Denotational Abstract Interpretation of Logic Programs

KIM MARRIOTT
Monash University

HARALD SØNDERGAARD
The University of Melbourne
and
NEIL D. JONES
DIKU, University of Copenhagen

Logic-programming languages are based on a principle of separation of "logic" and "control." This means that they can be given simple model-theoretic semantics without regard to any particular execution mechanism (or proof procedure, viewing execution as theorem proving). Although the separation is desirable from a semantical point of view, it makes sound, efficient implementation of logic-programming languages difficult. The lack of "control information" in programs calls for complex data-flow analysis techniques to guide execution. Since data-flow analysis furthermore finds extensive use in error-finding and transformation tools, there is a need for a simple and powerful theory of data-flow analysis of logic programs.

This paper offers such a theory, based on F. Nielson's extension of P. Cousot and R. Cousot's *abstract interpretation*. We present a denotational definition of the semantics of definite logic programs. This definition is of interest in its own right because of its compactness. Stepwise we develop the definition into a generic data-flow analysis that encompasses a large class of data-flow analyses based on the SLD execution model. We exemplify one instance of the definition by developing a provably correct groundness analysis to predict how variables may be bound to ground terms during execution. We also discuss implementation issues and related work.

Categories and Subject Descriptors: D.1.6 [**Programming Languages**]: Logic Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*;

H. Søndergaard has been supported by The Australian Research Council. The work of N. D. Jones is partially supported by grants from the European Community and the Danish Research Council.
Authors' addresses: K. Marriott, Department of Computer Science, Monash University, Clayton, Victoria, 3168, Australia; email: marriott@cs.monash.oz.au; H. Søndergaard, Department of Computer Science, The University of Melbourne, Parkville, Victoria, 3052, Australia; email: harald@cs.mu.oz.au; N. D. Jones, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen ϕ, Denmark.

invariants; *logics of programs*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Abstract interpretation, Boolean functions, data-flow analysis, global analysis, groundness analysis

1. INTRODUCTION

Data-flow analysis is an essential component of many programming tools. The main use of data-flow information is in compilers and other program transformers, where the analysis may guide various improvements of the generated code. Another use is to identify errors in a program, as done by program "debuggers" and type checkers. Stated generally, the purpose of program analysis is to decide whether some *invariant* holds at some *program point*. It may thereby be determined whether some transformation scheme is applicable and possibly what the exact form of the synthesis should be. The process of investigating such invariance is called *data-flow analysis*.

A fruitful way of viewing data-flow analysis was suggested some 30 years ago, according to which data-flow analysis is "pseudoevaluation," that is, a process that somehow mimics the normal execution of a program. Naur put this point of view to good use in explaining the type-checking component of the Gier Algol compiler, and Sintzoff later provided further examples of its usefulness (we give references later). P. Cousot and R. Cousot formalized the idea in their influential theory of *abstract interpretation*.

We later discuss abstract interpretation in more detail, but the following example conveys the basic idea: Rather than using integers as data objects, a data-flow analysis may use *neg*, *zero*, and *plus* to describe negative integers, 0, and positive integers, respectively. Then by reinterpreting operations like multiplication according to the "rules of signs," the data-flow analysis may establish certain properties of a program, such as "whenever control reaches this loop, x is assigned a negative value."

Abstract interpretation prescribes certain relations that should hold between a data-flow analysis and the semantics of the programming language in question. If these relations hold, the data-flow analysis is guaranteed to be correct. To formalize them, however, precise formal definitions of both semantics and data-flow analysis are required. The analysis-as-pseudoevaluation view is usually reified as a strong similarity between the two definitions, a certain degree of congruence. This naturally leads to viewing data-flow analysis simply as *nonstandard semantics*.

Abstract interpretation of *logic programs* has gained considerable currency during the last 10 years. The major impetus has been the quest for data-flow analyses that can improve code generation in Prolog compilers. Logic-programming languages are based on a principle of "separation of logic and control," which is desirable from a semantical point of view, but which also causes severe problems for implementation. The lack of "control information"

in programs provides a wide scope for compiler optimizations and, hence, data-flow analysis of these languages. This, and the simple semantics of logic programs, explains the currency that abstract interpretation has gained in logic programming. However, data-flow analysis of logic programs is in some ways more complex than analysis of more traditional languages, since data-flow is bidirectional (owing to unification) and control flow is in terms of backtracking, at least for top-down sequential execution.

Much of the research into abstract interpretation of logic programs has been devoted to designing abstract interpretation "frameworks" that consist of a generic data-flow algorithm with a few basic operations as parameters. A particular analysis is obtained by providing these parametric functions. An important property of a framework is that correctness of the resultant analysis is assured as long as the parametric functions correctly approximate the standard interpretation of these parametric functions. Such frameworks facilitate the development of structurally similar analyses and their proofs of correctness, and also allow different analyses to be easily implemented by means of a common "analysis engine."

Frameworks, however, though generic in a certain sense, are implicitly based on a single set of semantic equations that reflect the operational semantics of the language being analyzed and the particular information required. In fact, almost all existing frameworks model the SLD semantics of definite logic programs and collect information about the calls occurring at run time. A single framework is not suitable for all data-flow applications in logic programming. For example, if the operational semantics is changed, say, to a "bottom-up" evaluation or to allow dynamic atom scheduling, or if the language is extended, say, to allow constraints or negation, then the underlying semantic equations are changed, and so a new framework must be designed and proved correct.

Here we give a general abstract interpretation theory in which to compare and design these frameworks and their instances, that is, specific data-flow analyses. The key idea, due to Nielson, is to express the underlying semantic equations of each framework in a common *metalanguage*. The main pragmatic advantage is that, by a careful choice of the metalanguage, properties that are important for proving correctness of data-flow analysis can be established at the level of the metalanguage once and for all, rather than established for each framework or analysis. A "universal" metalanguage also facilitates a stepwise, derivational approach to the development of data-flow analyses. Note that these semantic equations can, with the right interpreter, be directly executed, providing a prototype analysis engine that, in turn, can be instantiated to perform various data-flow analyses.

We illustrate the use of our theory by defining an abstract interpretation framework for definite logic programs that models the standard SLD operational semantics and collects information about a particular query's answers. The definition can be easily extended to collect information about calls. Our semantic definition is considerably cleaner than previous definitions that have been suggested for SLD-style semantics. Its simplicity is mainly due to our use of constraints instead of classical substitutions, as this avoids tradi-

tional problems with renaming and allows us to express unification and composition as simple lattice operations. Finally, we use our abstract interpretation framework to define a simple yet highly precise groundness analysis. This analysis makes use of Boolean functions to trace the interrelation of groundness among variables.

In Section 2 we recapitulate some basic notions. In Section 3 we present the data-flow-analysis-as-approximate-computation view. Section 4 contains an introduction to "language-independent" abstract interpretation. In Section 5 we develop a denotational definition that captures the essence of SLD resolution, as seen from the point of view of many data-flow analyses. This definition is of some interest already because of its avoidance of substitutions and standardizing apart. In Section 6 we turn the semantic definition into a generic definition of a *data-flow semantics*, whose instances define a wide class of useful data-flow analyses for logic programs. As an example we define a groundness analysis in Section 7. In Section 8 we discuss the implementation of generic data-flow analyzers. In Section 9 we discuss related work, and Section 10 contains a concluding discussion.

Much of the material in this paper was first presented as a tutorial on abstract interpretation given at the North American Conference on Logic Programming in Cleveland, Ohio, in 1989. The current exposition is an extensively revised version of a paper presented at the Italian Logic Programming Conference (GULP) in 1990.

Readers are expected to be familiar with the theory of logic programming, and with denotational semantics at the level of the textbooks by Lloyd [1987] and Schmidt [1986], for example. We have tried to comply with the terminology of these two books. For a general introduction to abstract interpretation, see Abramsky and Hankin [1987].

2. PRELIMINARIES

In this section we recapitulate some basic notions and facts from domain theory and explain some notation that will be used through the paper.

A *preordering* on X is a binary relation that is reflexive and transitive. A *partial ordering* is a preordering that is antisymmetric. A set equipped with a partial ordering is a *poset*. Let $(X, \leq)$ be a poset. A (possibly empty) subset Y of X is a *chain* iff for all $y, y' \in Y$, $y \leq y' \vee y' \leq y$.

Let $(X, \leq)$ be a poset, and let Y be a subset of X. An element $x \in X$ is an *upper bound* for Y iff $y \leq x$ for all $y \in Y$. Dually, we may define a *lower bound* for Y. An upper bound x for Y is the *least upper bound* for Y iff, for every upper bound x' for Y, $x \leq x'$, and when it exists, we denote it by $\sqcup Y$. Dually, a lower bound x for Y is the *greatest lower bound* for Y iff, for every lower bound x' for Y, $x' \leq x$. When it exists, we denote the greatest lower bound for Y by $\sqcap Y$.

