a clause: $\mathcal{I}_{(-)}(L)$, called the *pre-state* of L, gives the i-state immediately before L is evaluated; $\mathcal{I}_{(+)}(L)$, called the *post-state* of L, gives the i-state that would be obtained immediately after the execution of L if there were no other literals executing concurrently. The definitions of admissible calling and success patterns are analogous to the sequential case:

DEFINITION 5.2. Given a predicate p in a program P, the set of admissible calling patterns $\texttt{CALLPAT}(p) \subseteq \mathcal{D}^n$ and the set of *admissible success patterns* $\texttt{SUCCPAT}(p) \subseteq \mathcal{D}^n \times \mathcal{D}^n$, are defined to be the smallest sets satisfying the following:

—If p is an exported predicate and I is a calling pattern for p in the class of queries specified by the user, then I is in $\texttt{CALLPAT}(p)$.

—Let q_0 be a predicate in the program, $I_c \in \texttt{CALLPAT}(q_0)$, and let there be a clause C in the program of the form

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

Let the initial i-state of C be A^{init}. The pre-state of the head of C is A^{init}, while its post-state is given by $\mathcal{I}_{(+)}(q(\bar{u}_0)) = \mathit{update\_i\_state}(A^{init}, \bar{u}_0, I_c)$.

The pre- and post-states for the literals in the body of C are obtained as follows: for any literal $L \equiv p(\bar{t})$ in the body of C, let $pred(L)$ be the set of immediate predecessors of L with respect to the control order $\prec$ associated with the clause C corresponding to the calling pattern I_c. Then,

—the pre-state $\mathcal{I}_{(-)}(L)$ is given by $\mathcal{I}_{(-)}(L) = \triangle_{r \in pred(L)} \mathcal{I}_{(+)}(r)$;
—the calling pattern for L is $cp = \mathcal{I}_{(-)}(L)(\bar{t})$, and cp is in CALLPAT(p); and
—if $\langle cp, sp \rangle \in$ SUCCPAT(p), then $\mathcal{I}_{(+)}(L) = update\_i\_state(\mathcal{I}_{(-)}(L), \bar{t}, sp)$; if there is no such tuple then $\mathcal{I}_{(+)}(L)$ maps each variable in the clause to $\emptyset$.

Given the control ordering $\prec$ for C associated with the calling pattern I_c, let $fin(C)$ denote the set of literals L that have no "successors" with respect to $\prec$, i.e. for which there is no L' such that $L \prec L'$. Let A_{fin} denote the i-state $A_{fin} = \triangle_{r \in fin(C)} \mathcal{I}_{(+)}(r)$. The success pattern for C is given by $I_s = A_{fin}(\bar{u}_0)$, and $\langle I_c, I_s \rangle$ is in SUCCPAT(q_0).

▮

The computation of the sets CALLPAT and SUCCPAT proceeds iteratively, as before, until there is no change to either set. The soundness of this approach hinges on the soundness of using the meet operation $\triangle$ on i-states to compose the i-states obtained individually from a set of literals that may have executed concurrently. To reason about this, it is necessary to formalize the notion of executing one literal "by itself". Given a control order $\prec$ for a clause C and a literal L in its body, let $L^{\Uparrow}$ denote the reflexive transitive closure of the predecessor relation over the body of C with respect to $\prec$: $L^{\Uparrow}$ gives the set of literals in the body of C that have to be executed in order to enable the execution of L to finish. This extends to sets of literals as follows: given a set of literals $\mathbf{L}$, $\mathbf{L}^{\Uparrow} = \cup_i \{L_i^{\Uparrow} \mid L \in \mathbf{L}\}$.

DEFINITION 5.3. Given a clause C and a set of literals S occurring in its body, $C|_S$, the *restriction* of C to S, is a clause defined as follows: the head of $C|_S$ is the head of C; the body of $C|_S$ is the set of literals S; and if the control order for C associated with a calling pattern is $\prec$, then the control order $\prec_S$ for $C|_S$ associated with that calling pattern given by the following: $s_1 \prec_S s_2$ for any two literals $s_1, s_2 \in S$ if and only if $s_1 \prec s_2$. ▮

The following lemma states that given a set of literals $\{L_1, \ldots, L_n\}$, if the post-states $\mathcal{I}_{(+)}(L_i)$ give sound descriptions variable instantiations after the execution of L_i "by itself", $1 \leq i \leq n$, then $\triangle_{i=1}^{n} A_i$ gives a sound description of variable bindings resulting from executing $\{L_1, \ldots, L_n\}$ concurrently if the instantiation set $\mathcal{D}$ is substitution-closed; in other words, that the composition of i-states resulting from individual executions using $\triangle$ is sound when the instantiation set is substitution-closed. Since synchronization has not been considered anywhere in this development, this gives a broad characterization of analyses for which synchronization issues can be ignored without compromising soundness.

LEMMA 5.1. *Consider the analysis of a clause C, with control order $\prec$, over an instantiation set $\mathcal{D}$. Let $\{L_1, \ldots, L_n\}$ be any set of literals in the body of C. Corresponding to each $L_i, 1 \leq i \leq n$, let A_i be an i-state such that for every program variable v of C, if $\sigma(v)$ can be bound to a term t after the execution of L_i for some σ-activation of $C|_{L_i^{\Uparrow}}$, then $t \in A_i(v)$. Let $A = \triangle_{i=1}^{n} A_i$ and $L^{\Uparrow} = \{L_1, \ldots, L_n\}^{\Uparrow}$.*

If $\mathcal{D}$ is substitution-closed, then for any $v \in \mathbf{V}_C$, if $\sigma(v)$ can be bound to a term t after some σ-activation of $C|_{L\Uparrow}$, then $t \in A(v)$.

