# On the Complexity of Dataflow Analysis of Logic Programs

SAUMYA K. DEBRAY
The University of Arizona

It is widely held that there is a correlation between complexity and precision in dataflow analysis, in the sense that the more precise an analysis algorithm, the more computationally expensive it must be. The details of this relationship, however, appear to not have been explored extensively. This article reports some results on this correlation in the context of logic programs. A formal notion of the "precision" of an analysis algorithm is proposed, and this is used to characterize the worst-case computational complexity of a number of dataflow analyses with different degrees of precision. While this article considers the analysis of logic programs, the technique proposed, namely the use of "exactness sets" to study relationships between complexity and precision of analyses, is not specific to logic programming in any way, and is equally applicable to flow analyses of other language families.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*nonprocedural languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*complexity of proof procedures*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Complexity, program analysis, Prolog

## 1. INTRODUCTION

It is widely held that there is a correlation, in dataflow analysis, between precision and complexity: at one extreme, there are very efficient algorithms that rarely give useful information about any program; at the other extreme are highly precise algorithms that may sometimes be very expensive computationally. Most algorithms for dataflow analysis tend to fall in between these extremes: some are generally quite efficient but may fail to be very precise under some circumstances, while others tend to be fairly precise but may not always be very efficient. The details of this tradeoff, however, do not appear to have been explored extensively. Some researchers have given complexity results for specific analyses (e.g., Jones and Muchnick [1981] and Myers [1981] for imperative languages, Hudak and Young [1986] for functional languages, Aiken and Lakshman [1994] for logic programs). However, these have typically addressed the complexity of particular analyses, rather than explore the correlation between precision and complexity of dataflow analysis in a systematic way. In the logic programming context, a variety of dataflow analysis

A preliminary version of this article appeared in *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming*, Vienna, July 1992.
This work was supported in part by the National Science Foundation under grant number CCR-8901283.
Author's address: Department of Computer Science, The University of Arizona, Tucson, AZ 85721; email: debray@cs.arizona.edu.

algorithms have been proposed, with varying degrees of precision, and worst-case complexities ranging from linear time [Debray 1992] to exponential in the input size [Codish et al. 1990; Jacobs and Langen 1989; Marriott et al. 1994; Muthukumar and Hermenegildo 1989; 1991]. These results do not, however, really give us any understanding of *intrinsic* tradeoffs between precision and complexity, in the sense of how expensive any “sufficiently precise” analysis must be.

A problem that arises immediately is that of making the notion of an analysis being “sufficiently precise” sufficiently precise. For example, it is important to be able to characterize the complexity/precision results in an algorithm-independent way, in order to avoid the possibility of being misled by design deficiencies in a particular algorithm. In the next section, we offer one simple way to address this problem, namely, to examine the class of programs for which an analysis gives exact results. We do not claim that this is the only possible characterization of the precision of dataflow analysis algorithms, or even that it is the best characterization. However, it appears to be suitable for our immediate objective, which is to explore the relationship between precision and complexity. The basic approach to examining the “intrinsic complexity” of an analysis is conceptually straightforward:

(1) Identify situations under which the analysis gives up information. Presumably, a program that does not give rise to any of these situations will not suffer any loss of information, i.e., the analysis will produce “exact” results for it. By carefully examining the analysis in this way, we can characterize the class of programs for which it produces exact results.
(2) Examine the complexity of extracting exact information about programs in this class. This typically involves examining how problems with known complexity can be encoded using these programs.

The results so obtained do not depend on specific choices of data structures or algorithms, and therefore are, in some sense, “portable” across analyses. This implies, on the other hand, that our results describe what is (or is not) possible, in an abstract sense, rather than give specific complexity results for particular analysis algorithms. For example, Theorem 5.3.1 of this article implies that the worst-case complexity of any analysis whose precision matches that of a groundness analysis using the *Prop* domain, described by Marriott et al. [1994], cannot be better than exponential in the size of the program. However, this does not rule out the possibility that a particularly stupid implementation of such an analysis may have a worst-case complexity much worse than this. The overall approach is quite general, in the sense that the examination of the relation between precision and complexity of dataflow analyses using classes of programs for which analyses are exact is applicable to any language family. The approach described here can also be used to examine independent sources of complexity in an analysis algorithm separately, and study their contributions to the overall complexity of the algorithm. The basic idea is similar to that described above: to examine the complexity contribution of some aspect of an analysis—for example, keeping track of aliasing between variables—we consider programs that do not give rise to the property being considered and examine the complexity of this class. This is illustrated in Section 5.3 via groundness analysis using the *Prop* domain [Marriott et al. 1994].

From the practitioner’s perspective, complexity-theoretic results are open to the criticism that they may not say much about the "actual" behavior of an algorithm. This is certainly true of the particular complexity results given in this article. Nevertheless, we believe that theoretical worst-case complexity results are useful for practitioners. The difference between the gloomy complexity results given here and the encouraging behavior observed "in practice" by experimental studies (see, for example, Le Charlier and Van Hentenryck [1994] and Van Hentenryck et al. [1994]) is that the former say what *could* happen, while the latter describe what *does* happen. Reconciling these apparently contradictory truths requires that we specify what we mean by the phrase "in practice" much more carefully than most empirical studies usually do. An important shortcoming of purely experimental studies is that, even if we discount the effects of different hardware platforms, implementation languages, and implementor skills, we are left with the problem that it is difficult to say anything rigorous about how representative a particular collection of benchmarks is. The methodology of this article suggests, however, that we can characterize specific classes of programs in a precise way and consider the complexity and precision of analyses on programs in these classes. The question of whether or not a collection of programs is "representative" for a particular application domain is, in our opinion, easier to consider objectively and verify empirically for classes of programs characterized in this manner than it is for a more or less arbitrarily chosen set of benchmark programs. An example of this is given by Marriott and Søndergaard [1993], who show that even though dataflow analysis using the *Prop* domain is EXPTIME-complete in general (see Section 5.3), it can be done in polynomial time for programs where the number of variables in any clause is bounded. The claim that "groundness analysis using *Prop* is reasonably efficient in practice" can then be reduced to the assertion that "it is possible, in most cases, to bound the number of variables appearing in a clause," which is somewhat easier to understand and to verify empirically. Apart from this, some researchers have considered an approach to program analysis where properties of input programs are computed, not by examining them directly, but by first transforming them to other (simpler) programs and then computing exact properties of the transformed programs [Codish and Demoen 1993; Hermenegildo et al. 1992]; it seems plausible (see Giacobazzi et al. [1992]) that other analyses may also be understandable in terms of exact analysis of transformed programs. The study of programs on which analyses give exact results can give insights into the algorithmic behavior of such analyses.

The remainder of the article examines various classes of dataflow analyses of logic programs and gives characterizations of their worst-case complexities. We consider two broad classes of dataflow analyses of logic programs. The first class, which distinguishes between different function symbols and constants, is important in the context of type inference (not all of the type analyses proposed in the logic programming literature maintain enough information about relationships among different program components to meet our assumptions about precision; type analyses that keep track of relationships between different components of terms and argument positions of predicates, and therefore are relevant for our purposes, include Cortesi et al. [1994], Janssens [1990], Mulkers et al. [1994], Winsborough [1988], and Van Hentenryck et al. [1994]) and sharing analysis [Bruynooghe 1986; King 1994; Mulkers et al. 1994]. The second class, which typically does not distinguish between different function symbols but focuses instead on groundness of variables and on dependencies between them, is important in the context of compile-time parallelization and optimization of logic programs [Chang et al. 1985; Debray 1989; Jacobs and Langen 1989; Marriott et al. 1994; Muthukumar and Hermenegildo 1989; 1991; Marriott and Søndergaard 1993]. Of the various dataflow analyses we have seen proposed in the literature, the vast majority tend to belong to one of these two classes. To keep the article self-contained, various concepts and notation used in the rest of the article are given in Section 2. Sections 3 and 4 discuss complexity results for analyses in the first class discussed above, while Section 5 considers analyses of the second class. Section 6 considers the effects of bounding the amount of information maintained about different possible execution behaviors of any predicate in a program, and Section 7 concludes.

## 2. PRELIMINARIES

### 2.1 Logic Programming: An Overview

Most logic programming languages are based on a subset of the first-order predicate calculus known as Horn clause logic. We assume an acquaintance with the usual terminology of first-order predicate logic. Following convention, a nullary function symbol will be referred to as a *constant* in the remainder of the article, while the phrase *function symbol* will be used for function symbols of nonzero arity. A definite Horn clause is of the form $G_0$ :− $G_1, \ldots, G_n$ and is read declaratively as "$G_0$ *if* $G_1$ *and* … *and* $G_n$." Here, each $G_i$ is an atomic formula and has the form $p(t_1, \ldots, t_n)$, where $p$ is an $n$-ary predicate symbol, and $t_1, \ldots, t_n$ are terms. The atoms $G_1, \ldots, G_n$ constitute the *body* of the clause. A *goal* is a set of atomic formulae $G_1, \ldots, G_n$, and is read declaratively as "$\neg G_1$ *or* … *or* $\neg G_n$." A predicate definition consists of a finite number of definite clauses whose heads all have the same predicate symbol. A logic program consists of a finite set of predicate definitions. Following the syntax of Edinburgh Prolog, we write the names of variables starting with uppercase letters, and the names of nonvariable (i.e., function and predicate) symbols starting with lowercase letters. The set of variables occurring in a term (goal, clause) $t$ is denoted by $vars(t)$.

A substitution is an idempotent mapping from a finite set of variables to terms. A substitution $\sigma_1$ is said to be *more general* than a substitution $\sigma_2$ if there is a substitution $\theta$ 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 $\sigma$ such that $\sigma(t_1) = \sigma(t_2)$; in this case, $\sigma$ 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* $mgu(t_1, t_2)$ that is unique up to variable renaming.

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 a variant of a clause in $P$ that has no variable in common with any of the goals $G_0, \ldots, G_i$; and $\theta = mgu(a_i, b)$. 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 $G$ with respect to a program $P$, and let $\theta_i$ be the unifier obtained when resolving the goal $G_{i-1}$ to obtain $G_i$, $1 \leq i \leq n$; 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 it 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 $\theta$, where $\theta$ is the substitution obtained by restricting the substitution $\theta_n \circ \cdots \circ \theta_1$ 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, the computation does not terminate.

Given a Horn program $P$, let $H_P$ denote the set of all ground atoms that can be constructed from predicate and function symbols in $P$. Consider an immediate-consequence operator $T_P : \mathcal{P}(H_P) \longrightarrow \mathcal{P}(H_P)$, where $\mathcal{P}(S)$ denotes the powerset of a set $S$: given a set of atoms $R$ known to be true, $T_P(R)$ gives the set of atoms that can immediately be inferred to be true from the clauses of the program $P$, read declaratively as implications. This operator is defined as follows: for any $R \in \mathcal{P}(H_P)$,

$$T_P(R) = \{A \mid A \text{ :- } B_1, \ldots, B_n \text{ is a ground instance of a clause in } P \text{ and } \{B_1, \ldots, B_n\} \subseteq R\}.$$

It can be shown that for any program $P$, the least fixpoint $\mathit{lfp}(T_P)$ of the operator $T_P$ exists and is unique, and is given by $\mathit{lfp}(T_P) = \cup_{i \geq 0} T_P^i(\emptyset)$ [Apt and van Emden 1982]. The fixpoint semantics of a logic program $P$ is usually defined to be $\mathit{lfp}(T_P)$. For any Horn program $P$, let $SS(P) \subseteq H_P$ denote the set of ground atoms that have successful SLD-derivations in $P$: Apt and van Emden [1982] show that $SS(P) = \mathit{lfp}(T_P)$.

Most logic programming languages, in practice, allow clause bodies to contain negated goals, and extend the operational semantics to SLDNF-resolution, which deals with such goals using the *negation-as-failure* rule. The essential idea here is that the execution of a negated goal $\texttt{not}(G)$ succeeds if $G$ is a ground goal all whose SLDNF-derivations are finite and failed (for a more precise definition, see Apt and Doets [1994]). Programs in this article that contain negated goals will be assumed to be handled using this rule.

For dataflow analysis purposes, we assume sometimes that a program may additionally specify a set of "exported" predicates, possibly with descriptions of their arguments. The intent here is to restrict our attention to possible program executions starting from goals that involve only such exported predicates and where the arguments to such predicates satisfy the specified descriptions.

### 2.2 Dataflow Analysis

Dataflow analysis is concerned with the inference, at compile time, of properties that hold at different points of a program when it is executed. Such properties can be specified by associating, with each program point, the set of all "environments" that may be obtained when execution reaches that point, over all possible executions of the program (possibly starting from some given initial state of interest): such a semantics for a program is referred to as its *collecting semantics*. Unfortunately, the sets of environments associated with program points cannot be guaranteed to be finite in general, making it impractical to carry out dataflow analyses by computing the collecting semantics. Instead, the collecting semantics is approximated using "descriptions," and analyses carried out by simulating the execution of the program using these descriptions. The domain of descriptions is often referred to as the *abstract domain*. Many dataflow analyses can be formalized in terms of a framework for program analysis known as *abstract interpretation* [Cousot and Cousot 1977; 1979; Marriott et al. 1994].

For purposes of dataflow analysis, predicates in a program are often treated as procedures that are called and from which control eventually returns to the caller, so an operational characterization of logic programs that is slightly different from SLDNF-resolution turns out to be convenient. Let $p(\bar{t})$ be the resolved atom in some SLDNF-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 $\theta$, then we also say that it can *return* with its arguments bound to $\theta(\bar{t})$. The description of a call, in terms of abstract domain elements, is referred to as a *calling pattern*, while the description of a return from a call is referred to as a *success pattern* (for more precise definitions of these terms, see Debray [1992]). Since a predicate can be called from many different points in a program, there may be more than one calling pattern for it; and since it may be defined by a number of different clauses, any given calling pattern may correspond to more than one success pattern. Let $G_0, \ldots, G_n$ be an SLDNF-derivation for a goal $G$ in a program $P$, and let $\theta_i$ be the unifier obtained when resolving the goal $G_{i-1}$ to obtain $G_i$, $1 \leq i \leq n$. Let $\psi_i$ denote the substitution $\theta_i \circ \cdots \circ \theta_1$. If there are variables $x, y \in vars(G_i)$, such that $\psi_i(x) = \psi_i(y) = t$ for some $i$, $0 \leq i \leq n$, and some nonground term $t$, then $x$ and $y$ are said to be *aliased together* in $G_i$: in this case, we also say that $x$ and $y$ are *aliases* of each other.