A poset for which every subset possesses a least upper bound and a greatest lower bound is a *complete lattice*. In particular, equipped with the subset ordering, the powerset of X, denoted by $\mathcal{P}X$, is a complete lattice. Let

X be a complete lattice. We denote $\sqcup\emptyset = \sqcap X$ by $\bot_X$ and $\sqcap\emptyset = \sqcup X$ by $\top_X$. The complete lattice X is *Noetherian* iff every ascending chain in X is finite.

Functions are generally used in their Curried form. Our notation for function application uses parentheses sparingly. We use redundant parentheses only when they would seem to help the eye. As usual, function space formation $X \to Z$ associates to the right, and function application to the left. We occasionally use Church's lambda notation for functions, and we use "$\circ$" for composition of functions.

Let $(X, \preceq)$ and $(Z, \preceq)$ be posets. A function $F\colon X \to Z$ is *monotonic* iff $x \preceq x' \Rightarrow Fx \preceq Fx'$ for all $x, x' \in X$. In what follows, monotonicity of functions is essential, so much that it is understood throughout this paper that $X \to Z$ denotes the space of *monotonic* functions from X to Z. Furthermore, the function space $X \to Z$ will always be ordered pointwise; that is, if $F, G\colon X \to Z$ are functions and the ordering on Z is $\preceq$, then the ordering $\sqsubseteq$ on $X \to Z$ is defined by $F \sqsubseteq G$ iff $\forall x \in X.Fx \preceq Gx$.

A function $F\colon X \to Z$ is *injective* iff $Fx = Fx' \Rightarrow x = x'$ for all $x, x' \in X$; and F is *bijective* iff there is a function $F'\colon Z \Rightarrow X$ such that $F \circ F'$ and $F' \circ F$ are identity functions.

Let X and Z be complete lattices. A function $F\colon X \to Z$ is *strict* iff $F\bot_X = \bot_Z$. Dually, F is *costrict* iff $F\top_X = \top_Z$.

A *fixpoint* for a function $F\colon X \to X$ is an element $x \in X$ such that $x = Fx$. If X is a complete lattice, then the set of fixpoints for (the monotonic) $F\colon X \to X$ is itself a complete lattice. The least element of this lattice is the *least fixpoint* for F, denoted by *lfp* F. Furthermore, defining

$$F \uparrow \alpha = \begin{cases} \sqcup\{F \uparrow \alpha' \mid \alpha' < \alpha\} & \text{if } \alpha \text{ is a limit ordinal,} \\ F(F \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal,} \end{cases}$$

there is some ordinal α such that $F \uparrow \alpha = lfp\ F$.[1] The sequence $(F \uparrow 0)$, $(F \uparrow 1), \ldots, (lfp\ F)$ is the (*completed*) *Kleene sequence* for F. If a monotonic function is defined on a *Noetherian* lattice, then it has a finite Kleene sequence.

Let X be a complete lattice. A predicate Q is *inclusive* on X iff, for all (possibly empty) chains $Y \subseteq X$, $Q(\sqcup Y)$ holds whenever (Qy) holds for every $y \in Y$. Inclusive predicates are admissible in *fixpoint induction*: Assume that $F\colon X \to X$ is monotonic and that Qx implies that $Q(Fx)$ for all $x \in X$. If Q is inclusive, then $Q(lfp\ F)$ holds.

Finally, a note about the "semantic brackets" $[\![$ and $]\!]$: We use these for quasi-quotation generally; that is, in this paper they are simply "Quine corners."

3. APPROXIMATE COMPUTATION

Abstract interpretation formalizes data-flow analysis by viewing it as *approximate computation*, in which one manipulates *descriptions* of data rather than the data themselves. In this section we detail this idea. In Section 4 we

[1] The origin of this assertion is in Kleene's first recursion theorem [Kleene 1952].

explain how it can be formalized in the theory of denotational abstract interpretation.

In general, the disadvantage of an approximate computation is that the result it yields is not as precise as that of a proper computation. But this is compensated for by two important advantages: First, one approximate computation may yield information about many proper computations at once. Second, approximate computation is usually much faster than the proper computation. In our case, where the concern is with the approximate computation of programs, the difference in speed between proper and approximate computation may be extreme: nontermination versus termination.

The idea of performing program analysis by approximate computation appeared very early in computer science. Naur identified the idea and applied it in work on the Gier Algol compiler in the early 1960s [Naur 1963]. He coined the term *pseudoevaluation* for what would later be described as "a process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not on their values" [Jensen 1965]. The same basic idea is found in work by Reynolds [1969] and by Sintzoff [1972]. Sintzoff used it for proving a number of well-formedness aspects of programs in an imperative language and for verifying termination properties.

By the mid-1970s, efficient data-flow analysis had been studied rather extensively by researchers such as Kam, Kildall, Tarjan, Ullman, and others (for references, see Hecht [1977]). As a powerful attempt to unify much of that work, a precise framework for discussing approximate computation (of imperative programs) was developed by P. Cousot and R. Cousot [1977; 1979]. The advantage of such a unifying framework is that it serves as a basis for understanding various data-flow analyses better, including their interrelation, and for discussing their correctness.

The overall idea of Cousot and Cousot was to define a "static" semantics that associates with each program point the set of possible storage states that may obtain at run time whenever execution reaches that point. A data-flow analysis can then be construed as a finitely computable approximation to the static semantics. The work of Cousot and Cousot later was extended to declarative languages. Such extensions are not straightforward. For example, there is no clear-cut notion of "program point" in functional or logic programs. Also, data-flow analysis of programs written in a programming language such as Prolog differs somewhat from the analysis of programs written in more conventional programming languages, because the data-flow is bidirectional, owing to unification, and the control flow is more complex, owing to backtracking.

Of the applications of abstract interpretation in functional programming, we mention work by Jones [1981] and by Jones and Muchnick [1981] on termination analysis for lambda expressions and, in a LISP setting, improved storage allocation schemes through reduced reference counting. The main application, though, has been *strictness analysis*, which is concerned with the problem of determining cases where applicative order may be safely used instead of normal order execution. The study of strictness analysis was

Table I. Multiplication of Parities

$\otimes$	$\bot$	odd	even	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
odd	$\bot$	odd	even	$\top$
even	$\bot$	even	even	even
$\top$	$\bot$	$\top$	even	$\top$

initiated by Mycroft [1981], and the literature on the subject is now quite extensive; see, for example, Nielson [1988].

We can describe approximate computation as the evaluation of a formula or a program, not over its standard domain, but over a nonstandard domain of "descriptions." The standard domain may consist of "states," sets of terms, atomic formulas, substitutions, or whatever, depending on how the semantics is modeled; the nonstandard domain is determined by the kind of program properties that we want to expose. Of course, when performing approximate computations, one must reinterpret all operators so as to apply to descriptions rather than to proper values.

Assume we have a data domain U and operators on U. In the example below, U will be $\mathbb{Z}$, the set of integers. We also assume we have a set X of descriptions. To be precise about the relation between values and descriptions, a *concretization function* $\gamma\colon X \to \mathcal{P}U$ is defined.[2] For every description $x \in X$, (γx) is the set of objects that x describes. Thus, γ is the semantic function for descriptions.

Example 3.1. Let X be the set $\{\bot, even, odd, \top\}$, ordered by $x \leq x'$ iff $x = \bot \vee x = x' \vee x' = \top$. The denotation of description $x \in X$ is given by $\gamma\colon X \to \mathcal{P}\mathbb{Z}$ by

$$\begin{aligned}\gamma\ \bot &= \emptyset,\\ \gamma\ even &= \{z \in \mathbb{Z} \mid z \text{ is even}\},\\ \gamma\ odd &= \{z \in \mathbb{Z} \mid z \text{ is odd}\},\\ \gamma\ \top &= \mathbb{Z}.\end{aligned}$$

Note that descriptions are ordered according to how large the set of objects they apply to is: The more imprecise, the "higher" they sit in the structure. The set $\{2, 4\}$ is approximated by the description *even*, the set $\{2, 3\}$ by $\top$, and multiplication of sets of integers by the operator $\otimes\colon X \times X \to X$, given in Table I. The example, incidentally, illustrates that application of an operator to a description of "everything," that is, $\top$, need not involve loss of information: The operator $\otimes$ is not costrict since, for example, $\top \otimes even = even$. Also note that the ordering on descriptions is, in a sense, opposite to the ordering used in domain theory: For descriptions, the *top* element corresponds to total lack of information.

[2] The terminology, including the use of the symbol γ, is due to P. Cousot and R. Cousot [1977].

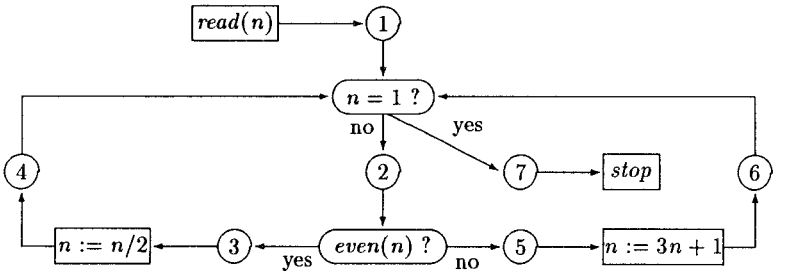

Fig. 1. Flow diagram.