Proof From the fact that the bindings of any variable obtained by executing a set of goals is independent of the order in which goals are executed. □

The notion of soundness is as in the sequential case. Then, we have:

THEOREM 5.2. *A flow analysis procedure over an instantiation set $\mathcal{D}$ is sound if and only if it is complete and $\mathcal{D}$ is substitution-closed.*

Proof Similar to that of Theorem 4.2. □

Termination follows from the facts that (*i*) since $\mathcal{D}$ is finite, each predicate can have only finitely many calling and success patterns; (*ii*) the instantiations of variables is nondecreasing, so that each program variable in a clause can pass through only finitely many different instantiations during analysis; and (*iii*) since each clause has a finite number of literals, there can be only finitely many control orders for each clause. The asymptotic worst case complexity for this case is the same as for the sequential case, since given an instantiation set $\mathcal{D}$ of size d, both cases involve at most d^a possible calling and success patterns. This implies, in particular, that for most programs the complexity of the analysis is $O(N \cdot \Psi(\mathcal{D}))$, where N is the size of the program and $\Psi(\mathcal{D})$ the worst case time complexity of comparing two elements of $\mathcal{D}$.

It is interesting to compare this approach to other proposals for dataflow analysis of parallel logic programs. Schemes for data dependency analysis to detect AND-parallelism have been proposed by Chang et al. [7], Jacobs and Langen [31], and Muthukumar and Hermenegildo [50]. These analyses have to explicitly keep track of dependencies between variables, and therefore are liable to be less efficient than the approach proposed here. They also presuppose Prolog's left-to-right ordering on literals within a clause, and restrict themselves to independent AND-parallelism, and are therefore less flexible than our approach. On the other hand, our assumption of substitution-closure means that we are unable to reason about dependencies between variables, rendering our instantiation sets less expressive in some cases. Gallagher et al. consider the static analysis of concurrent logic programs ignoring synchronization issues [24]. Codognet et al. describe two analysis algorithms for concurrent logic programs that take synchronization into account explicitly [8]: one of these considers every possible sequential interleaving of a set of concurrently executing agents, while the other uses a monotonicity property of the abstract domain to avoid considering every possible interleaving. However, the authors do not discuss the complexity of either algorithm.

6. Non-Noetherian Abstract Domains

An important requirement of static analyses is that they are expected to terminate, regardless of whether or not the program being analyzed would terminate when executed. Unfortunately, concrete computational domains are usually infinite, and static inference of nontrivial program properties recursively unsolvable, making it necessary to give some kind of finitely computable approximation to the desired information. This is usually done by imposing finiteness requirements on the abstract domain. The simplest – and strongest – such requirement is that the abstract domain contain only finitely many elements. Many commonly encountered static analyses for logic programs impose this requirement. This requirement can be weakened to allow an abstract domain to be infinite, provided that it is of *finite height*, i.e. its height is finite and bounded; that it satisfies the *finite chain property*, i.e. every chain in the abstract domain is finite; or that it is *Noetherian*, i.e. there are no infinite descending chains.[1]

There is, in general, a correlation in static analyses between the size of the abstract domain and the precision of analysis: in particular, the lengths of chains determine the "gaps" between successive approximations computed as an analysis iterates to a fixpoint. The larger the abstract domain, the more precise the analyses tend to be; in the limit, when the abstract domain coincides with the concrete domain, the analysis gives exact results. Thus, one would expect an analysis working with an abstract domain containing infinite chains to be more precise than analyses using the finite abstract domains considered in the previous sections. The problem, of course, is ensuring termination. This section takes a step towards accommodating infinite chains in the abstract domain without compromising uniform termination. It uses *depth abstraction* to ensure that at most finitely many points in the abstract domain are considered during analysis.

The elements of an instantiation set can be given names: e.g. in Example 3.1 above, '**c**' is the name of the set of ground terms of the language under consideration. Let d be any such name for an element in an i-set, and let $\mu(d)$ stand for the *denotation* of d, i.e. the element of the instantiation set that d names. Where there is no scope for confusion in the discussion that follows, we will sometimes not distinguish between a name d and its denotation $\mu(d)$. It is possible to consider a set of such names $\mathbf{N}$ for the elements of a instantiation set $\mathcal{D}$ as the constants of a first order language. Then, given a set of function symbols $\mathbf{F}$, ground terms over $\mathbf{F}$ and $\mathbf{N}$ can be interpreted as denoting sets of terms. Extending the notion of denotations μ in the natural way, we have

$$\mu(f(d_1, \ldots, d_n)) = \{f(t_1, \ldots, t_n) \mid t_i \in \mu(d_i), 1 \leq i \leq n\}$$

for any n-ary function symbol f in $\mathbf{F}$. This leads us to the notion of extended instantiation sets:

Definition 6.1. Suppose that $\mathbf{F}$ is the set of function symbols of the language under consideration. Given an instantiation set $\mathcal{D}$, the *extended instantiation set* $\mathcal{D}^\star(\mathbf{F})$ is defined to be the least set satisfying the following:

(1) if $d \in \mathcal{D}$ then $d \in \mathcal{D}^\star(\mathbf{F})$;
(2) if f is an n-ary function symbol in $\mathbf{F}$, $n \geq 0$, and $d_1, \ldots, d_n \in \mathcal{D}^\star(\mathbf{F})$ such that $d_i \neq \emptyset, 1 \leq i \leq n$, then $f(d_1, \ldots, d_n) \in \mathcal{D}^\star(\mathbf{F})$.

$\mathcal{D}$ is said to be the *underlying instantiation set* of the extended instantiation set $\mathcal{D}^\star(\mathbf{F})$. ▮

Where it is not necessary to refer explicitly to the set of function symbols $\mathbf{F}$ involved, the extended instantiation set will sometimes be written simply as $\mathcal{D}^\star$.