### 2.3 On the Precision of Dataflow Analyses

A natural characterization of the precision of a flow analysis algorithm is relative to some other algorithm: the first is more precise than the second if and only if for every program and every input the results obtained using the first algorithm are uniformly more precise than those obtained from the second. Cousot and Cousot [1977; 1979] use such a characterization to show that for any given language and abstract domain the space of all possible abstract interpretations forms a complete lattice when ordered according to this notion of precision, and thereby infer the existence of a "most precise" abstract interpretation. Because this expresses the precision of an analysis in terms of that of other analysis algorithms, we refer to this notion of precision as *relative precision*.

Such a characterization is unsuitable for our needs, however, because it is not immediately obvious how it can be used to obtain insights into the computational costs associated with analyses of different degrees of precision. What we need, rather, is a way to characterize the precision of an analysis in a way that does not require any reference to any other analysis algorithm. Our approach is to consider the precision of an analysis to be given by the class of programs for which the analysis is exact, i.e., where the analysis is *sound* (everything that can happen at runtime is predicted by the analysis) and *minimal* (everything that is predicted by the analysis can, in fact, happen at runtime). We refer to the class of programs for which an analysis algorithm $A$ gives exact results as the *exactness set* of $A$. In the language of abstract interpretation, the exactness set of an analysis algorithm $A$ is the class of programs for which the concretization of the fixpoint computed by $A$ coincides with the collecting semantics of the program.[1] The following result is not difficult to see:

PROPOSITION 2.3.1. *Given two analysis algorithms $A$ and $B$, if $A$ is relatively more precise than $B$, then the exactness set of $B$ is contained in that of $A$.*

The converse, however, need not hold. To see this, consider the following pair of analyses. $A$ is an analysis that pattern-matches on its input to determine if it is the usual `append` program to concatenate two lists: if so, $A$ produces a precomputed exact result; otherwise it does not produce any information at all. $B$ is an analysis that proceeds as follows: for any input program $P$, $B$ first uses a subsidiary analysis using depth-3 abstraction to analyze $P$; if this encounters a term with depth exceeding 3, resulting in some loss of information due to depth abstraction, $B$ returns all of the information computed by the subsidiary analysis; otherwise, if no information was lost during the subsidiary analysis, $B$ discards all of the information obtained from the subsidiary analysis and produces no information at all. $A$ yields nontrivial information only for the `append` program, for which it produces exact results; $B$ produces nontrivial information about a large number of programs but does not yield exact information about any program. Thus, even though the exactness set of $A$ properly contains that of $B$, it is clear that $A$ is not relatively more precise than $B$. However, all of the counterexamples to the converse of Proposition 2.3.1 that we have been able to construct—for example, the analysis $B$ above—discard information gratuitiously and therefore appear bizarre and unnatural. Based on this, and on the analyses that we have seen proposed in the literature, we conjecture that the converse of Proposition 2.3.1 holds for all "natural" static analysis algorithms, and that exactness sets are a reasonable measure of precision for practical purposes.

## 2.4 Some Simple but Interesting Classes of Programs

Certain classes of programs are especially interesting for examining the tradeoff between precision and complexity of dataflow analyses. Intuitively, they attempt to capture assumptions that are common in dataflow analysis algorithms. The following list enumerates the main classes of programs we consider, and the motivation for considering them.

*Failure-Free Programs.* This class contains programs that do not contain failed execution branches at runtime. This class is considered to deal with the complexity and precision of dataflow analyses, such as those of Chang et al. [1985], Debray [1989], Jacobs and Langen [1989], Marriott et al. [1994], and Muthukumar and Hermenegildo [1989; 1991], that ignore the possibility of failure.

*Function-Free Programs.* This class consists of programs that do not contain any function symbols or constants, i.e., where every argument of every literal is a variable. It is motivated by a variety of groundness and alias analyses used in compile-time optimization and parallelization, such as those of Chang et al. [1985], Marriott

[1]Note that this notion of "exactness" is strictly stronger than, for example, that proposed by Mannila and Ukkonen [1987].

et al. [1994], and Muthukumar and Hermenegildo [1989; 1991], which ignore the distinction among different function symbols.

*Datalog Programs.* This class consists of programs that do not contain any function symbols of nonzero arity. It is motivated by a variety of program analyses that rely on depth abstraction (see, for example, Codish et al. [1994], King [1994], Marriott and Søndergaard [1988], Santos Costa et al. [1991], Sato and Tamaki [1984], and Taylor [1989]). Any nontrivial depth abstraction must retain information about at least the principal functors of terms, and so will not discard any information about constants. Thus, by examining programs where there are no function symbols of nonzero arity, we can study the complexity of analyses using depth abstraction.

*Recursion-Free Programs.* This class consists of programs that contain no recursion. It is motivated by the desire to study the complexity and precision of dataflow analysis algorithms that sacrifice precision at recursive calls for efficiency reasons, e.g., Debray [1989].

*Alias-Free Programs.* This consists of programs that do not, at any point during execution, alias together two distinct variables. It is motivated by the fact that tracking aliasing may be expensive, so that we may be interested in discussing the extent to which an analysis problem may be simplified by not having to track aliasing precisely (see, for example, Debray [1992]).

*Bounded-Arity Programs.* This is a family of sets of programs: for each $n \geq 0$, the class of arity-$n$ programs consists of programs where every predicate and function symbol has arity at most $n$. In analyzing the complexity of their algorithms, some researchers have argued that in most programs encountered in practice, the arities of predicates do not increase as program size increases (see, for example, Debray [1989; 1992], Sagiv and Ullman [1984], and Ullman [1988]). Bounded-arity programs are used to study the complexity/precision behavior of various dataflow analyses under this assumption.

### 2.5 Some Useful Complexity Results

It is well known that the Satisfiability problem for propositional clauses, i.e., the problem of deciding whether an arbitrary propositional clause is satisfiable is NP-complete [Cook 1971]. In addition, certain other kinds of Boolean formulae are of particular interest to us for reasoning about computational complexity. These include Quantified Boolean Formulae, Monotone Boolean Formulae, Recursive Monotone Boolean Functions, and Schönfinkel-Bernays formulae.

*Definition* 2.5.1. A quantified Boolean formula is of the form $Q_1 x_1 \cdots Q_n x_n \varphi$, where each of the $Q_i$ is one of the quantifiers $\forall$ and $\exists$, and $\varphi$ is a propositional formula over the variables $x_1, \ldots, x_n$.

We assume that the quantified variables $x_i$ are all distinct, i.e., $i \neq j$ implies $x_i \neq x_j$: this is not a serious restriction, since the formula can be processed, in linear time, to rename variables and obtain an equivalent formula satisfying this requirement. The following result is due to Stockmeyer and Meyer [1973]:

THEOREM 2.5.2. *The problem QBF of determining whether an arbitrary quantified Boolean formula is true is PSPACE-complete.*

*Definition* 2.5.3. A Boolean formula $\varphi$ is said to be *monotone* if and only if the only connectives in $\varphi$ are $\wedge$ ("and") and $\vee$ ("or").

The following result is due to Bloniarz et al. [1984]:

THEOREM 2.5.4. *The Equivalence problem for monotone Boolean formulae, i.e., the problem of deciding, given two arbitrary monotone propositional formulae $\varphi$ and $\psi$, whether $\varphi$ and $\psi$ take on the same truth value for every truth assignment, is co-NP-complete.*[2]

*Definition* 2.5.5. A recursive monotone Boolean function (RMBF) is an equation $f(x_1, \ldots, x_n) = E$, where $E$ is an expression over the variables $x_1, \ldots, x_n$ with the syntax:

$$E ::= \mathbf{true} \mid \mathbf{false} \mid x_i (1 \leq i \leq n) \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid f(E_1, \ldots, E_n).$$

An *instance* of RMBF is a pair $\langle \mathsf{eq}, \mathsf{a} \rangle$, where $\mathsf{eq}$ is an equation $\mathsf{eq} \equiv f(x_1, \ldots, x_n) = E$, and $\mathsf{a}$ is a tuple of $n$ arguments $\mathsf{a} = \langle a_1, \ldots, a_n \rangle$. Let $\mathcal{B}$ denote the Boolean domain $\{\mathbf{true}, \mathbf{false}\}$ with the ordering $\mathbf{false} \sqsubseteq \mathbf{true}$. The instance $\langle \mathsf{eq}, \mathsf{a} \rangle$ is *true* if and only if $f(a_1, \ldots, a_n) = \mathbf{true}$ in the least fixpoint of the equation $\mathsf{eq}$ on the domain of Boolean functions $\mathcal{B}^n \longrightarrow \mathcal{B}$ (with the ordering $\sqsubseteq$ extended pointwise in the usual way).

The following result is due to Hudak and Young [1986]:

THEOREM 2.5.6. *RMBF is EXPTIME-complete in the length of the instance $\langle \mathsf{eq}, \mathsf{a} \rangle$, where $EXPTIME = \cup_{c \geq 0} DTIME[2^{n^c}]$.*

*Definition* 2.5.7. A Schönfinkel-Bernays formula is a first-order formula of the form $\exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n F$, where $F$ is a quantifier-free first-order formula over the variables $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ that does not contain any function symbols of nonzero arity nor any occurrences of the equality predicate '='.

The following result is due to Lewis [1980]:

THEOREM 2.5.8. *The problem of deciding the satisfiability of an arbitrary Schönfinkel-Bernays formula is NEXPTIME-complete, where $NEXPTIME = \cup_{c \geq 0} NTIME[2^{n^c}]$.*

### 2.6 Complexity Results for Analysis of Logic Programs: A Summary

Our complexity results for dataflow analysis of logic programs may be summarized as follows:

(1) Precise analysis of programs that contain no aliasing, where there are only two distinct constants and no function symbols of nonzero arity, is EXPTIME-complete. It follows that any analysis that is precise enough to give exact results for this class of programs has a worst-case complexity that is exponential in the size of the input program. This addresses, in particular, the complexity of various analyses that use depth abstraction, e.g., Codish et al. [1994], King

[2]Actually, Bloniarz, Hunt and Rosenkrantz prove a different but equivalent result, namely that the Inequivalence problem for monotone Boolean formulae is NP-complete.

[1994], Marriott and Søndergaard [1988], Santos Costa et al. [1991], Sato and Tamaki [1984], and Taylor [1989].

The problem is PSPACE-hard if analyses are required to deliver precise results only under the additional constraint that there is no recursion.

The problem is NP-complete if, in addition to the constraints listed above, we require also that the maximum arity of any predicate symbol is $O(1)$; it remains NP-complete even if the maximum arity is restricted to 3.

(2) If we allow function symbols of nonzero arity, then precise analysis of programs that contain negation, but contain no recursion or aliasing, is NEXPTIME-hard. This addresses the complexity of analyses that attempt to treat negation in a precise way, e.g., Marriott and Søndergaard [1988; 1992], and Marriott et al. [1990].

The problem is PSPACE-hard if precise results are required only for programs that contain no negation, recursion, or aliasing, and satisfy the additional constraint that the maximum arity of predicate and function symbols is $O(1)$. It remains PSPACE-hard even if the maximum arity of any function or predicate symbol is restricted to 2, and the maximum number of literals in the body of any clause is restricted to 2. This addresses the complexity of analyses that sacrifice precision only on encountering recursion, in particular a type analysis described by Van Hentenryck et al. [1994].

(3) Precise groundness and alias analysis of programs that contain no function symbols, where there are no failed execution branches at runtime, is EXPTIME-complete. The implication is that there are no fundamental improvements to the worst-case complexity even for analyses that are careful not to incur any overhead for keeping track of whether certain execution branches may fail at runtime. This result also addresses (one aspect of) the worst-case complexity of groundness analysis using the *Prop* domain, discussed by Marriott et al. [1994].

The problem is co-NP-complete if, in addition to the above constraints, we require also that there be no recursion, and that the maximum arity of any predicate be $O(1)$; it remains co-NP-complete even if the maximum arity of any predicate is restricted to 6.

The problem remains co-NP-complete even if the number of distinct calling and success patterns per predicate is $O(1)$. This addresses the worst-case complexity of any algorithm whose precision is similar to that of a groundness analysis proposed by Codish et al. [1990].

(4) Analysis algorithms with polynomial-time worst-case complexities can be obtained if dependencies between variables can be ignored or if such dependencies can be assumed to be transitive, and if the number of distinct calling and success patterns for any predicate is $O(1)$ Debray [1989; 1992].

## 3. ANALYSIS OF PROGRAMS WITH NONZERO-ARITY FUNCTION SYMBOLS

This section considers the analysis of programs that contain function symbols of nonzero arity. Information about constants and function symbols is of interest in the context of type inference and depth abstraction.

### 3.1 General Programs

It is well known that, in general, Horn programs containing both recursion and nonzero-arity function symbols are undecidable: indeed, Tärnlund [1977] shows that even the class of alias-free Horn programs containing at most one body literal per clause, with one binary function symbol and one constant, is Turing-complete. Since any analysis algorithm capable of precise analysis of arbitrary Horn clause programs would, as a special case, be able to determine whether the success set of a program is nonempty, and thereby infer whether or not the corresponding Turing machine has a terminating computation, this immediately implies the following:

THEOREM 3.1.1. *Precise analysis of alias-free Horn programs is undecidable. The problem remains undecidable even if we restrict ourselves to programs containing at most one binary function symbol and one constant, and one body literal per clause.*

### 3.2 Recursion-Free Programs Containing Negation

The first class of decidable Horn programs we consider is the class of alias-free recursion-free programs: our intent is to investigate the complexity of dataflow analysis algorithms that strive not to sacrifice precision except when recursion or aliasing is involved. We first consider programs that contain negated goals, in part to examine the complexity of dataflow analyses that attempt to take negation by finite failure into account (e.g., see Barbuti and Martelli [1988], Marriott and Søndergaard [1988; 1992], and Marriott et al. [1990]). We show that in this case, precise dataflow analysis is NEXPTIME-hard. To this end, we first describe a procedure that, given a Schönfinkel-Bernays formula $\varphi$, generates a logic program $P_\varphi$ that can be used to establish whether $\varphi$ is satisfiable. Let $\varphi$ be the formula $\exists x_1 \cdots \exists x_k \forall x_{k+1} \cdots \forall x_m F$ where $F$ is quantifier free, and does not contain any function symbols of nonzero arity or any occurrences of the equality predicate. It is known that such a formula has a model if and only if it has a model of size at most $k$ (Lewis [1980, Section 8]). One way to check whether there is a model for $\varphi$, therefore, is to guess an interpretation of size at most $k$ and check whether it satisfies $\varphi$. Our goal, therefore, is to devise a procedure whereby, for any given Schönfinkel-Bernays formula $\varphi$, we can generate a Prolog program $P_\varphi$ that can be executed to determine whether $\varphi$ is satisfiable.