To see how a computation using the descriptions in Example 3.1 can proceed, consider the flow diagram in Figure 1.[3] Program points are numbered edges. Descriptions can be propagated in the graph by appropriately interpreting the commands according to the "rules of parities." For example, the fact that n is odd at program point 5 is used to conclude that n is even at point 6. Such reasoning can easily be mechanized. Assuming that n is $\top$ at point 1, we get the following descriptions of n at other points: At 2 it is $\top$, at 3 it is *even*, at 4 it is $\top$, at 5 it is *odd*, at 6 it is *even*, and at 7 it is *odd*. This information justifies transformation of the program so as to avoid a number of tests; for example, the statement $n := 3n + 1$ can be replaced by $n := (3n + 1)/2$.

The above example is a simple case of data-flow analysis viewed as approximate computation. In general, there are three parameters in designing objects that may serve as descriptions: (1) the program properties that we want to expose, that is, *adequacy* of descriptions; (2) the *precision* that we hope to obtain; and (3) the *efficiency* of the resulting data-flow analysis. Keep in mind that we usually are interested in properties that are undecidable, and so we cannot hope for optimal precision in a strong sense. This is acceptable: A data-flow analysis need not tell the whole truth, although it should, of course, not contradict truth. In this way, approximate computation becomes to standard computation what numerical analysis is to mathematical analysis: Using numerical techniques, problems with no known analytic solution can be "solved" numerically; that is, a solution can be estimated within an interval of error.

It is clear that there is a trade-off between precision and efficiency. The finer-grained descriptions we use, the better data-flow analysis can be performed, but considerations of finite computability and efficiency put a limit on granularity. To obtain termination it is common to use descriptions that form Noetherian lattices. The reason is that if the analysis can be expressed as least fixpoint of a monotonic function on such a lattice then this fixpoint can be computed by means of the function's completed Kleene sequence.

[3] Collatz's problem in number theory (also known as the $3n + 1$ problem) amounts to determining whether this program terminates for all $n \in \mathbf{N}$. This problem is still unsolved.

Note that we are not primarily interested in the *results* of computations. For program analysis purposes, our main concern is to extract information about invariance at particular program points, such as "at this point, variable x is always assigned positive values" or "this term becomes ground during execution."

4. A DENOTATIONAL APPROACH

Let us briefly recapitulate the advantages of abstract interpretation. First, it is *semantics-based*, meaning that it forces data-flow analyses to be defined with direct reference to a formal semantics of the given programming language. Although perhaps somewhat restrictive, this discipline is what facilitates the development of *provably correct* data-flow analyses. Second, it is a unifying theory. Often, many seemingly unrelated data-flow analyses can be expressed as variants of a common form. Abstract interpretation tends to emphasize the essential differences as well as what is common in analyses, thus helping taxonomy.

The last point suggests that the design and implementation of a range of data-flow analyses may often be simplified first by designing a *framework* that is, essentially, a generic algorithm that captures the overall computation regime common to a large class of analyses, and then by providing the characteristic details of a given analysis in the form of various parametric functions. Indeed, in the area of logic programming, such generic frameworks and various implementations, called *analysis engines*, are now emerging.

However, abstract interpretation can be generalized even further so as to provide a theory of data-flow analysis that is *independent of any particular programming language or operational semantics*. We have in mind here Nielson's theory[4] of *denotational abstract interpretation* [Nielson 1982; 1988; 1989].

This section is concerned with explaining and customizing Nielson's theory to better serve our needs. There are two or three revisions that prove useful in the setting of logic programs. These revisions are necessary because Nielson was concerned with deterministic languages, but logic programs are nondeterministic. Most importantly, we include a join operator in the metalanguage, as join is the natural operation for combining sets of possible outcomes from a nondeterministic choice. Another difference is that we do not require continuity of functions. We show that Nielson's results still hold in this revised setting, as long as approximation is formalized in terms of a concretization function, rather than in terms of an abstraction function.

4.1 Language-Independent Abstract Interpretation

Assume we are given a language in which one can express the semantics of a wide variety of programming languages. The theory of abstract interpretation may then be developed in the framework of this *metalanguage* once and for all. In this way, we need not invent (or reinvent) abstract interpretation for

[4] Some of the ideas in Nielson's treatment can be traced back to work by Donzeau-Gouge [1978].

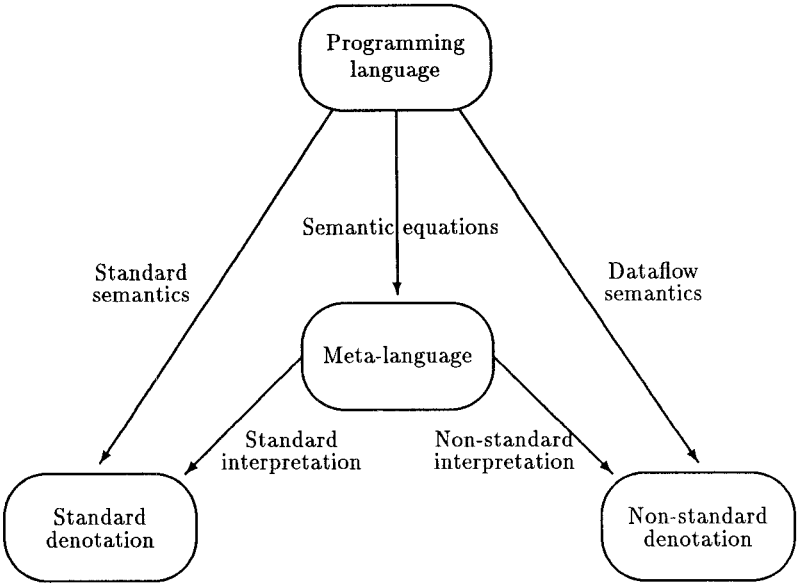

Fig. 2. Role of the metalanguage (after Nielson [1988]).

each new kind of programming language or operational semantics; each is just a special instance of the general theory. With a careful choice of metalanguage, properties that are important for the correctness of data-flow analyses can be established at the level of the metalanguage once and for all. They need not be reproved for individual analyses or even for individual programming languages. Figure 2 illustrates the point, which we push further in Section 4.2.

Expressing standard and nonstandard semantics in the same metalanguage also supports a *derivational* approach to the development of data-flow analyses. The *definition* of a data-flow analysis should be easily derived from that of the standard semantics. Only in a second stage should the analysis be implemented from its definition. Such a stepwise approach may be preferable to the task of proving some baroque data-flow procedure correct with respect to a semantic definition (or an interpreter).

Figure 3 illustrates how the denotational approach allows the separation of concerns. The two upper circles represent formal definitions in the metalanguage. The lower circles represent *implementations* of the semantic definitions. The vertical arrows represent the relation "implements" (or "abstracts" if read the other way). The point is that the relation (marked ? in the figure) between a data-flow procedure and an operational semantics is usually very complicated and that the problem of establishing the former's correctness with respect to the latter is difficult. In our approach it is broken up into two orthogonal issues: that of implementation (of a formally defined data-flow analysis) and that of correct *approximation* of one definition by another.

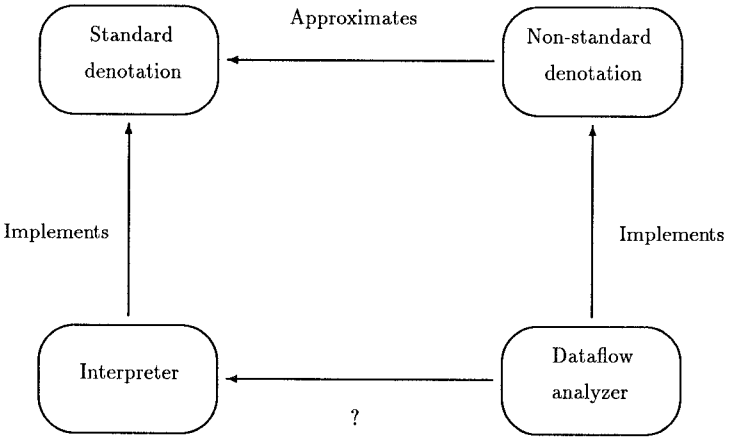

Fig. 3. Separating two orthogonal aspects of the data-flow analyzer.

The notion of approximation is made precise in Section 4.2. Later, when we give the formal semantic definitions (the two upper circles in Figure 3) relevant to logic programs, the reader will realize that they are virtually identical: A few "bricks" may differ, but the structure and mortar are identical. This is very useful, since the approximation relation will be shown to have the property that it will automatically hold at the top level if only it holds at the level of the few selected "bricks."

We stress that, to achieve this desirable situation, the exact choice of metalanguage is important. It is not the case that any language will do. On the one hand, the metalanguage must be sufficiently expressive to be useful, and on the other hand, it must be sufficiently curtailed for the all-important Theorem 4.4 below to hold.