[1]Some developments of dataflow analyses are based on the dual characterization of join-semilattices, in which case no infinite ascending chains are permitted.

The ordering $\trianglelefteq$ on the elements of an instantiation set, together with the meet $\triangle$, extend in the natural way to extended instantiation sets, which therefore form meet-semilattices under this ordering.

EXAMPLE 6.1. Let $\mathcal{D}$ be as in Example 3.1, and let $\mathbf{F} = \{a/0, f/1, g/1\}$. Then, $\mathcal{D}^\star$ is the set

$$\mathcal{D} \cup \{a, f(\mathbf{int}), g(\mathbf{int}), f(f(\mathbf{int})), f(g(\mathbf{int})), \ldots, f(\mathbf{c}), g(\mathbf{c}), f(f(\mathbf{c})), \ldots\}$$

□

If the set of function symbols $\mathbf{F}$ contains any symbol with arity greater than 0, then for any instantiation set $\mathcal{D}$, $\langle \mathcal{D}^\star(\mathbf{F}), \trianglelefteq \rangle$ contains infinite descending chains. To see this, suppose $\mathbf{F}$ contains a unary function symbol g. Then, the extended instantiation set, ordered by $\trianglelefteq$, contains the infinite chain

$$\mathsf{any},\ g(\mathsf{any}),\ g(g(\mathsf{any})),\ g(g(g(\mathsf{any}))), \ldots$$

It is possible to consider abstractions of extended instantiation sets, i.e., homomorphic images where a (possibly infinite) number of different elements are identified. Extended instantiation sets can contain infinite chains even when subjected to such abstractions, e.g. consider an instantiation set $\mathcal{D}$, and let the set of function symbols contain the empty list $nil/0$ and the list constructor '$\cdot$'/2. For any such extended instantiation set, consider a homomorphism that identifies lists of the same type, i.e. maps the elements

$$\{nil,\ \chi \cdot nil,\ \chi \cdot \chi \cdot nil, \ldots\}$$

for each $\chi \in \mathcal{D}$, into a single point $\mathbf{list}(\chi)$. In this case the "abstracted" extended instantiation set still contains infinite chains of the form

$$\mathsf{any},\ \mathbf{list}(\mathsf{any}),\ \mathbf{list}(\mathbf{list}(\mathsf{any})), \ldots$$

Though not considered explicitly, the techniques and results of this paper apply to such "abstracted" extended instantiation sets as well.

The analysis schemes described in Sections 4 and 5 can be guaranteed to terminate if the abstract domain contains no infinite chains. In this case, straightforward bottom-up fixpoint computations, possibly augmented by memo structures to avoid redundant computation [22; 61], suffice to compute the sets `CALLPAT` and `SUCCPAT` in finite time. However, if the abstract domain contains infinite chains, this approach can no longer be guaranteed to terminate. As an example, consider the program

$$p(X)\ :-\ p([X]).$$
$$?-p(0).$$

This generates an infinite sequence of pairwise incomparable calling patterns

$$\langle \mathbf{int} \rangle,\ \langle \mathbf{list}(\mathbf{int}) \rangle,\ \langle \mathbf{list}(\mathbf{list}(\mathbf{int})) \rangle, \ldots$$

Thus, the sets of admissible calling and success patterns cannot be computed in a finite amount of time. Extended instantiation sets contain infinite chains whenever the program contains function symbols of nonzero arity – a situation that holds in all but the simplest of cases. Stronger measures are therefore necessary to guarantee termination. To this end, we consider the notion of "depth abstractions" of elements of $\mathcal{D}^\star$:

DEFINITION 6.2. Given an abstraction structure $\langle \mathcal{D}, \iota \rangle$ and set of function symbols $\mathbf{F}$, the *depth-k abstraction* of an element d in the extended instantiation set $\mathcal{D}^\star(\mathbf{F})$, written $\mathcal{A}_k(d)$, is defined as follows:

$$\begin{aligned} \mathcal{A}_0(d) &= \iota(d); \\ \mathcal{A}_k(d) &= \textbf{if } d \in \mathcal{D} \text{ then } d; \textbf{ else } f(\mathcal{A}_{k-1}(d_1), \ldots, \mathcal{A}_{k-1}(d_n)) \\ &\qquad \textbf{where } d = f(d_1, \ldots, d_n). \ [\ k > 0\ ] \end{aligned}$$

▮

In the second clause of this definition, note that if d is not in $\mathcal{D}$, then from the definition of extended instantiation sets, it must be of the form $f(d_1, \ldots, d_n)$. The notion of depth abstractions extends in the natural way to i-patterns: if $I = \langle d_1, \ldots, d_n \rangle$ is an i-pattern, then $\mathcal{A}_k(I) = \langle \mathcal{A}_k(d_1), \ldots, \mathcal{A}_k(d_n) \rangle$.

PROPOSITION 6.1. *For any abstraction structure $\langle \mathcal{D}, \iota \rangle$ and any $k \geq 0$, the depth abstraction function $\mathcal{A}_k$ is a closure operator.* □

EXAMPLE 6.2. Given the extended instantiation set of Example 6.1, with underlying instantiation set $\mathcal{D}$ as in Example 3.1, we have

$$\begin{aligned} &\mathcal{A}_2(f(g(f(g(\mathsf{any}))))) \\ &\quad = f(\mathcal{A}_1(g(f(g(\mathsf{any}))))) \\ &\quad = f(g(\mathcal{A}_0(f(g(\mathsf{any}))))) \\ &\quad = f(g(\iota(f(g(\mathsf{any}))))) \\ &\quad = f(g(\mathbf{nv})). \end{aligned}$$

□

The depth of an element of $\mathcal{D}^\star$ is defined in the natural way. Consider an extended instantiation set $\mathcal{D}^\star$, with underlying instantiation set $\mathcal{D}$. For any d in $\mathcal{D}^\star$, we have

$$\begin{aligned} \mathit{depth}(d) = &\textbf{if } d \in \mathcal{D} \textbf{ then } 0; \textbf{ else } 1 + \max\{\mathit{depth}(d_1), \ldots, \mathit{depth}(d_n)\} \\ &\textbf{where } d = f(d_1, \ldots, d_n). \end{aligned}$$

As before, if d is not in $\mathcal{D}$, then it must be of the form $f(d_1, \ldots, d_n)$. This notion extends to i-patterns as follows: if $I = \langle d_1, \ldots, d_n \rangle$ is an i-pattern, then $\mathit{depth}(I) = \max\{\mathit{depth}(d_1), \ldots, \mathit{depth}(d_n)\}$.