We first define a mapping $\mathcal{S}$ that takes a quantifier-free first-order formula $\varphi$, which may contain the connectives $\wedge, \vee$, and $\neg$ (it is easy to extend the definition to include other connectives), and yields a pair $\langle G, V\rangle$. Here, $G$ is a Prolog goal, and $V$ is a variable in $G$, such that $\varphi$ is true (respectively, false) if the goal $G$ succeeds with answer substitution $\{V \mapsto \mathtt{t}\}$ (respectively, $\{V \mapsto \mathtt{f}\}$). We assume that each variable $x_i$ in $\varphi$ is associated with a logic variable $\mathtt{X}_i$; each $n$-ary predicate symbol $p$ in $\varphi$ is associated with an $n+1$-ary predicate symbol $\mathtt{p}$; and each atom $A_i$ in $\varphi$ is associated with a logic variable $\mathtt{A}_i$ that represents its truth value (different occurrences of the same atom are associated with the same logic variable). The transformation $\mathcal{S}$ is defined as follows:

(1) If $\varphi$ is a variable $x$ with associated logical variable $\mathtt{X}$, then $\mathcal{S}(\varphi) = \langle \varepsilon, \mathtt{X}\rangle$, where $\varepsilon$ is the empty sequence of goals.

(2) If $\varphi$ is a constant **true**, then $\mathcal{S}(\varphi) = \langle \mathtt{X} = \mathtt{t}, \mathtt{X}\rangle$; if $\varphi$ is a constant **false**, then $\mathcal{S}(\varphi) = \langle \mathtt{X} = \mathtt{f}, \mathtt{X}\rangle$.

(3) If $\varphi$ is of the form $\phi \wedge \psi$, with $\mathcal{S}(\phi) = \langle G_1, X_1 \rangle$ and $\mathcal{S}(\psi) = \langle G_2, X_2 \rangle$, then $\mathcal{S}(\varphi) = \langle G, X \rangle$, where $G =$ '$G_1, G_2, \texttt{and}(X_1, X_2, X)$' such that $X_1 \notin \mathit{vars}(G_2)$, $X_2 \notin \mathit{vars}(G_1)$, and $X \notin \mathit{vars}(G_1) \cup \mathit{vars}(G_2)$.

(4) If $\varphi$ is of the form $\phi \vee \psi$, with $\mathcal{S}(\phi) = \langle G_1, X_1 \rangle$ and $\mathcal{S}(\psi) = \langle G_2, X_2 \rangle$, then $\mathcal{S}(\varphi) = \langle G, X \rangle$, where $G =$ '$G_1, G_2, \texttt{or}(X_1, X_2, X)$' such that $X_1 \notin \mathit{vars}(G_2)$, $X_2 \notin \mathit{vars}(G_1)$, and $X \notin \mathit{vars}(G_1) \cup \mathit{vars}(G_2)$.

(5) If $\varphi$ is of the form $\neg\phi$, with $\mathcal{S}(\phi) = \langle G_1, X_1 \rangle$, then $\mathcal{S}(\varphi) = \langle G, X \rangle$, where $G =$ '$G_1, \texttt{neg}(X_1, X)$' such that $X \notin \mathit{vars}(G_1)$.

(6) If $\varphi$ is an atom $A \equiv p(\phi_1, \ldots, \phi_n)$ with associated logic variable $\texttt{A}$, and $\mathcal{S}(\phi_i) = \langle G_i, X_i \rangle$, $X_i \neq \texttt{A}, 1 \leq i \leq n$, then $\mathcal{S}(\varphi) = \langle G, \texttt{A} \rangle$, where $G =$ '$G_1, \ldots, G_n, \texttt{p}(X_1, \ldots, X_n, \texttt{A})$'.

In the discussion that follows, we will sometimes abuse notation by applying $\mathcal{S}$ to Boolean formulae involving function applications as well, treating a function application $f(t_1, \ldots, t_n)$ similarly to an atom with $n$ arguments. These will be clear from the context, and hopefully will not cause any confusion.

*Example* 3.2.1. Let $\varphi = [r(u, v, w) \wedge s(v, w, x)] \vee \neg r(u, v, w)$. Let the logic variables corresponding to the variables $u, v, w, x$ in $S(\varphi)$ be $\texttt{U}, \texttt{V}, \texttt{W}, \texttt{X}$ respectively, and let the logic variables associated with the atoms $r(u, v, w)$ and $s(v, w, x)$ be $\texttt{A1}$ and $\texttt{A2}$ respectively. Then, $\mathcal{S}(\varphi) = \langle G, \texttt{T} \rangle$, where $G$ is the goal:

```
r(U, V, W, A1), s(V, W, X, A2), and(A1, A2, T1),
r(U, V, W, A1), neg(A1, T2),
or(T1, T2, T).
```

Recall that our immediate objective is to define a translation procedure from Schönfinkel-Bernays formulae to Prolog programs such that the exact analysis of a program $P_\varphi$ yields satisfiability information about the formula $\varphi$ it was generated from. Any such procedure must be able to "guess" bindings for the existentially quantified variables in $\varphi$ and then consider all possible bindings for the universally quantified variables. Moreover, it is necessary to associate truth values with atomic subformulae of $\varphi$. Moreover, in order that the execution of $P_\varphi$ define an interpretation for $\varphi$, it is important to ensure that such atomic formulae are assigned consistent truth values *after* bindings have been chosen for the variables. This can be accomplished by maintaining a "symbol table" associating truth values with atomic formulae. Finally, since we are interested in recursion-free programs, the programs generated by the translation procedure should not contain recursion. The use of backtracking to simulate the "guessing" of bindings for existentially quantified variables, and of negation-by-failure to consider all possible bindings for universally quantified ones, is fairly straightforward. The only part of the translation that is somewhat delicate is the generation of routines for managing the "symbol tables," since (1) a symbol table for an $n$-ary predicate in a universe with $k$ constants must accommodate $k^n$ entries, one for each possible atom for that predicate, but in order for the reduction to work we must be able to generate these routines in polynomial time; and (2) for our purposes we cannot use recursion—this precludes, for example, the simple (and textually succinct) approach of using a list representation for the table and a recursive membership-checking predicate for lookups.

For simplicity, we maintain a different symbol table for each predicate symbol in the formula. The symbol table for an $n$-ary predicate $p_i$ is structured as a tree of depth $n$. Each internal node of the tree has $k$ children, corresponding to the $k$ different choices for atoms for an argument, and the leaves give truth values. Each path from the root of the tree to a leaf traces out a particular choice of constants for each of the $n$ argument positions of the predicate, i.e., speficies a ground atom for that predicate, and the leaf node gives a truth value for that atom. Table lookups for $p_i$ are managed by a group of $n+1$ predicates $\texttt{lookup}_{i,j}$: for each $j$, $1 \leq j \leq n$, $\texttt{lookup}_{i,j}$ uses the value of the $j$th argument of an atom to select the appropriate branch in the portion of the symbol table it is given. Suppose that the universe contains $k$ constants $\mathtt{a}_1, \ldots, \mathtt{a}_k$; then these predicates are defined as follows:

```
lookup_{i,j}(Atom, Table, Tval)  :-
      Atom = p_i(A_1, ..., A_j, ..., A_n),
      A_j = a_1,
      Table = table(Tab_1, _2, ..., _k),
      lookup_{i,j+1}(Atom, Tab_1, Tval).
lookup_{i,j}(Atom, Table, Tval)  :-
      Atom = p_i(A_1, ..., A_j, ..., A_n),
      A_j = a_2,
      Table = table(_1, Tab_2, ..., _k),
      lookup_{i,j+1}(Atom, Tab_2, Tval).
      ...
lookup_{i,j}(Atom, Table, Tval)  :-
      Atom = p_i(A_1, ..., A_j, ..., A_n),
      A_j = a_k,
      Table = table(_1, _2, ..., Tab_k),
      lookup_{i,j+1}(Atom, Tab_k, Tval).
```

Finally, $\texttt{lookup}_{i,n+1}$ corresponds to the case where a path has been traced from the root of a symbol table all the way to a leaf. This predicate is therefore defined by the single clause

```
lookup_{i,n+1}(_, Tval, Tval).
```

We are now in a position to describe the translation of a Schönfinkel-Bernays formula $\varphi$ to a Prolog program $P_\varphi$. Let $\varphi$ be the formula $\exists x_1 \cdots \exists x_k \forall x_{k+1} \cdots \forall x_m \psi$. With each predicate symbol $p_i$ in $\psi$, the program $P_\varphi$ associates a variable $\texttt{ST\_p}_i$ that corresponds to the symbol table for $p_i$. The program $P_\varphi$ is given by the following:

(1) $P_\varphi$ contains a predicate $\texttt{main}/0$ defined by the clause

```
main  :- choose(X_1), ..., choose(X_k), not(q(X_1,...,X_k,f)).
```

(2) The predicate $\texttt{q}/k+1$ is defined by the clause

$\texttt{q(X}_1,\ldots,\texttt{X}_k\texttt{,Y)} \ \ \texttt{:-} \ \ \texttt{choose(X}_{k+1}\texttt{)}, \ldots, \texttt{choose(X}_m\texttt{)}, G, \mathit{SymTabLookups}$

where: $\mathcal{S}(\psi) = \langle G, \mathtt{Y} \rangle$; and *SymTabLookups* is a set of literals for symbol table lookups. For each literal $A \equiv \mathtt{p}_i(X_1, \ldots, X_n, \mathtt{T})$ in $G$, corresponding to an atom with an $n$-ary predicate symbol $p_i$ in $\varphi$ with associated logic variable $\mathtt{T}$, *SymTabLookups* contains a literal lookup_$p_{i,1}$($A$, ST_$p_i$, T)

where ST_$p_i$ is the variable corresponding to the symbol table for the predicate symbol $p_i$. These symbol table lookups force the execution of the Prolog program $P_\varphi$ to assign consistent truth values to atomic subformulae of the original formula $\varphi$.

(3) $P_\varphi$ contains a predicate choose/1 defined by $k$ clauses

choose($a_1$).
...
choose($a_k$).

where $a_1, \ldots, a_k$ are constants not appearing elsewhere in $P_\varphi$.

(4) Corresponding to each $n$-ary predicate symbol $p$ in $\varphi$, $P_\varphi$ contains an $n+1$-ary predicate p defined by the clauses

p($X_1, \ldots, X_n$, t).
p($X_1, \ldots, X_n$, f).

Intuitively, these predicates are used to select truth values for the ground atoms when "guessing" a model for $\varphi$.

(5) $P_\varphi$ contains definitions of the predicates and/3, or/3, and neg/2:

```
or(t, t, t).     and(t, t, t).     neg(t, f).
or(t, f, t).     and(t, f, f).     neg(f, t).
or(f, t, t).     and(f, t, f).
or(f, f, f).     and(f, f, f).
```

(6) The only exported predicate is main/0.

*Example* 3.2.2. Let $\varphi = \exists x_1 \exists x_2 \forall x_3 \forall x_4 \forall x_5[(r(x_1, x_2, x_3) \wedge s(x_2, x_4, x_5)) \vee \neg r(x_1, x_2, x_3)]$. Then the program $P_\varphi$ contains, apart from the definitions of symbol table management routines and of the predicates and/3, or/3, neg/2, the following clauses:

```
main :- choose(X1), choose(X2), not( q(X1, X2, f) ).

q(X1, X2, Y) :-
     choose(X3), choose(X4), choose(X5),
     r(X1, X2, X3, A1), s(X2, X4, X5, A2),
     and(A1, A2, T1),
     r(X1, X2, X3, A1),
     neg(A1, T2),
     or(T1, T2, Y),
     lookup_r1(r(X1, X2, X3), ST_r, A1),
     lookup_s1(s(X2, X4, X5), ST_s, A2),
     lookup_r1(r(X1, X2, X3), ST_r, A1).

choose(a1).
choose(a2).

r(_, _, _, t).
r(_, _, _, f).
```

```
s(_, _, _, t).
s(_, _, _, f).
```

The following result is not difficult to see:

LEMMA 3.2.3. *Given any Schönfinkel-Bernays formula $\varphi$, the program $P_\varphi$ can be generated in time polynomial in $|\varphi|$.*

LEMMA 3.2.4. *For any Schönfinkel-Bernays formula $\varphi$, the program $P_\varphi$ constructed as described above is alias free.*

PROOF (SKETCH). To see that the execution of $P_\varphi$ cannot cause two distinct variables to become aliased together at any point, it suffices to verify that none of the predicates in the program causes any two variables to become aliased in any execution starting from any exported predicate (there is only one, **main**/0). This is obvious for **choose**/1, **and**/3, **or**/3, and **neg**/2. For each $n$-ary predicate symbol $p$ in $\varphi$, the fact that the corresponding $n+1$-ary predicate **p** in $P_\varphi$ does not alias any variables together follows from the fact that such a predicate **p** is necessarily called with its first $n$ arguments bound to constants (defined by **choose**/1), and all that **p** does is bind its last argument to a constant.