The present treatment is similar to Nielson's in his paper on strictness analysis [Nielson 1988]. There are, however, important differences. We do not formalize abstract interpretation in terms of an "*abstraction* function," but rather in terms of a "concretization function." In the theory of Cousot and Cousot [1977; 1979], this difference would be a matter of taste only, since the two functions uniquely determine each other. Here, however, the difference is important: We do not require the existence of an abstraction function, and Nielson does not require the existence of a concretization function. The reason for our choice is that we want our metalanguage to include a join operator, something that Nielson does not need, as he is concerned with deterministic programming languages only. In Nielson's framework, the addition of a join operation would invalidate the all-important Theorem 4.4 below, as shown in Example 4.1. In addition, we find it simpler to think in terms of concretization functions, since they are in a sense "semantic functions" for descriptions.

Like Nielson we use the formalism of denotational semantics. Although this is not the only choice for a metalanguage, its usefulness and suitability

should be apparent from the following sections: Both standard and nonstandard semantics are easily presented in the metalanguage. By choosing the right level of abstraction, they can be made highly congruent: If the standard semantics employs a certain operator on the standard domain, the nonstandard semantics should use a corresponding operator on the nonstandard domain.

The metalanguage is one of typed lambda expressions, and the language of type expressions is given by

$$T \in \mathit{Type} ::= S \mid L,$$
$$L \in \mathit{Lat} ::= D \mid T \to L,$$

where $S \in \mathit{Stat}$, $D \in \mathit{Dyn}$, *Stat* is a collection of *static* types, and *Dyn* is a collection of *dynamic* types. The difference between the two kinds is that (the interpretation of) a static type remains the same throughout all (standard and nonstandard) semantics, whereas a dynamic type may change. We call $\mathit{Stat} \cup \mathit{Dyn}$ the collection of *base* types, and *Lat* the collection of lattice types. When we give the semantics of these types shortly, it will become clear why S, $T \to S$, $T \to T \to S$, and so on are excluded from the lattice types.

The syntax of the metalanguage is given by

$e ::= c_t$	(base functions)
$\mid x_t$	(variables)
$\mid \lambda x\colon T.e$	(function abstraction)
$\mid e\, e'$	(function application)
$\mid$ if e then e' else e''	(conditional)
$\mid \mathit{lfp}\, e$	(least fixed point)
$\mid e \sqcup e'$	(join).

We are generally concerned with expressions that are *closed*; that is, every variable is bound by a lambda.

There is a notion of well typing for this language, which is explained shortly. Static types are interpreted as posets (ordered by identity), and dynamic types as complete lattices. A *type interpretation* $\mathbf{I}$ thus assigns a structure ($\mathbf{I}T$) to each base type T. By natural extension the semantics of types is defined as follows:

$\mathbf{I}[\![S]\!] = \mathbf{I}S$	(a (fixed) poset, ordered by identity),
$\mathbf{I}[\![D]\!] = \mathbf{I}D$	(some complete lattice),
$\mathbf{I}[\![T \to L]\!] = \mathbf{I}[\![T]\!] \to \mathbf{I}[\![L]\!]$	(ordered pointwise).

Recall that $\to$ on the right-hand side denotes monotonic function space.

By this, $\mathbf{I}[\![L]\!]$ is a complete lattice for every $L \in \mathit{Lat}$, which is why dynamic types are required to denote complete lattices. The semantic equations in the following sections define least fixpoints of monotonic functionals over domains of type $T \to L$, and the type discipline thus automatically guarantees

well definedness of the semantics. We do not require that domains with static types have excessive structure: Many domains that are to remain fixed throughout all (interpretations of) semantic definitions, such as "the set of programs," are best thought of simply as sets.

We now explain the notion of well typedness. The denotation of an expression e is relative to a type environment τ and a type T such that $\tau \vdash e: T$ (i.e., in environment τ, e has type T). The type rules are

$$\tau \vdash c_\iota : T \quad \text{iff} \quad T \text{ is the (fixed) type of } c_\iota, \qquad \tau \vdash x_\iota : T \quad \text{iff} \quad \tau x_\iota = T,$$

$$\frac{\tau[T/x] \vdash e: T'}{\tau \vdash (\lambda x: T.e): T \to T'}, \qquad \frac{\tau \vdash e: T \to T' \quad \tau \vdash e': T}{\tau \vdash (e\,e'): T'},$$

$$\frac{\tau \vdash e: \mathit{Bool} \quad \tau \vdash e': T \quad \tau \vdash e'': T}{\tau \vdash \text{if } e \text{ then } e' \text{ else } e'': T}, \qquad \frac{\tau \vdash e: L \to L}{\tau \vdash \mathit{lfp}\ e: L},$$

$$\frac{\tau \vdash e: L \quad \tau \vdash e': L}{\tau \vdash e \sqcup e': L}.$$

Here $\tau[T/x]$ is the type environment, which is identical to τ except that x is associated with type T. The type *Bool* is that of truth values, and L denotes a lattice type.

We are now ready to explain the semantics of the metalanguage. An *interpretation* $\mathbf{I}$ denotes a type interpretation (by an abuse of notation also called $\mathbf{I}$) together with an assignment of an element of $\mathbf{I}[\![T]\!]$ to each base function c of type T. By natural extension this gives the semantics of the metalanguage. Let the domain of τ be $\{x_1, \ldots, x_k\}$, and let $\tau x_\iota = T_\iota$ for $i \in \{1, \ldots, k\}$. Then $\mathbf{I}[\![e]\!]: \mathbf{I}[\![T_1]\!] \times \cdots \times \mathbf{I}[\![T_k]\!] \to \mathbf{I}[\![T]\!]$ is defined as follows (we let η abbreviate $(v_1, \ldots, v_k)$):

$$\begin{aligned}
\mathbf{I}[\![c_\iota]\!]\eta &= \mathbf{I}c_\iota, \\
\mathbf{I}[\![x_\iota]\!]\eta &= v_\iota, \\
\mathbf{I}[\![\lambda x: T.e]\!]\eta v &= \mathbf{I}[\![e]\!](v_1, \ldots, v_k, v), \\
\mathbf{I}[\![e\,e']\!]\eta &= \mathbf{I}[\![e]\!]\eta(\mathbf{I}[\![e']\!]\eta), \\
\mathbf{I}[\![\text{if } e \text{ then } e' \text{ else } e'']\!]\eta &= \text{if } \mathbf{I}[\![e]\!]\eta \text{ then } \mathbf{I}[\![e']\!]\eta \text{ else } \mathbf{I}[\![e'']\!]\eta, \\
\mathbf{I}[\![\mathit{lfp}\ e]\!]\eta &= \mathit{lfp}(\mathbf{I}[\![e]\!]\eta), \\
\mathbf{I}[\![e \sqcup e']\!]\eta) &= (\mathbf{I}[\![e]\!]\eta) \sqcup (\mathbf{I}[\![e']\!]\eta).
\end{aligned}$$

By varying the interpretation, one may obtain different semantics from the same set of semantic equations. The *standard interpretation* gives the usual input/output behavior of the program, while data-flow analyses may be expressed as "nonstandard" interpretations. The role of abstract interpretation is to give relationships between the standard and nonstandard interpretations that guarantee that the data-flow analysis safely approximates the standard semantics.

4.2 A Theory of Approximation

To relate standard and nonstandard interpretations, one must first explain what the descriptions in the nonstandard semantics mean in terms of the standard semantics. This is done by means of "insertions," where an insertion consists of a function that maps descriptions to the largest object that they describe.

Definition. An *insertion* is a triple (X, γ, Y), where X and Y are complete lattices and where the monotonic function $\gamma: X \to Y$ is costrict. A *concretization family*, $\Gamma = \{\gamma_D\}$, for interpretations $\mathbf{I}'$ and $\mathbf{I}$ is an indexed family of functions such that, for each $D \in Dyn$, $((\mathbf{I}'D), \gamma_D, (\mathbf{I}D))$ is an insertion.

The motivation for the definition of insertion is that the domain X of descriptions should "approximate" Y in the sense that the two have compatible structures, so that γ should be monotonic. Furthermore, every element in Y should have a corresponding description, so that γ should be costrict.

In Cousot and Cousot's theory of abstract interpretation, the function γ is required to have an *adjoined*, so-called *abstraction* function $\alpha: Y \to X$.

Definition. Let X and Y be complete lattices. The (monotonic) functions $\gamma: X \to Y$ and $\alpha: Y \to X$ are *adjoined* iff $\forall x \in X. \forall y \in Y. \alpha y \leq x \Leftrightarrow y \leq \gamma x$.

The abstraction function can be thought of as giving the best description of an object. Cousot and Cousot [1977] demand that an abstraction function exist. This they do on grounds that, otherwise, data-flow analyses may not yield the best possible result, given the set of descriptions chosen; see also Søndergaard [1989].