The idea behind depth abstraction is to provide a finite approximation to an infinite set of instantiation patterns that may arise during analysis. Since there are only finitely many literals in a program, only recursive calls can give rise to an infinite number of instantiation patterns at a program point. It suffices, therefore, to consider depth abstraction for recursive calls only. For this, the program must be preprocessed to identify recursive calls. Define the relation *calls* over predicates in a program as follows: p *calls* q if and only if either (i) there is a clause in the program of the form

$$p(\ldots) \ :- \ \ldots, \ q(\ldots), \ \ldots$$

or (ii) if there is a predicate r such that p calls r and r calls q. Each literal in the body of a clause in the program is associated with a bit, called the "recursion bit", that says whether or not it is a recursive call: given a clause

$$p(\bar{u}) \ :- \ q_1(\bar{u}_1), \ldots, q_n(\bar{u}_n)$$

the recursion bit of the i^{th} body literal $q_i(\bar{u}_i)$, denoted by ρ_i, is set to **true** if q_i calls p, and to **false** otherwise.

The definitions of the admissible calling and success pattern sets are very similar to those in Sections 4 and 5, except for the incorporation of depth abstraction functions δ_C and δ_S that operate on the `CALLPAT` and `SUCCPAT` tables respectively. Thus, for the sequential case we have

DEFINITION 6.3. Given a predicate p in a program P, the set of admissible calling patterns $\texttt{CALLPAT}(p) \subseteq \mathcal{D}^n$ and the set of *admissible success patterns* $\texttt{SUCCPAT}(p) \subseteq \mathcal{D}^n \times \mathcal{D}^n$, are defined to be the smallest sets satisfying the following:

—If p is an exported predicate and I is a calling pattern for p in the class of queries specified by the user, then I is in $\texttt{CALLPAT}(p)$.

—Let q_0 be a predicate in the program, $I_c \in \texttt{CALLPAT}(q_0)$, and let there be a clause in the program of the form

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

Let the i-state at the point immediately after the literal $q_j(\bar{u}_j), 0 \leq j \leq n$, be A_j, where

—A^{init} is the initial i-state of the clause;

—$A_0 = \mathit{update\_i\_state}(A^{init}, \bar{u}_0, I_c)$;

—for $1 \leq i \leq n$, $cp_i = \delta_C(q_i, A_{i-1}(\bar{u}_i), \rho_i)$ is in $\texttt{CALLPAT}(q_i)$;

—and if $\langle cp_i, sp_i \rangle$ is in $\texttt{SUCCPAT}(q_i)$, then $A_i = \mathit{update\_i\_state}(A_{i-1}, \bar{u}_i, sp_i)$; if there is no such tuple then A_i maps each variable in the clause to $\emptyset$.

The success pattern for the clause is given by $I_s = \delta_S(q_0, I_c, A_n(\bar{u}_0), \rho)$, where $\rho = \vee_{i=1}^{n} \rho_i$, and $\langle I_c, I_s \rangle$ is in $\texttt{SUCCPAT}(q_0)$.

∎

The definition for the case of parallel execution strategies is analogous, and is not considered separately. The depth abstraction function δ_C is defined as follows:

```
$\delta_C(p, C, \rho) =$
    **if** $(\rho = \textbf{true} \wedge \texttt{CALLPAT}(p) \neq \emptyset)$ **then** $\mathcal{A}_n(C)$
        **where** $n = \max\{depth(cp) \mid cp \in \texttt{CALLPAT}(p)\}$;
    **else** $C$.
```

In other words, whenever a recursive call is encountered for a predicate p that has been called already (i.e. $\texttt{CALLPAT}(p)$ is nonempty), the calling pattern is subjected to a depth-k abstraction, where k is the depth of the deepest calling pattern in $\texttt{CALLPAT}(p)$; nonrecursive calls are not subjected to such depth abstractions. Further, it is easy to see that if, given a calling pattern C, we have $depth(C) \leq k$, then $\mathcal{A}_k(C) = C$, so that the only calling patterns actually affected by this depth abstraction are those that are deeper than any currently in $\texttt{CALLPAT}(p)$.

The depth abstraction function δ_S is defined analogously:

```
$\delta_S(p, C, S, \rho) =$ **let** $Succ = \{sp \mid \langle C, sp \rangle \in \texttt{SUCCPAT}(p)\}$ **in**
        **if** $(\rho = \textbf{true} \wedge Succ \neq \emptyset)$ **then** $\mathcal{A}_n(S)$
            **where** $n = \max\{depth(sp) \mid sp \in Succ\}$;
        **else** $S$.
```

Note that whereas the calling pattern for a literal is subjected to depth abstraction via δ_C if it is a recursive call, the success pattern for a clause is subjected to depth abstraction via δ_S if any of the literals in the body of that clause is recursive. The basic idea is similar to Cousot's notion of *widening* [14]. In Cousot's development, however, any given widening operator is fixed for all programs, whereas our approach allows different programs to be treated differently. The following lemma shows that depth abstractions provide safe approximations to i-patterns:

LEMMA 6.2. *For any i-pattern I and natural number k, $\mu(I) \subseteq \mu(\mathcal{A}_k(I))$.*

Proof By induction on k. The base case uses the fact that the instantiation function ι is a closure operator, and hence extensive. □

THEOREM 6.3. *A flow analysis procedure over an extended instantiation set $\mathcal{D}^\star$ is sound if and only if it is complete and $\mathcal{D}$ is substitution-closed.*

Proof The proof of the *if* part follows the lines of Theorems 4.2 and 5.2, using Lemma 6.2 to show that replacing a calling or success pattern by a depth abstraction preserves soundness. The *only if* part is based on two observations: first, the analysis described above does not keep track of aliasing in the sequential case, and synchronization in the parallel case, so the arguments of Lemma 4.1 or Lemma 5.1, as appropriate, apply; and second, even if aliasing or synchronization information is maintained, information can be lost when depth abstraction is performed. □

We next show that the analysis terminates:

LEMMA 6.4. *For any finite set of function symbols $\mathbf{F}$ and instantiation set $\mathcal{D}$, the set $\{d \in \mathcal{D}^\star(\mathbf{F}) \mid depth(d) \leq n\}$ is finite for any finite n.*

Proof By induction on n. In the base case, the instantiation set $\mathcal{D}$ is finite by definition. The inductive case then follows from the finiteness of $\mathbf{F}$. □

THEOREM 6.5. *For any predicate p in a program, the sets* `CALLPAT`(p) *and* `SUCCPAT`(p) *contain at most finitely many elements.*

Proof Consider the first calling pattern cp encountered for an n-ary predicate p during analysis: it must have a finite depth n. It can be seen, from the algorithm for managing `CALLPAT`(p), that for any calling pattern cp' that corresponds to a recursive call and is later added to `CALLPAT`(p), $depth(cp') \leq n$. It follows from Lemma 6.4 that `CALLPAT`(p) contains at most finitely many calling patterns arising from recursive calls. Since the programs being analyzed are finite, it follows that `CALLPAT`(p) has at most finitely many entries. The argument for `SUCCPAT`(p) is similar. □

COROLLARY 6.6. *The analysis terminates.* □

7. Applications

This section describes applications of the dataflow analysis framework developed in the previous sections to two families of analyses that have attracted a significant amount of attention in the literature.

7.1 Mode Inference

Mode inference involves dataflow analysis to determine which parameters of a predicate are used as input parameters in a particular program, and which are used as output parameters. Modes can be thought of as representing prespecified sets of terms, the set of modes therefore comprising an instantiation set. Specific mode inference algorithms are described by, among others, Mellish [44; 45], Mannila and Ukkonen [40], Debray [15; 17], and Janssens and Bruynooghe [6; 34]. The algorithm given by Mellish [44; 45] uses an instantiation set that is not substitution-closed. However, this algorithm does not take aliasing into account, leading to loss of soundness [46]. The treatments given by Debray [17] and Janssens and Bruynooghe [34] take dependencies between variables into account explicitly, but this results in complex and cumbersome algorithms.

An alternative instantiation set, described in [66], is $\{\emptyset, \mathbf{c}, \mathbf{nv}, \mathsf{any}\}$ where any denotes the set of all terms; **nv** denotes the set of nonvariable terms; and **c** denotes the set of ground terms. The mode inference algorithm is given by Mannila and Ukkonen [40] is similar, except that it restricts itself to the instantiation set $\{\mathbf{c}, \mathsf{any}\}$. In either case, $\Psi(\mathcal{D}) = O(1)$, and the complexity of the analysis, for most programs, is linear in the size of the program.

Mode information finds many applications in the optimization of logic programs. For example, mode information can be used to generate specialized unification instructions where permissible [64; 65]; to detect determinacy and functionality of programs [18; 45]; to generate index structures more intelligently [29]; to reduce the amount of runtime tests in systems that support goal suspension [52]; to guide program transformation systems [16]; and in the integration of logic and functional languages [55].

7.2 Type Inference

Intuitively, types for functors and predicates represent sets of "values", which in our case are terms. Therefore, by appropriately defining the abstraction structure, it is possible to obtain various type systems for logic programming languages. There are two principal approaches to typing such languages: these describe, respectively, the sets of terms a predicate may be called with (its *calling type*), and those it may succeed with (its *success type*). Type inference systems of either kind can be obtained as instances of the flow analysis system described. Type inference of logic programs has been considered by, among others, Heintze and Jaffar [27], Janssens and Bruynooghe [34], Mishra [48], and Yardeni and Shapiro [69].

As an example, consider a type inference system based on a scheme due to Sato and Tamaki, which uses *depth abstractions* to provide finite descriptions of possibly infinite sets of terms [58]. The *depth* of a subterm in a term is defined to be the length of the path from the root of the term, in its tree representation, to the root of the subterm under consideration. The idea behind depth abstraction is the following: given some fixed $k \geq 0$, the depth-k abstraction of a term t, denoted by $\delta_k(t)$, is the term obtained by replacing subterms of t occurring at depth k by variables. For example, the depth-2 abstraction of the term $f(g(a), h(X, f(b, X)), Y)$ is $f(g(U), h(X, V), Y)$. The analysis of a program is carried out using depth-k abstractions, for some fixed k specified beforehand. Because subterms whose depths exceed the bound k are discarded during depth abstraction, analysis using a depth-k abstraction may fail to detect aliasing at depths occurring at depths greater than k. Because of this, a variable occurring in a depth-abstracted term may not correspond to a free variable at runtime. Soundness therefore requires that such variables be interpreted as denoting all possible terms.