It remains only to show that symbol table lookups do not give rise to any aliasing. For each $n$-ary predicate $p$, consider the lookup routines $\mathtt{lookup\_p}_{i,j}$, $0 \leq j \leq n$. The first argument of each of these lookup routines is ground by the time it is called, since all of the arguments to this term have been bound to constants by **choose**/1. Similarly, the third argument to each lookup routine is bound to one of the constants $\{\mathtt{t}, \mathtt{f}\}$ by the time it is called. It is then a straightforward induction on $n-j$ to show that the symbol tables constructed for any predicate do not contain repeated occurrences of any variable, and that because of this $\mathtt{lookup\_p}_{i,j}$ does not cause any aliasing to occur. □

THEOREM 3.2.5. *Exact analysis of recursion-free, alias-free logic programs containing negation is NEXPTIME-hard.*

PROOF. By reduction from the problem of deciding the satisfiability of Schönfinkel-Bernays formulae, which, from Theorem 2.5.8, is NEXPTIME-complete. From Lemma 3.2.2, given any Schönfinkel-Bernays formula $\varphi \equiv \exists x_1 \cdots \exists x_k \forall x_{k+1} \cdots \forall x_m \psi$, we can, in polynomial time, construct the program $P_\varphi$ as described above. Let the $k$ constants that are arguments of the predicate **choose**/1 in $P_\varphi$ be $\mathbf{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$. Further, for any $\mathbf{X} \subseteq \mathit{vars}(\psi)$, given any substitution $\theta : \mathbf{X} \longrightarrow \mathbf{A}$, let $\widehat{\theta}$ denote the "corresponding" substitution over the logic variables in $P_\varphi$, i.e., $\widehat{\mathbf{X}} = \{\mathtt{X}_i \mid x_i \in \mathbf{X}\}$, where $\mathtt{X}_i$ is the logic variable in $P_\varphi$ associated with the variable $x_i$ in $\varphi$, then $\widehat{\theta} : \widehat{\mathbf{X}} \longrightarrow \mathbf{A}$ is the substitution $\widehat{\theta}(\mathtt{X}_i) = \theta(x_i)$ for each $\mathtt{X}_i$ in $\widehat{\mathbf{X}}$. The proof proceeds as follows:

Let $\sigma : \{x_1, \ldots, x_k\} \longrightarrow \mathbf{A}$ be any substitution that grounds the variables $\{x_1, \ldots, x_k\}$. Since $\psi$ contains the variables $\{x_1, \ldots, x_k, x_{k+1}, \ldots, x_m\}$, a ground instance of the formula $\sigma(\psi)$ is false if and only if there is a substitution $\zeta : \{x_{k+1}, \ldots, x_m\} \longrightarrow \mathbf{A}$ such that $(\zeta \circ \sigma)(\psi)$ is false. It is a simple structural induction on $\psi$ to show that this can happen if and only if there is a substitution $\widehat{\zeta}$ such that execution of the the goal $(\widehat{\zeta} \circ \widehat{\sigma})(G)$, where $\mathcal{S}(\psi) = \langle G, X \rangle$, binds $X$

to `f`. This, in turn, can happen if and only if there are bindings for the variables $\mathtt{X}_{k+1}, \ldots, \mathtt{X}_m$ such that execution of the the goal

$$\mathtt{choose}(\mathtt{X}_{k+1}), \ldots, \mathtt{choose}(\mathtt{X}_m),\ G,\ \mathit{SymTabLookups},$$

where $\mathcal{S}(\psi) = \langle G, X \rangle$ and *SymTabLookups* is as defined earlier, binds $X$ to `f`. It follows, from the definition of the predicate `q` in $P_\varphi$, that a ground instance of $\sigma(\psi)$ is false if and only if there is a successful derivation for the goal $\widehat{\sigma}(\mathtt{q}(\mathtt{X}_1, \ldots, \mathtt{X}_k, \mathtt{f}))$. Since $P_\varphi$ does not contain any recursion, every derivation for queries involving only predicates defined in $P_\varphi$ must be finite. It follows that every ground instance of $\sigma(\psi)$ is true if and only if there is no successful derivation for the goal $\widehat{\sigma}(\mathtt{q}(\mathtt{X}_1, \ldots, \mathtt{X}_k, \mathtt{f}))$, i.e., that every derivation of this goal is finitely failed. But this can happen if and only if there are bindings for the variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$ such that the goal

$$\mathtt{choose}(\mathtt{X}_1), \ldots, \mathtt{choose}(\mathtt{X}_k), \mathtt{not}(\mathtt{q}(\mathtt{X}_1, \ldots, \mathtt{X}_k, \mathtt{f})) \tag{1}$$

has a successful SLDNF-derivation.

It is known that a Schönfinkel-Bernays formula $\exists x_1 \cdots \exists x_k \forall x_{k+1}, \ldots, x_m \psi$ has a model, i.e., is satisfiable, if and only if it has a model of size at most $k$ (see Lewis [1980, Section 8]). It follows, therefore, that this formula is satisfiable if and only if the goal (1) has a successful SLDNF-derivation. From the definition of the predicate `main`/0 in $P_\varphi$, it then follows that the predicate `main`/0 in the program $P_\varphi$ has a successful derivation—or, equivalently, has nonempty type—if and only if the formula $\varphi$ is satisfiable. The program $P_\varphi$ is recursion free by construction, and from Lemma 3.2.4 is alias free. It follows that precise analysis of recursion-free, alias-free logic programs containing negation is NEXPTIME-hard. □

We do not know, at this time, whether precise analysis of recursion-free logic programs containing negation is in NEXPTIME. Intuitively, since there is no recursion, one expects such programs to be decidable. However, because there may be function symbols with nonzero arity, a question about dataflow analysis information will generally correspond to a (possibly infinite) set of possible "concrete" queries, and it is not clear whether such questions can be answered in nondeterministic exponential time.

It is possible to strengthen Theorem 3.2.5 slightly. Some authors, for example, Mannila and Ukkonen [1987], have argued that a flow analysis algorithm whose complexity is exponential in the number of literals in a clause body may be tolerable in practice because in most commonly encountered programs the number of literals in clause bodies is small, and may be assumed to be $O(1)$. Now given any program $P$, we can derive from it an "equivalent" program $P'$ where no clause has more than two literals in its body. We compute a series of programs $P_0$, $P_1$, $\ldots$, $P_i$, $\ldots$, as follows: $P_0 = P$, and $P_{i+1}$ is obtained from $P_i$ as follows: let $C \equiv p(\bar{t})\ :-\ q_1(\bar{t}_1), q_2(\bar{t}_2), \ldots, q_n(\bar{t}_n)$ be any clause in $P_i$ whose body contains more than two literals. Let $U$ be the set of variables

$$(\mathit{vars}(\bar{t}) \cup \mathit{vars}(\bar{t}_1)) \cap (\mathit{vars}(\bar{t}_2) \cup \cdots \cup \mathit{vars}(\bar{t}_n)),$$

and let $\bar{u}$ be some enumeration of $U$. Let $p'$ be a new predicate symbol, with arity $|U|$, not appearing elsewhere in $P_i$. Define the clauses $C_1$ and $C_2$ as follows: $C_1 \equiv p(\bar{t})\ :-\ q_1(\bar{t}_1), p'(\bar{u})$; and $C_2 \equiv p'(\bar{u})\ :-\ q_2(\bar{t}_2), \ldots, q_n(\bar{t}_n)$. Then, $P_{i+1} =$

$(P_i \setminus \{C\}) \cup \{C_1, C_2\}$. The program $P'$ is the limit of the sequence of programs $P_0$, $P_1$, $\ldots$, obtained in this way. Since each application of this transformation replaces a clause in $P'$ with two clauses each containing fewer body literals, $P'$ can be computed in a finite number of steps, and does not contain any clause with more than two body literals. It is not difficult to see that the program $P'$ is equivalent to $P$ in the sense that for any predicate $p$ defined in $P$, a goal $p(\bar{t})$ succeeds in $P$ with answer substitution $\theta$ if and only if it succeeds in $P'$ with answer substitution $\theta$. Further, each clause $C$ in the original program $P$ with $n > 2$ body literals is replaced by $n - 2$ clauses in the transformed program $P'$, each having two body literals, each of these literals being no larger than the original clause $C$: this means that the transformed program is no more than quadratically larger than the original program, i.e., there is at most a polynomial growth in program size as a result of the transformation. We can therefore use the programs resulting from this transformation in the proof of Theorem 3.2.5 to obtain the following result:

THEOREM 3.2.6. *Exact analysis of recursion-free, alias-free logic programs containing negation is NEXPTIME-hard. It remains NEXPTIME-hard even if no clause in the program contains more than two body literals.*

This shows that, depending on how precise an analysis strives to be, it may have a bad worst-case complexity even if we assume that the number of literals per clause is $O(1)$.

### 3.3 Recursion-Free Programs without Negation

Since most dataflow analysis algorithms reported in the logic programming literature do not attempt to deal with negation very precisely, we next consider the analysis of programs that do not contain negation, i.e., Horn programs. We first consider analyses, such as the type analysis of Van Hentenryck et al. [1994], that sacrifice precision only when recursion is involved. We show that for such analyses, precise dataflow analysis is PSPACE-hard. To this end, we first describe a procedure that, given a quantified Boolean formula $\varphi$, generates a logic program $P_\varphi$ that can be used to establish the truth of $\varphi$. Let $\varphi$ be the formula $Q_1 x_1 \cdots Q_n x_n \psi$, where each of the $Q_i$ is a quantifier $\forall$ or $\exists$, and let $s_i$ denote the formula $Q_i x_i Q_{i+1} x_{i+1} \cdots Q_n x_n \psi$, $1 \leq i \leq n$. We assume that each variable $x_i$ in $\varphi$ is associated with a logic variable $\mathtt{X}_i$. The program $P_\varphi$ is given by the following:

(1) $P_\varphi$ contains a "root predicate" `main/0` that evaluates whether or not the formula $\varphi$ is true. This predicate is defined by the following clause:

```
main :- p1([], t).
```

This predicate is the only exported predicate in $P_\varphi$.

(2) For each $s_i, 1 \leq i \leq n$, $P_\varphi$ contains a predicate $\mathtt{p}_i(\mathtt{V}_i, \mathit{Tval})$. Here, $\mathtt{V}_i$ is a variable that, at runtime, gets bound to a list of truth values $[\mathtt{X}_{i-1}, \ldots, \mathtt{X}_1]$ corresponding to assignments to the variables $x_{i-1}, \ldots, x_1$ that are bound by the quantifiers "before," i.e., to the left of, $Q_i$ in $\varphi$; and $\mathit{Tval}$ is either `t` or `f` depending on whether or not the corresponding instance of $s_i$ is true. These predicates are defined as follows:

(a) If $Q_i = \forall$, then $\mathtt{p}_i$ is defined as follows:

```
pᵢ(Vᵢ, t)  :- pᵢ₊₁([t | Vᵢ], t), pᵢ₊₁([f | Vᵢ], t).
pᵢ(Vᵢ, f)  :- pᵢ₊₁([t | Vᵢ], f).
pᵢ(Vᵢ, f)  :- pᵢ₊₁([f | Vᵢ], f).
```

(b) If $Q_i = \exists$, then $\mathtt{p}_i$ is defined as follows:

```
pᵢ(Vᵢ, t)  :- pᵢ₊₁([t | Vᵢ], t).
pᵢ(Vᵢ, t)  :- pᵢ₊₁([f | Vᵢ], t).
pᵢ(Vᵢ, f)  :- pᵢ₊₁([t | Vᵢ], f), pᵢ₊₁([f | Vᵢ], f).
```

(3) The predicate $\mathtt{p}_{n+1}$ is defined by the clause

$\mathtt{p}_{n+1}\mathtt{([X}_n,\ldots,\mathtt{X}_1\mathtt{],\ Tval)\ \ :-\ eval(}Fmla\mathtt{,\ Tval).}$

where $Fmla \equiv \mathcal{T}(\psi)$ is a Prolog term representing the formula $\psi$. The mapping $\mathcal{T}$ is defined as follows:

— $\mathcal{T}(x_i) = \mathtt{X}_i$, where $\mathtt{X}_i$ is the logic variable associated with $x_i$ in $P_\varphi$;
— $\mathcal{T}(\mathbf{true}) = \mathtt{t}$, and $\mathcal{T}(\mathbf{false}) = \mathtt{f}$;
— $\mathcal{T}(\alpha \wedge \beta) = \mathtt{and}(A, B)$, where $A = \mathcal{T}(\alpha)$ and $B = \mathcal{T}(\beta)$;
— $\mathcal{T}(\alpha \vee \beta) = \mathtt{or}(A, B)$, where $A = \mathcal{T}(\alpha)$ and $B = \mathcal{T}(\beta)$;
— $\mathcal{T}(\neg\alpha) = \mathtt{neg}(A)$, where $A = \mathcal{T}(\alpha)$.