Like Bruynooghe [1991] and Bruynooghe and Janssens [1992], we do not require the existence of an abstraction function. Note that when it exists, it is uniquely defined by $\alpha y = \sqcap\{x \mid y \leq \gamma x\}$. The following definition allows us to characterize when an abstraction function exists:

Definition. Let X be a lattice, and let $Y \subseteq X$. We say that Y is *Moore-closed* (in X) iff

$$\forall Y' \subseteq Y. \sqcap Y' \in Y.$$

LEMMA 4.1 [COUSOT AND COUSOT 1979]. *Let (X, γ, Y) be an insertion. Then γ has an adjoint iff $\{\gamma x \mid x \in X\}$ is Moore-closed.*

Recalling our convention (Example 3.1) of ordering descriptions according to how large a set of objects they apply to, we now define what it means for a description to approximate another description safely.

Definition. Let $\Gamma = \{\gamma_D\}_{D \in Dyn}$ be a concretization family for interpretations $\mathbf{I}'$ and $\mathbf{I}$, and let $S \in Stat$ and $D \in Dyn$ be types. The relation $appr[\Gamma]_T$

is defined by

$$u\ appr[\Gamma]_S v \Leftrightarrow v = u$$
$$u\ appr[\Gamma]_D v \Leftrightarrow v \leq \gamma_D u$$
$$u\ appr[\Gamma]_{T \to L} v \Leftrightarrow \forall u', v'.u'\ appr[\Gamma]_T v' \Rightarrow (u\,u')appr[\Gamma]_L(v\,v').$$

When the concretization family Γ and type T is clear from the context, we shall write $appr[\Gamma]_T$ simply as $appr$.

LEMMA 4.2. *For all lattice types L and concretization families Γ, $appr[\Gamma]_L$ is inclusive.*

PROOF. The proof is by structural induction on the types. Let $Z \subseteq (\mathbf{I}[\![L]\!]) \times (\mathbf{I}'[\![L]\!])$ be a chain such that $x\ appr[\Gamma]_L\ y$ holds for all $(x, y) \in Z$. Clearly, $\sqcup Z = (\sqcup X, \sqcup Y)$, where $X = \{x \mid (x, y) \in Z\}$ and $Y = \{y \mid (x, y) \in Z\}$.

Consider the case $L \in Dyn$. By monotonicity of γ_L, $y \leq \gamma_L x \leq \gamma_L(\sqcup X)$ for all $(x, y) \in Z$. Thus, $\sqcup Y \leq \gamma_L(\sqcup X)$, and so $appr$ is inclusive for this case.

The other case is when $L = [\![T \to L']\!]$. Let $(x', y') \in (\mathbf{I}[\![T]\!]) \times (\mathbf{I}'[\![T]\!])$ be given. Then $W = \{(x\,x', y\,y') \mid (x, y) \in Z\}$ is a chain, and $\sqcup W = ((\sqcup X)x', (\sqcup Y)y')$. Assume that $x'\ appr[\Gamma]_T y'$ holds. Then $(xx')\ appr[\Gamma]_{L'}\ (yy')$ holds for all $(x, y) \in Z$, by the definition of $appr$. Since $appr[\Gamma]_{L'}$ is inclusive, $(x\,x')\ appr[\Gamma]_{L'}\ (y\,y')$ holds for $(x, y) = \sqcup Z = (\sqcup X, \sqcup Y)$. Since x' and y' were arbitrary, $(\sqcup X)\ appr[\Gamma]_L(\sqcup Y)$ holds, so $appr[\Gamma]_L$ is inclusive. □

Definition. A relation $R \subseteq X \times Y$ is *order-preserving* iff for all $x, x' \in X$ and $y, y' \in Y$, if xRy and $x \leq x'$ and $y' \leq y$, then $x'Ry'$. R is *additive* iff for all $x \in X$ and $y, y' \in Y$, if xRy and xRy', then $xR(y \sqcup y')$.

LEMMA 4.3. *For all lattice types L and concretization families Γ, $appr[\Gamma]_L$ is order-preserving and additive.*

PROOF. The proof is by straightforward structural induction on the types. □

These results lead to the following theorem, which extends an important result by Nielson [1988]. It says that, if we are given base functions $c_1, \ldots, c_n$ and $c'_1, \ldots, c'_n$ such that c'_i approximates c_i for every $i \in \{1, \ldots, n\}$, then the closed expression $e[c'_1, \ldots, c'_n]$ approximates $e[c_1, \ldots, c_n]$.

THEOREM 4.4. *Let Γ be a concretization family for interpretations $\mathbf{I}$ and $\mathbf{I}'$. If $(\mathbf{I}c)\ appr\ (\mathbf{I}'c)$ holds for every base function c, then, for any closed expression e, $(\mathbf{I}e)\ appr\ (\mathbf{I}'e)$.*

PROOF. The proof is by structural induction on the form of e. Most cases are straightforward, and thus, we show only two: $e = [\![lfp\ e']\!]$ because fixpoint induction is needed for that case; and $e = [\![e_1 \sqcup e_2]\!]$, since this is a construct that is not in Nielson's language.

Consider the case $e = [\![lfp\ e']\!]$. Note that e' must be a closed expression. Furthermore, the type discipline guarantees that $\mathbf{I}[\![e']\!] \in \mathbf{I}[\![L]\!] \to \mathbf{I}[\![L]\!]$ for some $L \in Lat$, while $\mathbf{I}'[\![e']\!] \in \mathbf{I}'[\![L]\!] \to \mathbf{I}'[\![L]\!]$.

Let g: $((\mathbf{I}[\![L]\!]) \times (\mathbf{I}'[\![L]\!])) \to ((\mathbf{I}[\![L]\!]) \times (\mathbf{I}'[\![L]\!]))$ be defined by $g(x, x') = (\mathbf{I}[\![e']\!]x, \mathbf{I}'[\![e']\!]x)$, and assume that $x\ appr\ x'$ holds. By the induction hypothesis, $(\mathbf{I}[\![e']\!]x)\ appr\ (\mathbf{I}'[\![e']\!]x)$. We can interpret this as a statement $Q(x, x') \Rightarrow Q(g(x, x'))$, where Q is inclusive by Lemma 4.2, and so $Q(lfp\ g)$ holds; that is, $(lfp(\mathbf{I}[\![e']\!]))\ appr\ (lfp\ (\mathbf{I}'[\![e']\!]))$. Thus, $(\mathbf{I}[\![lfp\ e']\!])\ appr\ (\mathbf{I}'[\![lfp\ e']\!])$, as desired.

Consider the case $e = [\![e_1 \sqcup e_2]\!]$. Note that e_1 and e_2 must be closed. The type discipline guarantees that $(\mathbf{I}[\![e_1]\!]), (\mathbf{I}[\![e_2]\!]) \in (\mathbf{I}[\![L]\!])$ for some lattice type L, while $(\mathbf{I}'[\![e_1]\!]), (\mathbf{I}'[\![e_2]\!]) \in (\mathbf{I}'[\![L]\!])$. Thus, $\mathbf{I}[\![e_1 \sqcup e_2]\!]$ and $\mathbf{I}'[\![e_1 \sqcup e_2]\!]$ are well defined.

By the induction hypothesis,

$$\big(\mathbf{I}[\![e_1]\!]\big)\, appr[\Gamma]_L \big(\mathbf{I}'[\![e_1]\!]\big) \qquad \text{and} \qquad \big(\mathbf{I}[\![e_2]\!]\big)\, appr[\Gamma]_L \big(\mathbf{I}'[\![e_2]\!]\big).$$

From Lemma 4.3, $appr[\Gamma]_L$ is order-preserving, so

$$\big(\mathbf{I}[\![e_1 \sqcup e_2]\!]\big)\, appr[\Gamma]_L \big(\mathbf{I}'[\![e_1]\!]\big) \qquad \text{and} \qquad \big(\mathbf{I}[\![e_1 \sqcup e_2]\!]\big)\, appr[\Gamma]_L \big(\mathbf{I}'[\![e_2]\!]\big).$$

Thus, by additivity of $appr[\Gamma]_L$, $(\mathbf{I}[\![e_1 \sqcup e_2]\!])\ appr[\Gamma]_L\ (\mathbf{I}'[\![e_1 \sqcup e_2]\!])$. □

Note the generality of this result: It immediately allows us to argue inductively the correctness of a whole data-flow analysis (i.e., nonstandard semantics) once certain primitive base functions are shown to be in the relation "safely approximates." This applies not only to logic-programming languages as discussed in this paper, but to any language whose semantics can be expressed in the metalanguage. (In fact, the metalanguage can be further extended in various natural directions [Nielson 1988].)

Example 4.1. In Nielson's theory of denotational abstract interpretation, the approximation relation is defined in terms of an "abstraction function" α: $Y \to X$, which maps a data object $y \in Y$ to its best approximation $x \in X$. Thus, x approximates y iff $\alpha y \leq x$.