Given a program where the maximum arity of any functor is m, the depth-n abstraction of a term is a tree whose size can be $O(m^n)$. Thus, $\Psi(\mathcal{D}) = O(m^n)$ in this case, and the complexity of the algorithm is $O(N \cdot m^n)$. As argued earlier, however, the maximum arity m is unlikely to grow as the program size increases, so that in practice, if n is fixed, the analysis usually takes time proportional to the size of the program.

A problem with using depth abstraction to provide type information is that recursive types cannot be expressed. This problem can be handled by representing types as rational terms, i.e. terms with "back edges". A scheme along these lines has been proposed by Janssens and Bruynooghe [34; 35]. One of the type systems considered here, and referred to as *rigid types*, uses substitution-closed elements. To obtain abstract domains of finite height, Janssens and Bruynooghe impose the restriction that any acyclic path starting at the root of such a rational term should contain no more than k occurrences of any particular functor, where the multiplicity bound k is a parameter that is fixed beforehand. For a multiplicity bound of k, if there are f function symbols, and the maximum arity of any function symbol in the program is m, the maximum length of any acyclic path is fk, so each element of the instantiation is a rational term containing at most m^{fk} nodes. Comparing two such terms of size n for equality takes time $O(n\alpha(n))$, where α is the pseudo-inverse of Ackermann's function [33]. It follows that $\Psi(\mathcal{D}) = O(m^{fk}\alpha(m^{fk}))$. As before, the arity m does not usually grow with the size of the program, so if k and f are fixed then the analysis takes time proportional to the size of the program in most cases. Note also that for rigid types, the techniques of Section 6 can be applied to obtain terminating analysis even when the abstract domain contains infinite chains. Because of this, the restriction on the multiplicity of functors can be removed, allowing for a more expressive type system.

Type information finds numerous applications in the optimization of logic programs, of which we list a few. In the literature on compiler optimization for traditional languages, "dead code" refers to code whose results are never used, while "unreachable code" refers to code that is never executed [1]. Detection of dead and unreachable code is straightforward using type information. To detect code that is never called, or that which execution can never succeed through, we perform a type analysis of the program. Analysis proceeds as described earlier: when it terminates, any clause whose set of calling patterns is either empty, or contains the null element $\emptyset$ in one or more positions, is never called and can safely be deleted from the program. Any clause with whose success pattern set is empty or contains the null element $\emptyset$ in one or more positions is one that execution cannot succeed through, and is a candidate for elimination as dead code. If it can be shown that none of the reachable clauses of the predicates called by this clause have any side effects, then the clause can be deleted without affecting the semantics of the program. A related use for type information is in *functor propagation*, which can be thought of as a bidirectional generalization of the notion of "constant propagation" in traditional languages, can reduce the amount of nondeterminism in programs and lead to more compact code. Sato and Tamaki also give an example where descriptions of success patterns obtained using a depth-abstraction analysis is used to transform a nondeterministic parser for a context-free language into a deterministic parser [58]. Type information can also be used to analyze "dynamic" logic programs, i.e. programs where code can be created and executed dynamically, e.g. via the use of constructs like Prolog's *assert*, *retract* and *call*, and thereby allow dataflow analysis techniques developed for "static" logic programs to programs that are dynamic [19]. The application of type information to code optimization in logic programs is discussed in [6; 20; 35; 67].

8. Related Work

There is a large body of work in static analysis of logic programs, see for example [2; 6; 7; 8; 15; 17; 18; 19; 23; 26; 27; 32; 34; 35; 44; 47; 49; 50; 57; 58; 69]. Frameworks for abstract interpretation of logic programs have been proposed by, among others, Barbuti et. al. [3], Bruynooghe [5], Corsini and Filé [9], Jones and Søndergaard [36], Kanamori and Kawamura [38], Marriott and Søndergaard [41; 42; 43], Mellish [46], Nilsson [53], and Winsborough [68]. The work of Barbuti et al. [3] and Marriott and Søndergaard [41] are fundamentally different from that presented here in that they propose "bottom-up" dataflow analyses based on various model-theoretic semantics of logic programs, whereas the development given here is a "top-down" analysis that relies on the operational behavior of programs. The developments of Bruynooghe [5], Jones and Søndergaard [36], Kanamori and Kawamura [38], and Mellish [46] resemble ours in that they, too, are concerned with "top-down" analyses. The work of Mellish, which first proposed a framework for flow analysis of logic programs, was developed in the context of an operational semantics for Prolog given in terms of execution traces. It did not associate success patterns with the corresponding call patterns, making for some loss in precision. Bruynooghe's framework is given in the context of an operational semantics for logic programs based on AND/OR trees. The framework of Kanamori and Kawamura is based on OLDT-resolution, which is essentially SLD-resolution augmented with extension tables. In contrast to these, the treatments of Jones and Søndergaard [36] and Winsborough [68] are based on denotational semantics for logic programs. Marriott and Søndergaard [42; 43] give a uniform presentation of top-down and bottop-up analyses by expressing both in terms of operations on lattices of substitutions. Bruynooghe [5], Corsini and Filé [9], Kanamori and Kawamura [38] and Nilsson [53] give algorithms for abstract interpretation of logic programs.

The fundamental difference between the various approaches given above and that described in this paper is that while most of the abovementioned research is concerned with general frameworks for a variety of dataflow analyses for logic programs, our primary concern is with analyses that are of more than theoretical interest, i.e. those that can be carried out efficiently. Thus, while most of the abovementioned works focus on various formal and semantic aspects of abstract interpretation, we strive to identify properties of flow analyses that guarantee efficient algorithms. Because of this, our analyses are not as expressive as some that have been described in the literature: for example, they are unable to reason about aliasing behaviors. This is not surprising, because it is well-known that there is a tradeoff between the computational cost of an analysis and its precision. What is significant, however, is that we can show that our algorithms are asymptotically optimal for most programs encountered in practice, and are also useful in a reasonably wide variety of contexts. Because of this, we expect these analyses to be both practically implementable, and practically useful. A work similar in spirit to ours is that of Le Charlier et al. [39], which gives a careful complexity analysis for an algorithm for static analysis of logic programs, and discusses a number of optimizations for improving its efficiency.