(4) The program $P_\varphi$ contains the definition for the predicate **eval**/2. The idea here is that, given a ground term $T$ representing a propositional formula, the execution of $\mathtt{eval}(T, \mathtt{t})$ succeeds if and only if $\varphi$ is true, while that of $\mathtt{eval}(T, \mathtt{f})$ succeeds if and only if $\varphi$ is false. The obvious definition of such a predicate would contain clauses such as

```
eval(and(F1, F2), t)  :- eval(F1, t), eval(F2, t).
eval(and(F1, _), f)  :- eval(F1, f).
eval(and(_, F2), f)  :- eval(F2, f).
```

and so on. The problem with such a definition is that it is recursive, while we are considering only recursion-free programs. One could imagine partial evaluation of the **eval** predicate above with respect to the input formula in order to remove the recursion, but a straightforward approach to this would result in a clause with size proportional to the input formula. With a little more effort, it is possible to take (a representation of) any particular propositional formula and produce a recursion-free program, where each clause has bounded size, to determine its truth value: if $\varphi \equiv Q_1 x_1 \cdots Q_n x_n \psi$, where $\psi$ is quantifier free, and the number of logical connectives in $\psi$ is $k$, then the truth value of any ground instance of $\psi$ can be evaluated using a recursion-free program containing $k+1$ predicates $\mathtt{eval}_0/2, \ldots, \mathtt{eval}_k/2$. Intuitively, for each $i$, $0 \le i \le k$, the predicate $\mathtt{eval}_i$ is capable of evaluating the truth of propositions containing at most $i$ logical connectives. The predicates $\mathtt{eval}_i, 1 \le i \le k$, are defined as follows:

```
evalᵢ(and(F1, F2), t)  :- evalᵢ₋₁(F1, t), evalᵢ₋₁(F2, t).
evalᵢ(and(F1, _), f)  :- evalᵢ₋₁(F1, f).
evalᵢ(and(_, F2), f)  :- evalᵢ₋₁(F2, f).

evalᵢ(or(F1, _), t)  :- evalᵢ₋₁(F1, t).
evalᵢ(or(_, F2), t)  :- evalᵢ₋₁(F2, t).
evalᵢ(or(F1, F2), f)  :- evalᵢ₋₁(F1, f), evalᵢ₋₁(F2, f).
```

```
evalᵢ(neg(F), t)  :- evalᵢ₋₁(F, f).
evalᵢ(neg(F), f)  :- evalᵢ₋₁(F, t).

evalᵢ(t, t).
evalᵢ(f, f).
```

The predicate $\mathtt{eval}_0/2$ is defined by the clauses

```
eval₀(t, t).
eval₀(f, f).
```

Finally, the predicate `eval/2` is defined as

```
eval(X, Y)  :- evalₖ(X, Y).
```

*Example* 3.3.1. Let $\varphi = \forall x \exists y \forall z[x \wedge \neg(y \vee z)]$. Then, apart from the definitions of the various $\mathtt{eval}_i/2$ predicates, the program $P_\varphi$ contains the following clauses:

```
main :- p1([], t).

p1(V1, t) :- p2([t|V1], t), p2([f|V1], t).   /* universal */
p1(V1, f) :- p2([t|V1], f).
p1(V1, f) :- p2([f|V1], f).

p2(V2, t) :- p3([t|V2], t).                  /* existential */
p2(V2, t) :- p3([f|V2], t).
p2(V2, f) :- p3([t|V2], f), p3([f|V2], f).

p3(V3, t) :- p4([t|V3], t), p4([f|V3], t).   /* universal */
p3(V3, f) :- p4([t|V3], f).
p3(V3, f) :- p4([f|V3], f).

p4([Z, Y, X], Tval) :- eval(and(X, neg(or(Y, Z))), Tval).
```

Each clause in the program $P_\varphi$ has a fixed structure and can be generated in $O(1)$ time. The total number of predicates in $P_\varphi$ is $O(|\varphi|)$, so the reduction requires $O(\log |\varphi|)$ space to keep track of them. The following result is then straightforward:

Lemma 3.3.2. *For any quantified Boolean formula $\varphi$, the program $P_\varphi$ can be constructed in polynomial time using logarithmic space.*

We observe that given any quantified Boolean formula $\varphi$, the program $P_\varphi$ constructed as above is recursion free and has bounded arity (the arity of every predicate and function symbol is bounded by 2), and each clause contains at most two body literals. Further, it is easy to verify, by inspection, that the program is alias free. This leads immediately to the following result:

Theorem 3.3.3. *Exact analysis of recursion-free, alias-free Horn programs is PSPACE-hard. The problem remains PSPACE-hard even if the arity of each function and predicate symbol is restricted to at most 2, and there are at most 2 literals in the body of any clause.*

Proof (Sketch). By reduction from QBF, which, from Theorem 2.5.2, is PSPACE-complete. From Lemma 3.3.2, given any quantified Boolean formula $\varphi$, we can, in logarithmic space, construct the program $P_\varphi$ as described above. The proof proceeds in two parts. First we show, by induction on $m$, that for any propositional formula $\psi$ containing at most $m$ logical connectives the goal $\texttt{eval}_m(T, \texttt{Tval})$, where $T = \mathcal{T}(\psi)$, evaluates correctly the truth value Tval of $\psi$ for any given truth assignment for its variables. The base case, with $m = 0$, involves only truth values (since, by assumption, all propositional variables in $\psi$ have truth assignments already), and is trivial from the definition of $\texttt{eval}_0/2$. The inductive case then uses a straightforward structural induction on the formula $\psi$. After this, given a quantified Boolean formula $\varphi$ with $n$ quantifiers, it is a simple induction on $n$ to show that $\varphi$ is true if and only if the execution of p succeeds, i.e., if and only if main/0 has nonempty type. □

Note that here, as in the case of Theorem 3.2.6, bounding the number of body literals per clause does not improve the worst-case complexity of precise dataflow analysis.

## 4. ANALYSIS OF DATALOG PROGRAMS

The construction used in Theorem 3.3.3 relies on the existence of a binary function symbol, which is used to represent truth assignments as lists of truth values. In this section we examine the extent to which disallowing function symbols with nonzero arity simplifies dataflow analysis. As discussed in Section 2.4, these results are applicable to analyses that use depth abstraction, such as Codish et al. [1994], King [1994], Marriott and Søndergaard [1988], Santos Costa et al. [1991], Sato and Tamaki [1984], and Taylor [1989].

### 4.1 Recursive Datalog Programs

Consider the translation of a recursive monotone Boolean function (RMBF) to a Horn program. Let $\varphi = \langle \mathsf{eq}, \mathsf{a} \rangle$ be an instance of RMBF, where $\mathsf{eq} \equiv f(x_1, \ldots, x_n) = E$ is a recursive equation and $\mathsf{a} \equiv \langle a_1, \ldots, a_n \rangle$ is a tuple of $n$ arguments. We can construct a program $P_\varphi$ to evaluate $\varphi$, as follows:

(1) Let $\mathcal{S}(E) = \langle G, \texttt{Tval} \rangle$, where the mapping $\mathcal{S}$ is defined above, then $P_\varphi$ contains a clause $\texttt{f}(\texttt{X}_1, \ldots, \texttt{X}_n, \texttt{Tval})\ \texttt{:-}\ G$.

(2) Let $\mathcal{S}(f(a_1, \ldots, a_n)) = \langle G, \texttt{X} \rangle$, then $P_\varphi$ contains a "root predicate" main/0 defined by the clause $\texttt{main :-}\ G,\ \texttt{X} = \texttt{t}$. This is the only exported predicate in $P_\varphi$.

(3) $P_\varphi$ contains definitions of the predicates and/3, or/3, and neg/2:

```
or(t, t, t).     and(t, t, t).     neg(t, f).
or(t, f, t).     and(t, f, f).     neg(f, t).
or(f, t, t).     and(f, t, f).
or(f, f, f).     and(f, f, f).
```

*Example* 4.1.1. Let $\varphi = \langle \mathsf{eq}, \mathsf{a} \rangle$ be an instance of RMBF, where eq is given by

$$g(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge g(g(x_3 \vee x_2, x_3, x_1), x_3, x_1 \wedge x_2))$$

and $\mathsf{a} = \langle \mathbf{true}, \mathbf{false}, \mathbf{false} \rangle$. Then, $P_\varphi$ contains, apart from the definitions of the predicates and/3, or/3, and neg/2, the following clauses:

```
g(X1, X2, X3, Tval) :-
     or(X3, X2, U1),
     g(U1, X3, X1, U2), and(X1, X2, U3),
     g(U2, X3, U3, U4),
     and(X2, U4, U5), or(X1, U5, Tval).

main :- X1 = t, X2 = f, X3 = f, g(X1, X2, X3, X), X = t.
```

LEMMA 4.1.2. *Let $\varphi = \langle \mathsf{eq}, \mathsf{a} \rangle$ be an instance of RMBF. Then, the program $P_\varphi$ can be constructed in time polynomial in $\mid \varphi \mid$.*

THEOREM 4.1.3. *Exact analysis of alias-free Datalog programs containing at least two distinct constants is EXPTIME-complete.*

PROOF (SKETCH). The proof follows by reduction from RMBF, which, from Theorem 2.5.6, is EXPTIME-complete. From Lemma 4.1.2, given an instance $\varphi$ of RMBF, the program $P_\varphi$ can be generated in time polynomial in the length of $\varphi$. It is then a straightforward fixpoint induction to show that the root predicate **main**/0 in $P_\varphi$ has nonempty type if and only if $\varphi$ is true. This shows that the problem is EXPTIME-hard.

To see that the problem is in EXPTIME, consider the execution of a program $P$ containing $p$ different predicates and $c$ distinct constants. Without loss of generality, assume that each predicate has arity $a$ (we can always add dummy arguments to predicates that have too few arguments in order to comply with this assumption). The execution of $P$ can be simulated via a program $P'$ that has one predicate of arity $a+1$ and $c+p$ constants: the idea is to add one argument to each literal in $P'$ to indicate which predicate in $P$ is being referred to, e.g., a literal $q(t_1, \ldots, t_n)$ in $P$ corresponds to a literal $interp(q, t_1, \ldots, t_n)$, where *interp* is the single predicate appearing in $P'$. Thus, the complexity of evaluating $P$ is no larger than that of $P'$.

Now consider the complexity of evaluating $P'$: to simplify notation, let $b = a+1$ and $n = c+p$, i.e., $P'$ has $n$ constants and a single predicate of arity $b$. Then, the total number of distinct atoms possible for *interp* is $n^b$, and the computational cost for exact analysis of $P'$ can be no greater than that of evaluating all of $P'$ explicitly. First, consider the cost of evaluating $lfp(T_{P'}) = \cup_{i \geq 0} T^i_P(\emptyset)$ by iteratively computing the limit of the sequence $T_P(\emptyset)$, $T^2_P(\emptyset)$, ..., $T^i_P(\emptyset)$, ...: this can take no more than $n^b$ iterations, with each iteration examining no more than $O(n^b)$ atoms, and hence involves $O(n^{2b})$ work. Apt and van Emden [1982] show that the set $SS(P')$ of ground atoms of $P'$ that have successful derivations coincides with $lfp(T_{P'})$, so the cost of evaluating $SS(P')$ is also $O(n^{2b})$. Since $n^{2b}$ is $O(2^{n^b})$, it follows that the evaluation of $P'$—and, therefore, that of the original program $P$—is in EXPTIME. □

It is straightforward to use the transformation discussed at the end of Section 3.2 to extend this result to programs where no clause has more than two body literals.

### 4.2 Recursion-Free Datalog Programs

As Theorem 4.1.3 shows, precise analysis of programs can be expensive even if the analysis algorithm does not try to deal with compound terms and aliasing. Now it may not be unreasonable for an analysis algorithm to try to be precise for nonrecursive programs, but to surrender some precision when dealing with recursive calls. To examine the complexity of such analyses, we next consider the case where the exactness sets of the analyses do not contain recursive programs. For this, we first describe a procedure that takes a quantified Boolean formula $\varphi$ and generates a program $P_\varphi$ that can be used to determine whether $\varphi$ is true. The basic idea behind this construction is very similar to that used in the proof of Theorem 3.3.3. Let $\varphi$ be the formula $Q_1 x_1 \cdots Q_n x_n \psi$, where each of the $Q_i$ is a quantifier $\forall$ or $\exists$, and let $s_i$ denote the formula $Q_i x_i \cdots Q_n x_n \psi$, $1 \leq i \leq n$. The program $P_\varphi$ is given by the following:

(1) Corresponding to the variable $x_i$ in $\varphi$ we associate a variable $\mathtt{X}_i$ in $P_\varphi$.

(2) $P_\varphi$ contains a "root predicate" `main/0` that evaluates whether or not the formula $\varphi$ is true. This predicate is defined by the following clause:

```
main  :- p1(t).
```

The only exported predicate in $P_\varphi$ is `main/0`.

(3) For each $s_i, 1 \leq i \leq n$, $P_\varphi$ contains a predicate $\mathtt{p}_i(\mathtt{V}_1, \ldots, \mathtt{V}_{i-1}, \mathit{Tval})$. Here, $\mathtt{V}_1, \ldots, \mathtt{V}_{i-1}$ are variables that, at runtime, get bound to truth values that correspond to truth assignments to the variables $x_1, \ldots, x_{i-1}$ bound by the quantifiers "before," i.e., to the left of, $Q_i$ in $\varphi$; and *Tval* is either `t` or `f` depending on whether or not the corresponding instance of $s_i$ is true. These predicates are defined as follows:

(a) If $Q_i = \forall$, then $\mathtt{p}_i$ is defined as follows:

```
pi(V1,...,Vi-1, t)  :-
    pi+1(V1,...,Vi-1, t, t), pi+1(V1,...,Vi-1, f, t).
pi(V1,...,Vi-1, f)  :- pi+1(V1,...,Vi-1, t, f).
pi(V1,...,Vi-1, f)  :- pi+1(V1,...,Vi-1, f, f).
```

(b) If $Q_i = \exists$, then $\mathtt{p}_i$ is defined as follows:

```
pi(V1,...,Vi-1, t)  :- pi+1(V1,...,Vi-1, t, t).
pi(V1,...,Vi-1, t)  :- pi+1(V1,...,Vi-1, f, t).
pi(V1,...,Vi-1, f)  :-
    pi+1(V1,...,Vi-1, t, f), pi+1(V1,...,Vi-1, f, f).
```

(4) The predicate $\mathtt{p}_{n+1}$ is defined by the clause $\mathtt{p}_{n+1}(\mathtt{X}_1, \ldots, \mathtt{X}_n,\ \mathtt{A})\ \ \text{:-}\ G$ where $\mathcal{S}(\psi) = \langle G, \mathtt{A} \rangle$, $\mathcal{S}$ being the mapping defined at the beginning of Section 3.2, and $\mathtt{X}_i \neq \mathtt{A}, 1 \leq i \leq n$.