In this setting Theorem 4.4 does not hold, because of the join operator. To see this consider the lattices in Figure 4. Clearly, $a \sqcup_Y b = \top$, and $a \sqcup_X b = c$. With the abstraction function α: $Y \to X$ defined by $\alpha y = y$, we have that a approximates a and that b approximates b, but c does not approximate $\top$.

Perhaps surprisingly, the relation $appr[\Gamma]_T$ is, in general, neither reflexive nor transitive. For counterexamples, see Nielson's [1989] Example 5.1.3 for lack of reflexivity and Marriott's [1993] Example 5.1 for lack of transitivity. Reflexivity and transitivity would seem to be essential requirements to *appr* if the relation is to be used as an ordering on analyses. This motivates the following definition:

Definition. A lattice type $T \in Type$ is *first-order* iff it can be generated by the grammar

$$T ::= D \mid D \to T \mid S \to T,$$

where $S \in Stat$ and $D \in Dyn$.

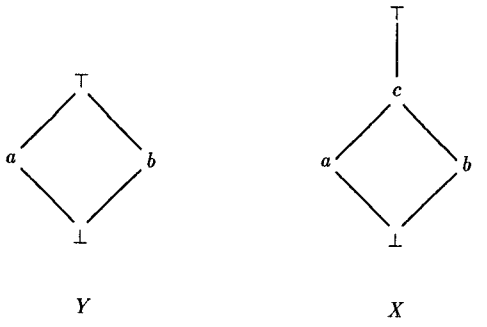

Fig. 4. Example lattices.

For example, $(S \to D) \to D$ is not first-order.

The following proposition says that approximation is transitive for first-order types.

PROPOSITION 4.5. *Let* $\mathbf{I}$, $\mathbf{I}'$, *and* $\mathbf{I}''$ *be interpretations and let e be a closed expression of first-order lattice type. Then* $(\mathbf{I}e)$ $appr[\Gamma']$ $(\mathbf{I}'e)$ *and* $(\mathbf{I}'e)$ $appr[\Gamma''](\mathbf{I}''e)$ *implies* $(\mathbf{I}e)$ $appr[\Gamma'' \circ \Gamma'](\mathbf{I}''e)$, *where* $\Gamma'' \circ \Gamma'$ *stands for the concretization family* $\{\gamma''_D \circ \gamma'_D \mid D \in Dyn, \gamma'_D \in \Gamma', \gamma''_D \in \Gamma''\}$.

PROOF. The proof is by induction on types. In the case of D or $S \to T$, it is straightforward. So assume that the assertion holds for types D and T, and we will show that it holds for types $D \to T$. Assume that v $appr[\Gamma'' \circ \Gamma']_D$ v''. Then v $appr[\Gamma']_D$ v' and v' $appr[\Gamma'']_D$ v'', where $v' = \gamma'_D v$. Assume that u $appr[\Gamma']_{D \to T}$ u' and u' $appr[\Gamma'']_{D \to T}$ u''. Then (uv) $appr[\Gamma']_T$ $(u'v')$ and $(u'v')$ $appr[\Gamma'']_T$ $(u''v'')$. By the induction hypothesis, $(u\,v)$ $appr[\Gamma'' \circ \Gamma']_T$ $(u''v'')$. It follows that u $appr[\Gamma'' \circ \Gamma']_{D \to T}$ u''. □

This result is useful because it allows the stepwise development and proof of approximations. At first sight, the restriction to first-order types looks serious, since the proposition does not seem to offer any help if we are working with continuation-based denotational definitions. However, one has to remember that the proposition will only be applied to the semantic functions *for a program as a whole* and that the restriction therefore has little impact.

Approximation is also reflexive for first-order types, as there is a natural "identity" approximation.

Definition. Let $\mathbf{I}$ be an interpretation. The *identity* concretization family for $\mathbf{I}$ is the concretization family $\Gamma_{Id} = \{\gamma_D\}$, where, for each $D \in Dyn$, $\gamma_D x = x$.

PROPOSITION 4.6. *Let* $\mathbf{I}$ *be an interpretation with an identity concretization family* Γ_{Id}, *and let e be a closed expression of first-order lattice type. Then* $(\mathbf{I}e)$ $appr[\Gamma_{Id}]$ $(\mathbf{I}e)$.

PROOF. We actually prove that for all first-order lattice types, L, u $appr[\Gamma_{Id}]_L$ v iff $v \leq u$. The proof is by induction on types.

The base case when $L \in Dyn$ follows from the definition of Γ_{Id}. Consider the case when L is $D \to L'$. Now $u\ appr[\Gamma_{Id}]_{D \to L'}\ v$ iff, for all u' and v', $u'\ appr[\Gamma_{Id}]_D\ v' \Rightarrow (u\,u')\ appr[\Gamma_{Id}]_{L'}\ (v\,v')$. By the induction hypothesis, $u\ appr[\Gamma_{Id}]_{D \to L'}\ v$ iff, for all u' and v', $v' \leq u' \Rightarrow (v\,v') \leq (u\,u')$, which by monotonicity of u and v holds iff $v \leq u$. The case when L is $S \to L'$ is similar. □

In this section we have introduced a general theory of abstract interpretation that is based on a simple metalanguage and formalized in terms of concretization functions. The theory carefully separates the two concerns: approximation and implementation. The metalanguage is a variant of the typed lambda-calculus and has been carefully chosen so that correctness of approximation lifts from basic expressions to all expressions in the language (Theorem 4.4). This facilitates proofs of correctness.

In general, approximation in this theory is not transitive or reflexive, but we have given a simple condition that ensures that approximation is transitive and reflexive (Propositions 4.5 and 4.6). This allows us to develop data-flow analyses in a stepwise, derivational manner. In the remainder of the paper, we illustrate an application of the theory.

5. SLD RESOLUTION: A CONSTRAINT LOGIC PROGRAMMING VIEW

Interpreters and compilers for logic-programming languages are usually based on SLD resolution. Therefore, it is reasonable to base the abstract interpretation of such languages on some formalization of SLD resolution. We demonstrate the utility of the language-independent theory of abstract interpretation developed in the last section by using it to give a generic analysis framework for logic programs that captures the SLD semantics.

In this paper we are only concerned with definite programs [Lloyd 1987]. A *definite program*, or *program*, is a finite set of clauses. A *clause* is of the form $H \leftarrow B$, where H, the *head*, is an *atom* and B, the *body*, is a finite sequence of atoms. We let *Var* denote the (countably infinite) set of variables, *Term* the set of terms, *Pred* the set of predicate symbols, *Atom* the set of atoms, *Clause* the set of clauses, and *Prog* the set of (definite) programs.

We assume that we are given a function *vars*: $(Prog \cup Atom \cup Term) \to \mathcal{P}\ Var$, such that $(vars\ S)$ is the set of variables that occur in the syntactic object S. A syntactic object S is *ground* iff it is constructed without variables; that is, $vars\ S = \emptyset$.

A *substitution* is an almost-identity mapping $\theta \in Sub \subseteq Var \to Term$. Substitutions are not distinguished from their natural extensions to other syntactic categories. Our notation for substitutions is fairly standard. For instance, $\{x \mapsto a\}$ denotes the substitution θ such that $(\theta x) = a$ and $(\theta V) = V$ for all variables $V \neq x$. The *restriction* of a substitution θ to a set of variables W, written $\theta|_W$, is the substitution θ' defined by $\theta' V$ is θV if $V \in W$; otherwise, it is V. The *domain* of a substitution θ, written *dom* θ, is the set of variables $\{V \in Var \mid \theta V \neq V\}$.

A *unifier* of A, $H \in Atom$ is a substitution θ such that $(\theta A) = (\theta H)$. If such a unifier exists, then A and H are *unifiable*. A unifier θ of A and H is

an (idempotent) *most general unifier* of A and H iff $\theta' = \theta' \circ \theta$ for every unifier θ' of A and H. Two atoms A and H have a most general unifier whenever they are unifiable. The auxiliary function *mgu*: $Atom \to Atom \to \mathcal{P}\, Sub$ is defined as follows: If A and H are unifiable, then $(mgu\ A\ H)$ yields a singleton set consisting of a most general unifier of A and H. Otherwise, $(mgu\ A\ H) = \emptyset$.

Let $G = \leftarrow A_1, \ldots, A_n$ be a *query* with *selected atom* A_i, and let $C = H \leftarrow A'_1, \ldots, A'_k$ be a clause. If A_i and H are unifiable, then

$$\theta[\![A_1, \ldots, A_{i-1}, A'_1, \ldots, A'_k, A_{i+1}, \ldots, A_n]\!]$$

is a *resolvent* of G and C with unifier θ, where $\{\theta\} = mgu\ A_i H$.