The tradeoff between precision and efficiency can be seen by contrasting our linear-time mode analysis algorithm with one proposed by Marriott, Søndergaard and Jones [43]. The algorithm of Marriott et al. is based on a notion called *downward closure* that is closely related to the notion of substitution closure discussed in this paper, but less restrictive: e.g. unlike our approach, it allows reasoning about certain kinds of aliasing and sharing. The algorithm of Marriott et al manipulates propositional formulae constructed from variable names appearing in the program using only the connectives $\Leftrightarrow$, $\wedge$, and $\vee$. The abstract domain is the (finite) set of such formulae, modulo logical equivalence, which is ordered by implication ($\Rightarrow$) and forms a complete distributive lattice. The analysis iteratively computes a sequence of formulae until a fixpoint is reached, i.e. until two formulae φ_1 and φ_2 are obtained on successive iterations such that φ_1 is equivalent to φ_2. However, the equivalence problem for monotone propositional formulae is known to be co-NP-complete [30], so that unless P = NP, the parameter $\Psi(\mathcal{D})$ for this algorithm is exponential in the number of variables in a clause. This implies that each iteration of the analysis of Marriott et al. can, in the worst case, take time exponential in the maximum number of variables in a clause, unless P = NP. Further, the height of their abstract domain—and hence, the number of iterations that may be necessary to attain a fixpoint—also grows (faster than linearly) with the number of variables being considered. Thus, while the analysis of Marriott et al. is more precise than ours, it is significantly more expensive.

9. Conclusions

Despite the conceptual elegance of logic programming languages, good optimizing compilers capable of sophisticated analysis and optimization are necessary if such languages are to be competitive with more traditional languages. Moreover, in order that the analysis and optimization of large programs be possible, it is necessary that such analysis algorithms be efficient. A number of problems arise in this context: aliasing effects can make analysis computationally expensive for sequential logic programming languages; synchronization problems can complicate the analysis of parallel logic programming languages; and finiteness restrictions to guarantee termination can limit the expressive power of such analyses. Our main result is to give a simple characterization of a family of flow analyses where these issues can be ignored without compromising soundness. This results in algorithms that are simple to verify and implement, and efficient in execution. Based on this approach, we describe an efficient algorithm for flow analysis of sequential logic programs, extend this approach to handle parallel executions, and finally describe how infinite chains in the analysis domain can be accommodated without losing uniform termination.

10. Acknowledgements

The observation that depth abstraction need be applied only at recursive calls in abstract domains that are not Noetherian, rather than at every call, is due to Maurice Bruynooghe; for this, and many other valuable suggestions, we are grateful. Niels Jorgensen made several very helpful comments on the material of this paper. Comments by the anonymous referees helped improve the contents and presentation of the paper substantially.

REFERENCES

A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

A. K. Bansal and L. Sterling, "An Abstract Interpretation Scheme for Logic Programs Based on Type Expression", *Proc. International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 422-429.

R. Barbuti, R. Giacobazzi and G. Levi, "A Declarative Approach to Abstract Interpretation of Logic Programs", TR-20/89, Dept. of Computer Science, University of Pisa, 1989.

G. Birkhoff, *Lattice Theory*, AMS Colloquium Publications vol. 25, 1940.

M. Bruynooghe, "A Framework for the Abstract Interpretation of Logic Programs", Research Report CW 62, Dept. of Computer Science, Katholieke Universiteit Leuven, Oct. 1987.

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

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

C. Codognet, P. Codognet and M. Corsini, "Abstract Interpretation of Concurrent Logic Languages", *Proc. North American Conference on Logic Programming*, Austin, TX, Oct. 1990.

M. Corsini and G. Filé, "The Abstract Interpretation of Logic Programs: A General Algorithm and its Correctness", Research Report, Dept. of Mathematics, University of Padova, Sept. 1988.

K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic", *ACM Transactions on Programming Languages and Systems 8*, 1 (Jan. 1986), pp. 1-49.

J. S. Conery, *Parallel Execution of Logic Programs*, Kluwer, 1987.

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

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

P. Cousot, "Semantic Foundations of Program Analysis", in *Program Flow Analysis: Theory and Applications*, eds. S. S. Muchnick and N. D. Jones, Prentice-Hall, 1981.

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

S. K. Debray, "Unfold/Fold Transformations and Loop Optimization of Logic Programs", *Proc. SIGPLAN-88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, pp. 297-307.

S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems* vol. 11, no. 3, June 1989, pp. 419-450.

S. K. Debray and D. S. Warren, "Functional Computations in Logic Programs", *ACM Transactions on Programming Languages and Systems* vol 11 no. 3, June 1989, pp. 451-481.

S. K. Debray, "Flow Analysis of Dynamic Logic Programs", *J. Logic Programming* vol. 7 no. 2, Sept. 1989, pp. 149-176.

S. K. Debray, "A Simple Code Improvement Scheme for Prolog", *J. Logic Programming* (to appear). (Preliminary version appeared in *Proc. Sixth International Conference on Logic Programming*, Lisbon, June 1988.)

S. K. Debray and R. Ramakrishnan, "Canonical Computations of Logic Programs", unpublished manuscript, Dept. of Computer Science, University of Arizona, Tucson, July 1990.

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

J. Gallagher and E. Shapiro, "Using Safe Approximations of Fixed Points for Analysis of Logic Programs", *Proc. META88, Workshop on Meta-programming in Logic Programming*, Bristol, June 1988.

J. Gallagher, M. Codish, and E. Shapiro, "Specialization of Prolog and FCP Programs using Abstract Interpretation", *New Generation Computing* vol. 6, pp. 159-186, 1988.