(5) $P_\varphi$ contains definitions for the predicates `and/3`, `or/3`, and `neg/2`:

```
or(t, t, t).        and(t, t, t).       neg(t, f).
or(t, f, t).        and(t, f, f).       neg(f, t).
or(f, t, t).        and(f, t, f).
or(f, f, f).        and(f, f, f).
```

*Example* 4.2.1. Let $\varphi = \forall x \exists y \forall z[x \wedge \neg(y \vee z)]$. Then the program $P_\varphi$ contains, apart from the definitions for `and/3`, `or/3`, and `neg/2`, the following clauses:

```
main :- p1(t).

p1(t) :- p2(t, t), p2(f, t).
```

```
p1(f) :- p2(t, f).
p1(f) :- p2(f, f).

p2(V1, t) :- p3(V1, t, t).
p2(V1, t) :- p3(V1, f, t).
p2(V1, f) :- p3(V1, t, f), p3(V1, f, f).

p3(V1, V2, t) :- p4(V1, V2, t, t), p4(V1, V2, f, t).
p3(V1, V2, f) :- p4(V1, V2, t, f).
p3(V1, V2, f) :- p4(V1, V2, f, f).

p4(X, Y, Z, A) :- or(Y, Z, U1), neg(U1, U2), and(X, U2, A).
```

THEOREM 4.2.2. *Exact analysis of recursion-free, alias-free Datalog programs containing at least two distinct constants is PSPACE-hard.*

PROOF. Similar to that of Theorem 3.2.7. □

Again, we can use the transformation discussed at the end of Section 3.2 to extend this result to programs where no clause has more than two body literals.

### 4.3 Recursion-Free Bounded-Arity Datalog Programs

In the construction used in the proof of Theorem 4.2.2, the maximum arity of predicates increases as the number of quantifiers increases. In the literature on dataflow analysis of logic programs, some researchers have assumed that the maximum arity of predicates in any program is bounded, i.e., $O(1)$. We next consider the effect of imposing this additional restriction on the complexity of analyses. For this, we describe a procedure that, given a Boolean formula $\varphi$, generates a logic program $P_\varphi$ that can be used to test whether or not $\varphi$ is satisfiable.

(1) $P_\varphi$ contains a predicate `choose_tval`/2, defined by the clauses

```
choose_tval(t).
choose_tval(f).
```

Let the variables occurring in $\varphi$ be $x_1, \ldots, x_n$, and let them be associated with $n$ logic variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$. $P_\varphi$ contains a clause for the root predicate `main`/0, defined by the clause

$$\mathtt{main}\ \ :-\ \mathtt{choose\_tval(X_1)},\ \ldots,\ \mathtt{choose\_tval(X_n)},\ G,\ \mathtt{U = t.}$$

where $\mathcal{S}(\varphi) = \langle G, \mathtt{U}\rangle$, with the mapping $\mathcal{S}$ as defined at the beginning of Section 3.2.

(2) $P_\varphi$ contains definitions for the predicates `and`/3, `or`/3, and `neg`/2, as shown earlier.

(3) The only exported predicate in $P_\varphi$ is `main`/0.

*Example* 4.2.3. Let $\varphi = (x \wedge \neg y) \vee (\neg(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z))$. Let the variables $x, y$, and $z$ in $\varphi$ be associated with the logic variables `X`, `Y`, and `Z`, respectively, in $P_\varphi$. Then, $P_\varphi$ contains, apart from the definitions of the predicates `choose_tval`/1, `and`/3, `or`/3, and `neg`/2, the following clause:

```
main :-
   choose_tval(X), choose_tval(Y), choose_tval(Z),
   neg(Y,T1), and(X,T1,T2),
   neg(Y,U1), or(X,U1,U2), neg(Z,U3), or(U2,U3,U4), neg(U4,U5),
   neg(X,V1), or(V1,Y,V2), or(V2,Z,V3),
   and(U5,V3,W1),
   or(T2,W1,W2),
   W2 = t.
```

LEMMA 4.2.4. *Given any Boolean formula $\varphi$, the program $P_\varphi$ can be constructed in time polynomial in the size of $\varphi$.*

THEOREM 4.2.5. *Precise analysis of recursion-free, alias-free, bounded-arity Datalog programs containing at least two distinct constants is NP-complete. It remains NP-complete even if no predicate has arity exceeding 3.*

PROOF (SKETCH). The proof is by reduction from the satisfiability problem for propositional clauses. For any propositional clause $\varphi$, we show, by structural induction on $\varphi$, that $\varphi$ is satisfiable if and only if the type for the predicate **main**/0 in the program $P_\varphi$ is nonempty. This shows that the problem is NP-hard.

To show that the problem is in NP, consider any recursion-free, alias-free, bounded-arity Datalog program $P$ that contains $c \geq 2$ distinct constants. Let $a$ be the maximum arity of any predicate in $P$. For any predicate $p$, to determine whether a particular tuple of constants is in the relation of $p$, it suffices to "guess" an execution path in the program, then verify that this path does generate that tuple. Since there is no recursion, such a path can be given by simply selecting, for each literal $q(\ldots)$ in the body of a clause, which clause for $q$ to use during resolution. The total length of such an execution path is no greater than the size of the program. Since the arity of each predicate is bounded, the total number of variables involved in the execution is also no larger than the size of the program. Thus, for each $n$-ary predicate and $n$-tuple of constants, verifying whether that tuple is in the relation of the predicate can be carried out in time polynomial in the size of $P$. Now the number of tuples for any predicate is $O(c^a)$, which is polynomial in the size of the program since $a$ is $O(1)$; and further, the number of predicates in $P$ is no greater than the size of the program. The theorem follows. □

## 5. ANALYSIS OF FUNCTION-FREE PROGRAMS

As seen from the discussion of the previous section, precise analysis of programs containing function symbols can be quite difficult. In this section, we consider programs that do not contain any function symbols and do not have any failed execution branches. This is motivated by a variety of groundness and aliasing analyses that have been proposed in the literature in recent years, primarily in the context of compiler parallelization and code optimization of logic programs [Chang et al. 1985; Debray 1989; Jacobs and Langen 1989; Marriott et al. 1994; Muthukumar and Hermenegildo 1989; 1991]. Such analyses typically do not keep track of the function symbols of the various terms that a variable might be bound to, and as a result may not be able to detect execution branches that fail at runtime. The precision of such algorithms can be examined by considering programs that do not contain any function symbols or constants, and also do not contain any failed execution branches (it may seem strange to consider the precision of groundness analyses by considering programs where all terms are variables: the idea is to see how well an analysis is able to propagate ground values that are given as arguments in a query in order to determine what other variables become ground at different program points).

For the complexity results of this section, we rely on aliasing to mimic the evaluation of monotone Boolean formulae. The essential idea here is to associate, with each proposition $\varphi$, two logic variables $X_{\varphi L}$ and $X_{\varphi R}$. A truth value of **true** is denoted by having $X_{\varphi L}$ and $X_{\varphi R}$ aliased together, while a truth value of **false** is denoted by having these variables independent, i.e., not aliases. If the variables associated with a formula $\varphi$ are $X_{\varphi L}$ and $X_{\varphi R}$, then we say that $\varphi$ is *evaluated into* the pair $(X_{\varphi L}, X_{\varphi R})$. As a special case, each variable $x$ in a proposition $\varphi$ is associated with two logic variables $\mathtt{X}_L$ and $\mathtt{X}_R$.

Conjunction is simulated by a predicate `and/6` that is defined as follows:

```
and(Xl, Xr, Yl, Yr, Xl, Yr) :- Xr = Yl.
```

The first two argument positions of `and/6` correspond to the truth value for one of the conjuncts; the third and fourth arguments correspond to the truth value for the other conjunct; and the last two arguments are the truth value of the entire conjunction. The intuition is that the last two argument positions are aliased together (denoting a truth value of **true**) if and only if the first two arguments are aliased (indicating that the first conjunct has truth value **true**) and that the third and fourth arguments are also aliased (indicating that the second conjunct also has a truth value **true**).

Disjunction is simulated by a predicate `or/6` that defined as follows:

```
or(Xl, Xr, _, _, Xl, Xr).
or(_, _, Yl, Yr, Yl, Yr).
```

The argument positions of `or/6` have the same significance as for `and/6`: the first two arguments correspond to one disjunct, the third and fourth arguments to the other disjunct, and the last two argument positions denote the truth value of the disjunction itself. It is not difficult to see that the last two argument positions of `or/6` can be aliased together (denoting a truth value of **true**) if and only if either the first and second arguments are aliased together, or the third and fourth arguments are aliased together, or both, i.e., if and only if at least one of the disjuncts has a truth value of **true**.

### 5.1 Function-Free, Failure-Free Programs Containing Recursion

We now describe a mapping $\mathcal{F}$ that, given any (recursive) monotone Boolean formula $\varphi$, yields a triple $\langle G, X, Y\rangle$. Here, $G$ is a Prolog goal, and $X, Y$ are variables in $G$, such that $\varphi$ is true if and only if $X$ and $Y$ become aliased together when $G$ is executed. The construction proceeds as follows:

(1) If $\varphi$ is a propositional variable $x$ associated with the logic variables $X_L$ and $X_R$, then $\mathcal{F}(\varphi) = \langle \varepsilon, X_L, X_R\rangle$, where $\varepsilon$ denotes the empty sequence of goals.

(2) If $\varphi$ is the constant **true**, then $\mathcal{F}(\varphi) = \langle X = Y, X, Y\rangle$; if $\varphi$ is the constant **false**, then $\mathcal{F}(\varphi) = \langle \varepsilon, X, Y\rangle$.

(3) If $\varphi$ is of the form $\phi \wedge \psi$, with $\mathcal{F}(\phi) = \langle G_\phi, X_{\phi L}, X_{\phi R}\rangle$ and $\mathcal{F}(\psi) = \langle G_\psi, X_{\psi L}, X_{\psi R}\rangle$, then $\mathcal{F}(\varphi) = \langle G, X_{\varphi L}, X_{\varphi R}\rangle$, where

$$G = `G_\phi,\ G_\psi,\ \mathtt{and}(X_{\phi L}, X_{\phi R}, X_{\psi L}, X_{\psi R}, X_{\varphi L}, X_{\varphi R})'$$

where $X_{\phi L}$ and $X_{\phi R}$ are not in $vars(G_\psi)$; $X_{\psi L}$ and $X_{\psi R}$ are not in $vars(G_\phi)$; and $X_{\varphi L}$ and $X_{\varphi R}$ are not in $vars(G_\phi) \cup vars(G_\psi)$.

(4) If $\varphi$ is of the form $\phi \vee \psi$, with $\mathcal{F}(\phi) = \langle G_\phi, X_{\phi L}, X_{\phi R}\rangle$ and $\mathcal{F}(\psi) = \langle G_\psi, X_{\psi L}, X_{\psi R}\rangle$, then $\mathcal{F}(\varphi) = \langle G, X_{\varphi L}, X_{\varphi R}\rangle$, where

$$G = `G_\phi,\ G_\psi,\ \mathtt{or}(X_{\phi L}, X_{\phi R}, X_{\psi L}, X_{\psi R}, X_{\varphi L}, X_{\varphi R})'$$

where $X_{\phi L}$ and $X_{\phi R}$ are not in $vars(G_\psi)$; $X_{\psi L}$ and $X_{\psi R}$ are not in $vars(G_\phi)$; and $X_{\varphi L}$ and $X_{\varphi R}$ are not in $vars(G_\phi) \cup vars(G_\psi)$.

(5) If $\varphi$ is of the form $f(\phi_1, \ldots, \phi_n)$, with $\mathcal{F}(\phi_i) = \langle G_i, X_{iL}, X_{iR}\rangle$, then $\mathcal{F}(\varphi) = \langle G, X_{\varphi L}, X_{\varphi R}\rangle$, where

$$G = `G_1, \ldots, G_n,\ \mathtt{f}(X_{1L}, X_{1R}, \ldots, X_{nL}, X_{nR}, X_{\varphi L}, X_{\varphi R})'$$

where $X_{\varphi L}$ and $X_{\varphi R}$ are not in $vars(G_i), 1 \le i \le n$; and $i \ne j$ implies $X_{iL}$, $X_{iR}$ are not in $vars(G_j)$, $1 \le i, j \le n$.

*Example* 5.1.1. Let $\varphi = (x \wedge y) \vee (y \wedge (x \vee z))$, then

$$\begin{aligned}\mathcal{F}(\varphi) = \ & \mathtt{and}(\mathtt{X}_L, \mathtt{X}_R, \mathtt{Y}_L, \mathtt{Y}_R, \mathtt{U}_L, \mathtt{U}_R),\\ & \mathtt{or}(\mathtt{X}_L, \mathtt{X}_R, \mathtt{Z}_L, \mathtt{Z}_R, \mathtt{V}_L, \mathtt{V}_R),\\ & \mathtt{and}(\mathtt{Y}_L, \mathtt{Y}_R, \mathtt{V}_L, \mathtt{V}_R, \mathtt{W}_L, \mathtt{W}_R),\\ & \mathtt{or}(\mathtt{U}_L, \mathtt{U}_R, \mathtt{W}_L, \mathtt{W}_R, A_L, A_R).\end{aligned}$$

It is not difficult to see that the following lemma is true:

LEMMA 5.1.2. *Given any (recursive) monotone Boolean formula $\varphi$, the sequence of atoms $\mathcal{F}(\varphi)$ can be constructed in time polynomial in the size of $\varphi$.*

We first consider the translation of a recursive monotone Boolean function (RMBF) to a function-free Horn program. Let $\varphi = \langle \mathsf{eq}, \mathsf{a}\rangle$ be an instance of RMBF, where $\mathsf{eq} \equiv f(x_1, \ldots, x_n) = E$ is a recursive equation, and $\mathsf{a} \equiv \langle a_1, \ldots, a_n\rangle$ is a tuple of $n$ arguments. We construct a program $P_\varphi$ to evaluate $\varphi$, as follows:

(1) Let $\mathcal{F}(E) = \langle G, \mathtt{U}_L, \mathtt{U}_R\rangle$; then $P_\varphi$ contains a clause

$$\mathtt{f}(\mathtt{X}_{1L}, \mathtt{X}_{1R}, \ldots, \mathtt{X}_{nL}, \mathtt{X}_{nR}, \mathtt{U}_L, \mathtt{U}_R)\ :-\ G.$$

(2) Let $\mathcal{F}(f(a_1, \ldots, a_n)) = \langle G, \mathtt{U}_L, \mathtt{U}_R\rangle$; then $P_\varphi$ contains a "root predicate" $\mathtt{p}/2$ defined by the clause $\mathtt{p}(\mathtt{U}_L, \mathtt{U}_R)\ :-\ G$.

(3) $P_\varphi$ contains the definitions for the predicates $\mathtt{and}/6$ and $\mathtt{or}/6$ defined earlier.