Let P be a definite program, and let G be a query. An *SLD derivation* of $P \cup \{G\}$ consists of

—a maximal sequence $G_0, G_1, \ldots$ of queries with $G_0 = G$,
—a sequence $C_0, C_1, \ldots$ of fresh variants of clauses from P (i.e., the variables in C_i are consistently replaced by variables not in $C_0, \ldots, C_{i-1}$, or G), and
—a sequence $\theta_0, \theta_1 \ldots$ of substitutions,

such that, for all i, G_{i+1} is a resolvent of G_i and C_i with unifier θ_i. An SLD derivation may be finite or infinite. Assume it is finite, with final elements G_{n+1}, C_n, and θ_n. Then, if G_{n+1} is empty, the derivation is *successful*; otherwise, it is *finitely failed*. If it is successful, the *computed answer* is $\theta_n \circ \cdots \circ \theta_0$, restricted to the set of variables occurring in G.

We will, in fact, not need substitutions and unification as defined above until Section 7. Perhaps surprisingly, they play no role in the formal semantics we are about to present. Inspired by constraint logic languages, we let the role of substitutions be played by term equations. There are several reasons for this, but most importantly, we have the following:

(1) It is simpler to think of term equations ordered by logical consequence than substitutions ordered by some complicated "instantiation" ordering. It also provides a pleasing uniformity in the present context, since in Section 7, we "approximate" term equations by propositional formulas (or Boolean functions), again ordered by logical implication.
(2) It does away with the awkward directionality of substitutions. Semantically, the distinction between $\{x \mapsto y\}$ and $\{y \mapsto x\}$ is unnecessary and hampers full abstraction.
(3) Substitutions do not form a complete lattice, and it is common to restrict attention to *idempotent* substitutions. That approach seems unnatural, in particular, since the class of idempotent substitutions is not closed under composition; witness $\{x \mapsto f(y)\} \circ \{y \mapsto x\}$.

The other characteristic of our definition is that unification is not only performed when a clause is entered, but also when it is exited. This is to enforce a regime in which "only local variables matter." It reflects the fact that a compiler, optimizing code for a given clause, will mainly need data-flow

information expressed in terms of variables local to the clause. Finally, our definition assumes the standard (left-to-right) computation rule.

5.1 Existentially Quantified Term Equations

We now describe the most important semantic domain involved in the definitions to come.

Definition. An *ex-equation* is a possibly existentially quantified conjunction of basic equations $T_1 = T_2$. The conjunction may be empty, in which case we denote it by *true*. We call the set of ex-equations *Eqn*.

We consider ex-equations in the context of an alphabet given by some program/query. The semantics of a basic ex-equation is given by the term algebra over that alphabet. Ordering the elements by logical consequence and considering equivalence classes in the usual way, we obtain a partially ordered set with *true* as the greatest element and with *false*, the equivalence class of unsatisfiable elements, as the smallest.

Example 5.1. The ex-equation $\exists y.x = f(y)$ corresponds to the substitution $\{x \mapsto f(y)\}$, and so does the ex-equation $x = f(y)$. The difference between the two is that the latter specifies a relation between two program variables, x and y, whereas the former merely states that x is constrained to take certain forms. Said differently, the *name* y is significant in the latter, but not in the former. We thus have two different kinds of placeholders in terms.

We can extend the definition of a "unifier" to ex-equations in a natural way. This provides the link between our semantic definition and the more usual definition using substitutions.

Definition. A *unifier* of ex-equation e is a substitution θ such that $\models (\theta e)$. We let *unif* e denote the set of unifiers of e.

Thus, θ is a unifier of the atoms A and H iff it is a unifier of the ex-equation $A = H$. Note that application of a substitution to an ex-equation requires that the existentially quantified variables be renamed away from the variables in the substitution, so as to avoid name clashes.

Example 5.2. The ex-equation $\exists y.x = f(y)$ has unifiers $\{x \mapsto f(z)\}$, $\{x \mapsto f(a)\}$, and so on. Unifiers for $x = f(y)$ include $\{x \mapsto f(y)\}$, $\{x \mapsto f(a),\ y \mapsto a\}$, and $\{x \mapsto f(z),\ y \mapsto z\}$, but not $\{x \mapsto f(z)\}$ or $\{x \mapsto f(a)\}$. The ex-equation *false* has no unifier.

5.2 The Base Semantics

We now make use of ex-equations to give a semantic definition that is suitable as a basis for abstract interpretation and that captures SLD resolu-

[5] *Corresponds* should be taken loosely: *Comb*, unlike composition, is commutative; and *unify* is noncommutative. The commutativity of *comb* restores the intuition of SLD resolution as constraint manipulation, an intuition that is lost with substitutions and their composition If e and e' are constraints, then *composing* them simply means forming their conjunction. The reason why *unify* is noncommutative will become clear shortly.

tion. The definition makes use of the auxiliary functions, *comb*, which corresponds to composition, and *unify*, which corresponds to computing the most general unifier.[5]

We shall need a particular kind of variable renaming: A *name toggle* is an involution; that is, a bijection $\rho \in Ren \subset Var \to Var$, which satisfies $\rho = \rho^{-1}$. We do not distinguish a name toggle from its natural extension to atoms and clauses. The function ren: $\mathscr{P}\ Var \to \mathscr{P}\ Var \to Ren$ generates name toggles: $(ren\ U\ W)$ is some name toggle, such that, for all V in W, $ren\ U\ W\ V \notin U$. Note that this is renaming of variables in W "away from" U.

Definition. The function $comb$: $\mathscr{P}\ Eqn \to \mathscr{P}\ Eqn \to \mathscr{P}\ Eqn$ is defined by

$$comb\ E\ E' = \{e \wedge e' \mid e \in E \wedge e' \in E'\}.$$

The function $restrict$: $\mathscr{P}\ Var \to \mathscr{P}\ Eqn \to \mathscr{P}\ Eqn$ is defined by

$$restrict\ U\ E = \{\exists \overline{U}.e \mid e \in E\},$$

where $\overline{U}$ denotes $(vars\ E) \setminus U$. The function $unify$: $Atom \to Atom \to \mathscr{P}\ Eqn \to \mathscr{P}\ Eqn$ is defined by

$$unify\ H\ A\ E = restrict(vars\ H)(comb\{H = \rho A\}(\rho E)),$$

where $\rho = ren(vars\ H)((vars\ A) \cup (vars\ E))$.

Notice that the *calling* atom A is renamed away from the clause head H. The auxiliary functions are defined to operate on *sets* E of conjunctions, rather than on single conjunctions, because in the next section it proves useful to have the broader definitions.

Example 5.3. Let e be $\exists u.(y = u \wedge z = f(u))$, and let e' be $y = a$. Then $comb\{e\}\{e'\}$ is $\{y = a \wedge z = f(a)\}$.

Let $A_1 = p(a, x)$, $A_2 = p(y, z)$, and let e be $\exists u.(y = u \wedge z = f(u))$. Then we have that $(unify\ A_1 A_2\{e\}) = \{x = f(a)\}$. Notice that this ex-equation does not constrain y or z; only variables in A_1 may be constrained. On the other hand, we have that $(unify\ A_2 A_1\{e\}) = \{y = a\}$, in which both x and z are unconstrained: With name toggle $\{y \mapsto y', z \mapsto z', y' \mapsto y, z' \mapsto z\}$, we get

$$\begin{aligned} unify\ A_2 A_1\{e\} &= \{\exists y', z'.(y = a \wedge z = x \wedge \exists u.(y' = u \wedge z' = f(u)))\} \\ &= \{y = a\}. \end{aligned}$$

Notice also that $(unify\ A_1 A_1\{e\}) = \{true\}$, while $(unify\ A_2 A_2\{e\}) = \{e\}$.

Let $A_3 = p(f(y), z)$. Then $(unify\ A_3 A_2\{true\}) = \{true\}$. There is no "occur check problem" since variables in A_2 are renamed.

Finally, let $A_4 = p(x, x)$. Then $(unify\ A_4 A_2\{e\}) = \emptyset$, corresponding to failure of unification.

To summarize, $unify\ H A\{e\}$ is the result of conjoining $H = A$ with e and then restricting the result to variables in H. As a first step of unification, variables in A and e are consistently renamed so as to avoid collisions with variables in H. Unification can be done with respect to a set E, and $unify\ H A E$ is then $\{unify\ H A\{e\} \mid e \in E\}$. Notice how the restriction to variables in H means that $unify$ is not symmetric with respect to H and A.