M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

R. Giacobazzi and L. Ricci, "Pipeline Optimizations in AND-Parallelism by Abstract Interpretation", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990, pp. 291-305.

N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, Jan 1990, pp. 197-209.

M. V. Hermenegildo, "An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs", *Proc. Third International Conference on Logic Programming*, London, July 1986. Springer-Verlag LNCS vol. 225, pp. 25-39.

T. Hickey and S. Mudambi, "Global Compilation of Prolog", *J. Logic Programming* vol. 7 no. 3, Nov. 1989, pp. 193-230.

H. B. Hunt III and R. E. Stearns, "Monotone Boolean Formulas, Distributive Lattices, and the Complexities of Logics, Algebraic Structures, and Computation Structures (Preliminary Report)", *Proc. Third Symposium on Theoretical Aspects of Computer Science*, Orsay, France, Jan. 1986, pp. 277-287. Springer-Verlag LNCS vol. 210.

D. Jacobs and A. Langen, "Compilation for Restricted AND-Parallelism", *Proc. European Symposium on Programming 1988*, Springer-Verlag LNCS vol. 300.

D. Jacobs and A. Langen, "Accurate and Efficient Approximation of Variable Aliasing in Logic Programs", *Proc. North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989, pp. 154-165.

J. Jaffar, "Efficient Unification over Infinite Terms", *New Generation Computing* vol. 2 no. 3, 1984, pp. 207-219.

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

G. Janssens, "Deriving Run-time Properties of Logic Programs by means of Abstract Interpretation", PhD Dissertation, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

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

L. V. Kale, "The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs", *Proc. Fourth International Conference on Logic Programming*, Melbourne, May 1987, pp. 616-632. MIT Press.

T. Kanamori and T. Kawamura, "Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation", Draft Report, Mitsubishi Electric Corp., Japan, 1987.

B. Le Charlier, K. Musumbu, and P. Van Hentenryck, "A Generic Abstract Interpretation Algorithm and its Complexity Analysis", Research Paper RP-90/9, Institut d'Informatique, Univ. of Namur, Belgium, 1990. To appear in *Proc. Eighth International Conference on Logic Programming*, Paris, June 1991.

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

K. Marriott and H. Søndergaard, "Bottom-Up Abstract Interpretation of Logic Programs", *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug 1988, pp. 733-748. MIT Press.

K. Marriott and H. Søndergaard, "Semantics-based Dataflow Analysis of Logic Programs", *Information Processing 89*, ed. G. Ritter, North Holland, 1989, pp. 601-606.

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

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.

C. S. Mellish, "Some Global Optimizations for a Prolog Compiler", *J. Logic Programming* vol. 2 no. 1 (Apr. 1985), pp. 43-66.

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

C. S. Mellish, "Using Specialization to Reconstruct Two Mode Inference Systems", Manuscript, Dept. of Artificial Intelligence, University of Edinburgh, Feb. 1990.

P. Mishra, "Toward a Theory of Types in Prolog", *Proc. 1984 IEEE Symposium on Logic Programming*, Atlantic City, 1984, pp. 289-298.

A. Mulkers, W. Winsborough and M. Bruynooghe, "Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990, pp. 747-764.

K. Muthukumar and M. Hermenegildo, "Determination of Variable Dependence Information at Compile Time through Abstract Interpretation", *Proc. North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989 (to appear).

E. W. Myers, "A Precise Inter-procedural Data Flow Algorithm", *Proc. Eighth ACM Symposium on Principles of Programming Languages*, 1981, pp. 219-230.

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

U. Nilsson, "A Systematic Approach to Abstract Interpretation of Logic Programs", PhD Dissertation, Dept. of Computer and Information Science, Linköping University, Sweden, 1989.

G. D. Plotkin, "A Note on Inductive Generalization", *Machine Intelligence 5*, B. Meltzer and D. Michie (eds.), Elsevier, New York, 1970, pp. 153-162.

U. S. Reddy, "Transformation of Logic Programs into Functional Programs", *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, NJ, Feb. 1984, pp. 187-196. IEEE Press.

J. C. Reynolds, "Transformational Systems and the Algebraic Structure of Atomic Formulas", *Machine Intelligence 5*, B. Meltzer and D. Michie (eds.), Elsevier, New York, 1970, pp. 135-151.

H. Søndergaard, "An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction", *Proc. European Symposium on Programming 86*, Saarbrucken, Mar. 1986.

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

E. Y. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", Technical Report CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, Feb. 1983.

M. Smythe, "Powerdomains", *J. Computer and System Sciences 16*, 1 (1978), 23-36.

H. Tamaki and T. Sato, "OLD-Resolution with Tabulation", *Proc. Third International Conference on Logic Programming*, London, July 1986. Springer-Verlag LNCS vol. 225, pp. 84-98.

S. Taylor, *Parallel Logic Programming Techniques*, Prentice-hall, 1989.

K. Ueda, *Guarded Horn Clauses*, D. Eng. Thesis, University of Tokyo, 1986.

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

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

R. Warren, M. Hermenegildo and S. K. Debray, "On the Practicality of Global Flow Analysis of Logic Programs", *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug. 1988, pp. 684-699. MIT Press.

J. L. Weiner and S. Ramakrishnan, "A Piggy-back Compiler for Prolog", *Proc. SIGPLAN-88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, pp. 288-296.

W. Winsborough, "Automatic, Transparent Parallelization of Logic Programs at Compile Time", PhD Dissertation, Dept. of Computer Science, University of Wisconsin, Madison, 1988.

E. Yardeni and E. Y. Shapiro, "A Type System for Logic Programs", in *Concurrent Prolog: Collected Papers* vol. 2, MIT Press, 1987, pp. 211-244.