(4) The only exported predicate in $P_\varphi$ is $\mathtt{p}/2$.

*Example* 5.1.3. Consider the RMBF of Example 4.1.1, with $\varphi = \langle \mathsf{eq}, \mathsf{a}\rangle$, where $\mathsf{eq}$ is given by

$$f(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge f(f(x_3 \vee x_2, x_3, x_1), x_3, x_1 \wedge x_2)$$

and $\mathsf{a} = \langle \mathbf{true}, \mathbf{false}, \mathbf{false}\rangle$. Then, $P_\varphi$ contains, apart from the definitions of the predicates $\mathtt{and}/6$ and $\mathtt{or}/6$, the following clauses:

```
f(X1l, X1r, X2l, X2r, X3l, X3r, Vl, Vr) :-
     or(X3l, X3r, X2l, X2r, U1l, U1r),
     f(U1l, U1r, X3l, X3r, X1l, X1r, U2l, U2r),
     and(X1l, X1r, X2l, X2r, U3l, U3r),
     f(U2l, U2r, X3l, X3r, U3l, U3r, U4l, U4r),
     and(X2l, X2r, U4l, U4r, U5l, U5r),
     or(X1l, X1r, U5l, U5r, Vl, Vr).

p(Ul, Ur) :- X = Y, f(X, Y, _, _, _, _, Ul, Ur).
```

LEMMA 5.1.4. *Let $P_\varphi$ be the program obtained from any (recursive) monotone Boolean formula $\varphi$ as described above, and let $Q$ be the query ?-* `p(U, V)` *where* `U` *and* `V` *are distinct variables. Then, the SLD-tree for the query $Q$ and the program $P_\varphi$ does not contain any failed execution branches.*

PROOF. From the definition of the function $F$ that generates $P_\varphi$, there are no nonvariable terms in any of the clauses in $P_\varphi$. Then, if the query $Q$ does not contain any nonvariable terms, then SLD-resolution cannot give rise to any nonvariable terms at any point. This implies that unification cannot fail at any point in the SLD-resolution. Hence the SLD-tree for the query $Q$ and the program $P_\varphi$ does not contain any failed branches. □

THEOREM 5.1.5. *Exact alias analysis of function-free, failure-free Horn programs is EXPTIME-complete.*

PROOF (SKETCH). The proof is by reduction from RMBF, which from Theorem 2.5.6 is EXPTIME-complete. Given an instance $\varphi$ of RMBF, a function-free program $P_\varphi$ whose execution mimics the evaluation of $\varphi$ can be generated, as described above: from Lemma 5.1.2 this construction takes time polynomial in the size of $\varphi$. Now consider a query ?- `p(U, V)` where `U` and `V` are distinct variables: from Lemma 5.1.4 the SLD-tree for this query does not contain any failed execution branches. Further, the evaluation of this query results in `U` and `V` becoming aliased together if and only if $\varphi$ is true (the details of this argument are essentially similar to that for Theorem 4.1.3). Thus, any analysis that gives exact results for function-free, failure-free Horn programs will infer that `U` and `V` are aliases after the evaluation of this query if and only if $\varphi$ is true. Since, from Theorem 2.5.6, RMBF is EXPTIME-complete, it follows that precise alias analysis of function-free, failure-free programs is EXPTIME-hard.

The argument that the problem is in EXPTIME is a generalization of that for Theorem 4.1.3. The main issue is that in this case we have to deal with nonground atoms. This can be done using an immediate-consequence operator based on nonground atoms, which can be obtained as a modification of the s-semantics presented by Falaschi et al. [1989]. □

It is straightforward to use the transformation discussed at the end of Section 3.2 to extend this result to programs where no clause has more than two body literals.

### 5.2 Function-Free Programs without Recursion

We next consider the complexity of dataflow analysis of function-free, failure-free programs if precision is sacrificed at recursive calls. To this end, we describe a procedure that, given two monotone Boolean formulae $\varphi$ and $\psi$, generates a logic program $P_{\varphi\psi}$ that can be used to establish whether or not $\varphi$ and $\psi$ are equivalent. The construction goes as follows:

(1) $P_{\varphi\psi}$ contains the clauses

```
choose_truth_val(X, X).
choose_truth_val(X, Y).
```

This simulates the two truth values that any variable in $\varphi$ and $\psi$ may take on.

(2) $P_{\varphi\psi}$ contains the clauses defining the predicates `and/6` and `or/6` given earlier.

(3) $P_{\varphi\psi}$ contains a clause for a predicate `p/4` defined as follows:

$\mathtt{p}(\mathtt{X}_{\varphi L}, \mathtt{X}_{\varphi R}, \mathtt{X}_{\psi L}, \mathtt{X}_{\psi R})$ :−
  $\mathtt{choose\_tval}(\mathtt{X}_{1L}, \mathtt{X}_{1R}), \ldots, \mathtt{choose\_tval}(\mathtt{X}_{nL}, \mathtt{X}_{nR}), G_\varphi, G_\psi.$

where $\mathcal{F}(\varphi) = \langle G_\varphi, \mathtt{X}_{\varphi L}, \mathtt{X}_{\varphi R}\rangle$, $\mathcal{F}(\psi) = \langle G_\psi, \mathtt{X}_{\psi L}, \mathtt{X}_{\psi R}\rangle$, $X_{\varphi L}$ and $X_{\varphi R}$ are not in $vars(G_\psi) \cup \{\mathtt{X}_{1L}, \mathtt{X}_{1R}, \ldots, \mathtt{X}_{nL}, \mathtt{X}_{nR}\}$, and where $X_{\psi L}$ and $X_{\psi R}$ are not in $vars(G_\varphi) \cup \{\mathtt{X}_{1L}, \mathtt{X}_{1R}, \ldots, \mathtt{X}_{nL}, \mathtt{X}_{nR}\}$.

(4) The only exported predicate in $P_{\varphi\psi}$ is `p/4`.

*Example* 5.2.1. Let $\varphi = x \wedge (y \vee z)$ and $\psi = (x \wedge y) \vee (y \wedge z) \vee x$; then the program $P_{\varphi\psi}$ contains, apart from the definitions of the predicates `and/6`, `or/6`, and `choose_tval/2`, the following predicate:

```
p(A_l, A_r, B_l, B_r) :-
     choose_truth_val(X_l, X_r),
     choose_truth_val(Y_l, Y_r),
     choose_truth_val(Z_l, Z_r),
     or(Y_l, Y_r, Z_l, Z_r, U_l, U_r),    /* begin Formula 1 */
     and(X_l, X_r, U_l, U_r, A_l, A_r),   /* end   Formula 1 */
     and(X_l, X_r, Y_l, Y_r, V_l, V_r),   /* begin Formula 2 */
     and(Y_l, Y_r, Z_l, Z_r, W_l, W_r),
     or(V_l, V_r, W_l, W_r, T_l, T_r),
     or(T_l, T_r, X_l, X_r, B_l, B_r).    /* end   Formula 2 */
```

LEMMA 5.2.2. *Let $\varphi$ and $\psi$ be any pair of monotone Boolean formulae; then $\varphi$ and $\psi$ are equivalent if and only if, given the program $P_{\varphi\psi}$ and the query*

?- $\mathtt{p}(\mathtt{X}_L, \mathtt{X}_R, \mathtt{Y}_L, \mathtt{Y}_R)$

*the analysis infers that $\mathtt{X}_L$ and $\mathtt{X}_R$ are aliased at the point immediately after the query if and only if $\mathtt{Y}_L$ and $\mathtt{Y}_R$ are aliased.*

PROOF. The proof proceeds via a straightforward structural induction to show that given a monotone Boolean formula $\varphi$ and a truth assignment for its variables (in the form of certain pairs of variables being aliased or not) the evaluation of the goal $G$, where $\mathcal{F}(\varphi) = \langle G, X_L, X_R\rangle$, results in $X_L$ and $X_R$ becoming aliased together if and only if $\varphi$ evaluates to true for that truth assignment. It follows immediately that given two monotone Boolean formulae $\varphi$ and $\psi$ these formulae are equivalent if and only if for every possible choice of truth assignments to the variables occurring in these formulae the evaluation of the goal '$G_\varphi, G_\psi$', where

$\mathcal{F}(\varphi) = \langle G_\varphi, X_L, X_R \rangle$ and $\mathcal{F}(\psi) = \langle G_\psi, Y_L, Y_R \rangle$, results in $X_L$ and $X_R$ becoming aliased together when and only when $Y_L$ and $Y_R$ become aliased. Suppose the set of variables occurring in $\varphi$ and $\psi$ is $\{U_1, \ldots, U_n\}$; then this is equivalent to saying that for every branch in the SLD-tree for the goal

$$\texttt{choose\_tval}(U_{1L}, U_{1R}), \ldots, \texttt{choose\_tval}(U_{nL}, U_{nR}), G_\varphi, G_\psi$$

$X_L$ and $X_R$ are aliases if and only if $Y_L$ and $Y_R$ are aliases. Given the definition of the predicate p/4 in the program $P_\varphi$, the lemma follows readily from this. □

We are now in a position to prove the following result:

THEOREM 5.2.3. *Exact alias analysis for recursion-free, function-free, failure-free, bounded-arity programs is co-NP-complete. It remains co-NP-complete even if no predicate has arity exceeding 6.*

PROOF. The proof is by reduction from the Equivalence problem for monotone Boolean formulae, which, from Theorem 2.5.4, is co-NP-complete. Given any two monotone Boolean formulae $\varphi$ and $\psi$, it follows from Lemma 5.1.2 that the program $P_{\varphi\psi}$ can be constructed in time polynomial in the size of $\varphi$ and $\psi$. It follows, from Lemma 5.2.2, that exact alias analysis for recursion-free, function-free, failure-free, bounded arity programs is co-NP-hard.

To show that the problem is in co-NP, we show that the complement of the problem is in NP. To this end, consider any recursion-free, function-free, failure-free, bounded arity program: we wish to show that if there is some execution branch of the program which aliases together two variables $X_L$ and $X_R$ but which leaves two other variables $Y_L$ and $Y_R$ unaliased (intuitively, such an execution branch corresponds to a "witness" that two formulae are not equivalent), then this can be found in polynomial time by a nondeterministic Turing machine. It is possible to "guess" nondeterministically an execution path that aliases $X_L$ and $X_R$ but not $Y_L$ and $Y_R$ by guessing, for each literal in the program, which clause of the corresponding predicate to choose, and then verify that this execution path does, in fact, cause the appropriate aliasing behavior. Since there is no recursion, the total length of such an execution path is linear in the size of the program. Since the programs under consideration have bounded arity, the total number of variables that may involved is also linear in the size of the program. Thus, the verification step can be carried out in time polynomial in the size of the program. The theorem follows. □

## 5.3 Groundness Analysis

Groundness analyses seek to determine which variables are guaranteed to be bound to ground terms at various program points. In general, this requires reasoning about the groundness of arguments to predicates, and about sharing and aliasing between these arguments, at the time of a call and when that call returns. It turns out that even if all we have is an analysis that only gives information about groundness of different argument positions at entry and exit from a predicate (i.e., does not provide any sharing or aliasing information), it is possible to extract enough aliasing information to obtain the following results:

THEOREM 5.3.1. *Exact groundness analysis for function-free, failure-free Horn programs is EXPTIME-complete.*

PROOF. The proof is by reduction from RMBF, and is essentially similar to that of Theorem 5.1.5. Given an instance $\varphi$ of RMBF, a function-free program $P_\varphi$ whose execution mimics the evaluation of $\varphi$ can be generated in polynomial time, as described earlier. Now consider a query ?- `p(U, V)`, where `U` and `V` are distinct variables. Any groundness analysis that gives exact results for function-free, failure-free Horn programs will infer that the groundness of `U` and `V` are equivalent after the evaluation of this query—i.e., `U` is ground if and only if `V` is ground—if and only if $\varphi$ is true. It follows, from Theorem 2.5.6, that precise groundness analysis of function-free, failure-free programs is EXPTIME-hard. The argument that the problem is in EXPTIME is similar to that in the proof of Theorem 5.1.5. □

Similarly, proceeding as in Theorem 5.2.3, it is not difficult to show the following:

THEOREM 5.3.2. *Exact groundness analysis for recursion-free, function-free, failure-free, bounded-arity programs is co-NP-complete.*

While most groundness and alias analyses proposed in the logic programming literature are not precise enough to include the class of recursion-free, function-free, failure-free, bounded-arity programs in their exactness sets, we know of at least one analysis algorithm, proposed by Marriott et al. [1994], that appears to be precise enough to give exact results for this class of programs. This analysis requires the solution of a co-NP-hard problem at each iteration to determine whether a fixpoint has been attained (the analysis manipulates propositional formulae containing the connectives $\wedge$, $\vee$, and $\Leftrightarrow$, and checking to see whether a fixpoint has been attained involves deciding the equivalence of two such formulae: this is a proper generalization of the equivalence problem for monotone Boolean formulae, and therefore, from Theorem 2.5.4, is co-NP-hard). However, this source of complexity is absent in recursion-free programs, and one may reasonably inquire after the worst-case complexity of this analysis in this case. Thus, Theorem 5.3.2 illustrates that there are at least two independent sources of complexity in this algorithm, a fact that is by no means obvious from the description of the algorithm. This demonstrates how our techniques can be used to separate out different sources of complexity in an analysis algorithm.

## 6. BOUNDING THE NUMBER OF CALLING AND SUCCESS PATTERNS

One possible reason an analysis may be imprecise is that, for efficiency reasons, it may not keep track of all the different calling and/or success patterns it encounters for a predicate—it may instead compute a single worst-case summary for each predicate to obtain a conservative approximation to these sets. Such a strategy is used in a number of analyses, for example, Chang et al. [1985], Debray and Warren [1988], and Mellish [1985]: it does not appear too unreasonable if we believe that for most programs encountered in practice, predicates have specific argument positions used consistently with the same mode, type, etc. (see, for example, Drabent [1987]). In this section, we examine whether such a strategy can lead to fundamental improvements in the worst-case behavior of analysis algorithms.

If an analysis approximates a set of calling and/or success patterns by their least upper bound in the abstract domain, there will be a loss in precision in general. However, if every predicate in the program under consideration has at most one calling pattern and one success pattern, then there is no loss of precision due to this approximation. To examine complexity issues for analyses that compute worst-case approximations in this way, we focus on programs where each predicate has a bounded number of calling and success patterns. Of course, whether or not this is true for a given program depends partly on the program, and partly on the abstract domain under consideration, since a single calling pattern for one abstract domain may correspond to a number of different calling patterns for a different abstract domain that is larger and has finer granularity. However, when we refer to an analysis algorithm we assume implicitly that the algorithm is defined with respect to some specific abstract domain, so it makes sense to talk about the class of programs that, for a given analysis, have a bounded number of calling and/or success patterns.

The first such restriction we consider is described by Codish et al. [1990], who describe a bottom-up analysis that maintains at most one success pattern per program clause (since the analysis is a bottom-up one, there is no notion of calling patterns). The worst-case complexity of the resulting analysis is $O(2^n)$, where $n$ is the size of the program [Codish et al. 1990]. We show that unless $P = NP$, it may not be possible to do better than this:

THEOREM 6.1. *Exact groundness and alias analyses of recursion-free, function-free, failure-free programs, where the number of distinct calling and success patterns for any predicate is $O(1)$, is co-NP-complete.*

PROOF. Identical to that of Theorem 5.2.3. We observe that in the program $P_{\varphi\psi}$ constructed there, each predicate has arity at most 6, and so for an abstract domain of fixed size $k$ can have at most $O(k^6) = O(1)$ calling and success patterns. The result follows. □

If, however, variable dependency information is used in a fairly limited way, it is possible to obtain analysis algorithms with polynomial-time worst-case complexity. One of the properties that complicates the handling of sharing among variables is that sharing is not transitive in general: the fact that a program variable $X$ may share variables with a program variable $Y$ at runtime, and that $Y$ may share variables with a program variable $Z$, does not imply that $X$ shares with $Z$. The representation and processing of sharing and dependency information can be simplified considerably, at the cost of some precision, by assuming that sharing is transitive. Define the class of *transitive-sharing* programs to be those programs where all sharing among variables is transitive. Then, the results of Debray [1989] imply the following:

THEOREM 6.2. *There exist polynomial-time dataflow analysis algorithms whose exactness sets are contained in the class of function-free, failure-free, bounded-arity, transitive-sharing programs where each predicate has at most one calling and one success pattern.*

If aliasing can be disregarded entirely, analyses can be carried out quite efficiently: it is shown, in Debray [1992], that abstract domains that allow aliasing to be ignored

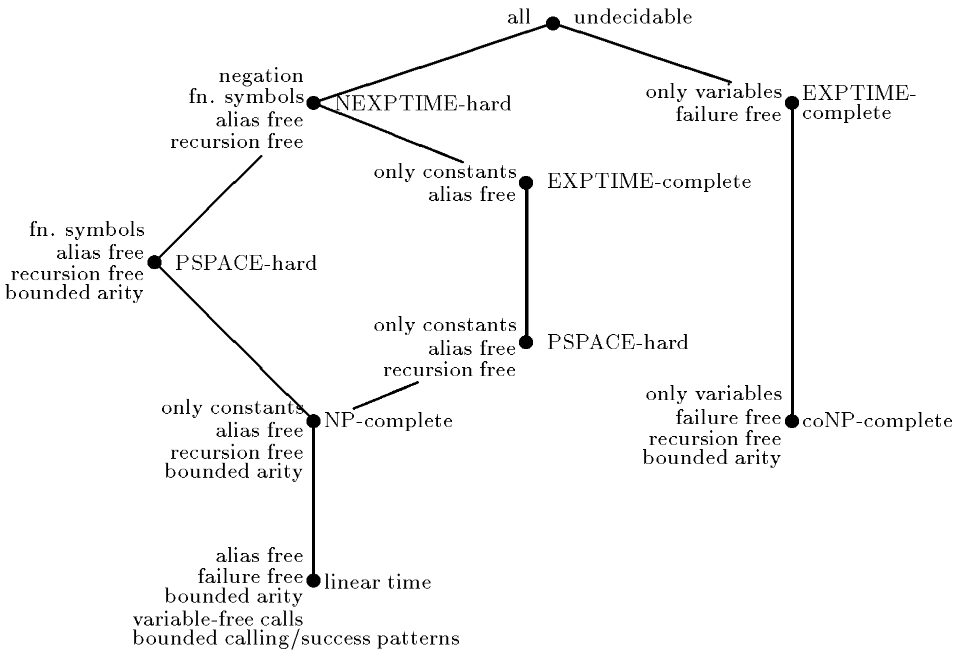


Fig. 1. The correlation between precision and complexity in dataflow analysis of logic programs.

admit sound flow analysis algorithms whose worst-case complexity, assuming $O(1)$ calling and success patterns per predicate, is linear in program size. Define a program to satisfy the *variable-free calls* property if none of the calls arising during the execution of that program contain variables. We have the following result [Debray 1992]:

THEOREM 6.3. *There exist linear-time dataflow analysis algorithms whose exactness sets are contained in the class of alias-free, failure-free, bounded-arity programs that satisfy the variable-free calls property and where each predicate has at most one calling and one success pattern.*

Analyses satisfying these requirements include the *rigid type* analysis of Janssens [1990], and Sato and Tamaki's depth-abstraction analysis [Sato and Tamaki 1984].

## 7. CONCLUSIONS

While it is generally believed that there is a correlation between complexity and precision of flow analysis algorithms, in the sense that "sufficiently precise" analyses must also be correspondingly expensive, little work appears to have been done on quantifying this correlation. This article takes a step toward formally addressing this issue in the context of logic programming. We offer a formal characterization of the "precision" of dataflow analyses. We consider the implications, with regard to the precision/complexity tradeoff, of some of the different ways in which various analysis algorithms may sacrifice precision. Our results, which are summarized in Figure 1, indicate, somewhat surprisingly, that even for classes of programs whose syntactic structure is extremely simple, the worst-case complexity of precise dataflow analysis can be exponential in the program size for a wide variety of dataflow analyses. The implication is that the complexity cannot be any better when considering the entire Prolog language, even for relatively modest dataflow analyses of logic programs.

ACKNOWLEDGMENTS

I am grateful to Steve Mahaney and Sampath Kannan for patiently instructing me in the elements of complexity theory. Discussions with Nevin Heintze and Niels Jørgensen were very helpful in crystallizing the ideas presented here. The results of Lewis [1980], on which Theorem 3.2.5 is based, were brought to my attention by Mark Johnson. Manuel Hermenegildo, Dean Jacobs, Anno Langen, Kim Marriott, and Harald Søndergaard graciously tolerated, and responded to, innumerable questions regarding the precision of their flow analysis algorithms. Pascal Van Hentenryck pointed out a number of errors in an earlier version of this article. Comments by the anonymous referees helped improve both the contents and the presentation of the article.

REFERENCES

Aiken, A. and Lakshman, T. K. 1994. Directional type checking of logic programs. In *Proceedings of the 1st International Static Analysis Symposium*. Springer-Verlag, New York, 43–60.

Apt, K. R. and Doets, K. 1994. A new definition of SLDNF-resolution. *J. Logic Program. 18,* 2 (Feb.), 177–190.

Apt, K. R. and van Emden, M. H. 1982. Contributions to the theory of logic programming. *J. ACM 29,* 3 (July), 841–862.

Barbuti, R. and Martelli, M. 1988. A tool to check the non-floundering logic programs and goals. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming*, P. Deransart, B. Lorho, and J. Małuszynski, Eds. Springer-Verlag, New York, 58–67.

Bloniarz, P. A., Hunt, H. B., III, and Rosenkrantz, D. J. 1984. Algebraic structures with hard equivalence and minimization problems. *J. ACM 31,* 4 (Oct.), 879–904.

Bruynooghe, M. 1986. Compile time garbage collection. In *Proceedings of the IFIP Working Conference on Program Transformation and Verification*. Elsevier, Amsterdam.

Chang, J., Despain, A. M., and DeGroot, D. 1985. And-parallelism of logic programs based on a static data dependency analysis. In *Digest of Papers, Compcon 85*. IEEE Computer Society, Washington, D.C., 218–225.

Codish, M. and Demoen, B. 1993. Analysing logic programs using "prop"-ositional logic programs and a magic wand. In *Proceedings of the 1993 International Symposium on Logic Programming*. MIT Press, Cambridge, Mass., 114–129.

Codish, M., Dams, D., and Yardeni, Y. 1990. Abstract unification and a bottom-up analysis to detect aliasing in logic programs. Tech. Rep. CS90-10, Dept. of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. May.

Codish, M., Falaschi, M., and Marriott, K. 1994. Suspension analyses for concurrent logic programs. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 649–686.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*. ACM, New York, 151–158.

Cortesi, A., Le Charlier, B., and Van Hentenryck, P. 1994. Combinations of abstract domains for logic programming. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*. ACM, New York, 227–239.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*. ACM, New York, 269–282.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*. ACM, New York, 238–252.

Debray, S. K. 1992. Efficient dataflow analysis of logic programs. *J. ACM 39,* 4 (Oct.), 949–984.

Debray, S. K. 1989. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst. 11,* 3 (July), 419–450.

Debray, S. K. and Warren, D. S. 1988. Automatic mode inference for logic programs. *J. Logic Program. 5,* 3 (Sept.), 207–229.

Drabent, W. 1987. Do logic programs resemble programs in conventional languages? In *Proceedings of the IEEE Symposium on Logic Programming*. IEEE, New York, 389–396.

Falaschi, M., Levi, G., Palamidessi, C., and Martelli, M. 1989. Declarative modeling of the operational behavior of logic languages. *Theoret. Comput. Sci. 69*, 3, 289–318.

Giacobazzi, R., Debray, S. K., and Levi, G. 1992. A generalized semantics for constraint logic programs. In *Proceedings of the 1992 Conference on Fifth Generation Computer Systems*. ICOT, Tokyo, 582–591.

Hermenegildo, M., Warren, R., and Debray, S. K. 1992. Global flow analysis as a practical compilation tool. *J. Logic Program. 13,* 4 (Aug.), 349–366.

Hudak, P. and Young, J. 1986. Higher-order strictness analysis in the untyped lambda calculus. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM, New York, 97–109.

Jacobs, D. and Langen, A. 1989. Accurate and efficient approximation of variable aliasing in logic programs. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, Cambridge, Mass., 154–165.

Janssens, G. 1990. Deriving run-time properties of logic programs by means of abstract interpretation. Ph. D. thesis, Katholieke Universiteit Leuven, Belgium.

Jones, N. D. and Muchnick, S. S. 1981. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall, Englewood Cliffs, N.J., 381–393.

King, A. 1994. Depth-$k$ sharing and freeness. In *Proceedings of the 11th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 553–568.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM Trans. Program. Lang. Syst. 16,* 1 (Jan.), 35–101.

Lewis, H. R. 1980. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci. 21*, 317–353.

Mannila, H. and Ukkonen, E. 1987. Flow analysis of Prolog programs. In *Proceedings of the 4th IEEE Symposium on Logic Programming*. IEEE, New York, 205–214.

Marriott, K. and Søndergaard, H. 1993. Precise and efficient groundness analysis of logic programs. *ACM Lett. Program. Lang. Syst. 2,* 1–4 (Mar.–Dec.), 181–196.

Marriott, K. and Søndergaard, H. 1992. Bottom-up dataflow analysis of normal logic programs. *J. Logic Program. 13,* 2–3 (July), 181–204.

Marriott, K. and Søndergaard, H. 1988. Bottom-up abstract interpretation of logic programs. In *Proceedings of the 5th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 733–748.

Marriott, K., Søndergaard, H., and Dart, P. 1990. A characterization of non-floundering logic programs. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, Cambridge, Mass., 661–680.

MARRIOTT, K., SØNDERGAARD, H., AND JONES, N. D. 1994. Denotational abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 607–648.

MELLISH, C. S. 1985. Some global optimizations for a Prolog compiler. *J. Logic Program. 2,* 1 (Apr.), 43–66.

MULKERS, A., WINSBOROUGH, W., AND BRUYNOOGHE, M. 1994. Live-structure dataflow analysis for Prolog. *ACM Trans. Program. Lang. Syst. 16,* 2 (Mar.), 205–258.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the 8th International Conference on Logic Programming.* MIT Press, Cambridge, Mass., 49–63.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989. Determination of variable dependence information at compile time through abstract interpretation. In *Proceedings of the North American Conference on Logic Programming.* MIT Press, Cambridge, Mass., 166–185.

MYERS, E. W. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages.* ACM, New York, 219–230.

SAGIV, Y. AND ULLMAN, J. D. 1984. Complexity of a top-down capture rule. Tech. Rep. STAN-CS-84-1009, Dept. of Computer Science, Stanford Univ., Stanford, Calif. July.

SANTOS COSTA, V., WARREN, D. H. D., AND YANG, R. 1991. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *Proceedings of the 11th International Conference on Logic Programming.* MIT Press, New York, 443–456.

SATO, T. AND TAMAKI, H. 1984. Enumeration of success patterns in logic programs. *Theoret. Comput. Sci. 34,* 227–240.

STOCKMEYER, L. J. AND MEYER, A. R. 1973. Word problems requiring exponential time. In *Proceedings of the 5th ACM Symposium on Theory of Computing.* ACM, New York, 1–9.

TÄRNLUND, S. 1977. Horn clause computability. *BIT 17,* 215–226.

TAYLOR, A. 1989. Removal of dereferencing and trailing in Prolog compilation. In *Proceedings of the 6th International Conference on Logic Programming.* MIT Press, Cambridge, Mass., 48–60.

ULLMAN, J. D. 1988. The complexity of ordering subgoals. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems.* ACM, New York, 74–81.

VAN HENTENRYCK, P., CORTESI, A., AND LE CHARLIER, B. 1994. Type analysis of Prolog using type graphs. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM, New York, 337–348. Extended version to appear in *J. Logic Program.*

WINSBOROUGH, W. 1988. Automatic, transparent parallelization of logic programs at compile time. Ph. D. thesis, Computer Science Dept., Univ. of Wisconsin, Madison.

Received January 1994; revised December 1994; accepted January 1995