The following definition captures the essence of an SLD refutation-based interpreter using a standard computation rule and a parallel search rule. The domain *Atom* is ordered by identity; *Den* is ordered pointwise:

Definition. The *base semantics* has the semantic domain

$$Den = Atom \to \mathscr{P}\ Eqn \to \mathscr{P}\ Eqn$$

and semantic functions

$$\begin{aligned}\mathbf{P_{bas}}&: Prog \to Den,\\ \mathbf{C_{bas}}&: Clause \to Den \to Den,\\ \mathbf{B_{bas}}&: Body \to Den \to (\mathscr{P}\ Eqn) \to (\mathscr{P}\ Eqn).\end{aligned}$$

It is defined as follows.[6]

$$\begin{aligned}\mathbf{P_{bas}}[\![P]\!] &= lfp(\sqcup_{C \in P}(\mathbf{C_{bas}}[\![C]\!])),\\ \mathbf{C_{bas}}[\![H \leftarrow B]\!]\, d\, A\, E &= \cup_{e \in E}\ comb\,\{e\}\,(unify\, A\, H(\mathbf{B_{bas}}[\![B]\!]\, d\,(unify\, H\, A\,\{e\}))),\\ \mathbf{B_{bas}}[\![nil]\!]\, d\, E &= E,\\ \mathbf{B_{bas}}[\![A : B]\!]\, d\, E &= \mathbf{B_{bas}}[\![B]\!]\, d\,(d\, A\, e).\end{aligned}$$

The statements we are interested in generating from a data-flow analysis are of the form "whenever execution reaches this point, so-and-so holds." In saying so, we do not actually say that execution does reach the point. In particular, "whenever the computation terminates, so-and-so holds" concludes nothing about termination. As nontermination is not distinguished from finite failure, we can assume a *parallel* search rule, rather than the customary depth-first rule, thereby simplifying the semantic equations. Owing to the use of a parallel search rule, the execution of a program naturally yields a *set* of answers, as opposed to a sequence.

Example 5.4. Consider the following list concatenation program P:

$append(nil, y, y).$
$append(u : x, y, u : z) \leftarrow append(x, y, z).$

Let $A = append(x, y, a : nil)$. Execution of P yields two instantiations of the variables in A. We have that $\mathbf{B_{bas}}[\![P]\!]A\,\{true\} = \{x = nil \wedge y = a : nil,\ x = a : nil \wedge y = nil\}$.

A note about semantical modeling and data-flow analysis may be appropriate at this point. We have defined $\mathbf{B_{bas}}$ such that given a program P and a query A it returns the set of computed answer constraints. That is, $\mathbf{B_{bas}}$

[6] Strictly speaking, the singleton set constructor $\{\cdot\}$, as used in the definition, is not part of our metalanguage and is commonly avoided in denotational definitions as it is not monotonic. Its use here causes no problems: The semantic functions are well defined. Subsequent definitions will avoid using $\{\cdot\}$ so as to utilize Theorem 4.4. Finally, $\sqcup_{C \in P}$ is just an abbreviation for the repeated use of the metalanguage's $\sqcup$.

selects what is considered the relevant information from the SLD tree for $P \cup \{\leftarrow A\}$, namely, for each success leaf, the conjunction of the constraints that decorate the path from the success leaf to the root. Other information is "forgotten." For example, $(\mathbf{B}_{\mathbf{bas}}[\![P]\!]A)$ contains no information about the length of derivations, the constraints that decorate paths leading to failure nodes, or how variables outside A become constrained during execution.

This is quite in accordance with the common understanding of SLD semantics as the *result* of applying SLD resolution. However, data-flow analysis aims at extracting information about a program's *execution*, that is, about some of the very details that our "SLD semantics" forgets. For example, a data-flow analysis that aims at determining which calls may appear at run time cannot afford to disregard those paths in the SLD tree that lead to failure. For this reason we need a notion of "extended semantics," in which the calls to a clause (restricted to the local variables in the clause) are collected. It is straightforward to modify the base semantics so that it collects this information. To limit the number of semantic definitions, however, we shall not do this and will ignore this issue for the remainder of this paper. Readers should keep in mind, however, that ultimately the touchstone for the correctness of a data-flow analysis returning information about calls to a clause is its soundness with respect to this extended semantics.

At this stage we have given a very simple denotational definition of logic programs that captures the SLD operational semantics. Its simplicity is due to the use of existentially quantified term equations, rather than substitutions, and of a parallel, rather than sequential, search rule.

6. DATA-FLOW ANALYSIS OF LOGIC PROGRAMS

The semantic definitions given in the previous section have been designed to capture the essence of SLD resolution. In this section we develop a generic data-flow analysis framework from these definitions by factoring out certain operations.

First we introduce some imprecision in the semantics. So far, for all ex-equations generated by a clause, track was kept of the particular ex-equation the clause was called with, so that the meet (conjunction) of generated equations and corresponding call equations could be computed. We now abandon this approach in order to get closer to a data-flow semantics. The point is that, in a data-flow semantics, a "description" x will replace E, the set of "current" ex-equations, and since descriptions are generally atomic (in the sense of nondecomposable), it makes little sense to refer to *elements* of x. As a first step, we thus replace "$\bigcup_{e \in E} \cdots e \cdots$" in the definition of the base semantics by "$\cdots E \cdots$."

Definition. The *lax semantics* has semantic functions

$$\mathbf{P}_{\mathbf{lax}}: \mathit{Prog} \to \mathit{Den},$$
$$\mathbf{C}_{\mathbf{lax}}: \mathit{Clause} \to \mathit{Den} \to \mathit{Den},$$
$$\mathbf{B}_{\mathbf{lax}}: \mathit{Body} \to \mathit{Den} \to (\mathscr{P}\ \mathit{Eqn}) \to (\mathscr{P}\ \mathit{Eqn}).$$

It is defined as follows:

$$\begin{aligned}
\mathbf{P}_{\mathbf{lax}}[\![P]\!] &= lfp(\sqcup_{C \in P}(\mathbf{C}_{\mathbf{lax}}[\![C]\!])),\\
\mathbf{C}_{\mathbf{lax}}[\![H \leftarrow B]\!]\, d\, A\, E &= comb\, E(unify\, A\, H\, (\mathbf{B}_{\mathbf{lax}}[\![B]\!]\, d(unify\, H\, A\, E))),\\
\mathbf{B}_{\mathbf{lax}}[\![nil]\!]\, d\, E &= E,\\
\mathbf{B}_{\mathbf{lax}}[\![A : B]\!]\, d\, E &= \mathbf{B}_{\mathbf{lax}}[\![B]\!]\, d(d\, A\, E).
\end{aligned}$$

This semantics is only an approximation to $\mathbf{P}_{\mathbf{bas}}$, as the following example shows. However, the lack of precision introduced with $\mathbf{P}_{\mathbf{lax}}$ is acceptable for the purpose of data-flow analysis: It has no impact on applications of abstract interpretation to programs.

Example 6.1. Consider the program P:

$q(x, y, z) \leftarrow p(x, y), r(x, y, z).$
$p(a, v).$
$p(u, a).$
$r(u, v, v).$

and the query $A = \leftarrow q(x, y, z)$. We have that

$$\mathbf{P}_{\mathbf{bas}}[\![P]\!]A\,\{true\} = \{false, x = a \wedge y = z, y = a \wedge z = a\}.$$

However,

$$\mathbf{P}_{\mathbf{lax}}[\![P]\!]A\,\{true\} = \{false, x = a \wedge y = z, y = a \wedge z = a, x = a \wedge y = a \wedge z = a\}.$$

PROPOSITION 6.1. $\mathbf{P}_{\mathbf{lax}}$ *appr* $\mathbf{P}_{\mathbf{bas}}$.

PROOF. It is not hard to show that $\mathbf{C}_{\mathbf{lax}}$ *appr* $\mathbf{C}_{\mathbf{bas}}$. The assertion follows by Theorem 4.4. □

To extract run-time properties of pure Prolog programs, one can develop a variety of nonstandard interpretations of the preceding semantics. To clarify the presentation, we extract from the lax semantics a *data-flow* semantics that contains exactly those features that are common to all of the nonstandard interpretations that we want. It leaves the interpretation of one domain X and two base functions, c and u, unspecified. These missing details of the data-flow semantics are to be filled in by interpretations. In the standard (lax) interpretation of this semantics, $\mathbf{I}_{\mathbf{lax}}$, X is $\mathscr{P}$ *Eqn*, c is *comb*, and u is *unify*. In a nonstandard interpretation, X is assigned whatever set of "descriptions" we choose for approximating sets of ex-equations. So X should be a complete lattice that corresponds to $\mathscr{P}$ *Eqn* in the standard semantics in a way laid down by some insertion $(X, \gamma, (\mathscr{P}\ Eqn))$. The interpretation of c and u should approximate *comb* and *unify*, respectively. *Prog*, *Clause*, and *Atom* are static and have the obvious fixed interpretation. They are ordered by identity. As usual, *Den* is ordered pointwise.

Definition. The *data-flow semantics* has the domain

$$Den = Atom \to X \to X;$$

has semantic functions

$$\begin{aligned}&\mathbf{P}: Prog \to Den,\\ &\mathbf{C}: Clause \to Den \to Den,\\ &\mathbf{B}: Body \to Den \to X \to X;\end{aligned}$$

and has base functions

$$\begin{aligned}&c: X \to X \to X,\\ &u: Atom \to Atom \to X \to X.\end{aligned}$$

It is defined as follows:

$$\begin{aligned}\mathbf{P}[\![P]\!] &= lfp(\sqcup_{C \in P}(\mathbf{C}[\![C]\!])),\\ \mathbf{C}[\![H \leftarrow B]\!]\, d\, A\, x &= c\, x\, (u\, A\, H(\mathbf{B}[\![B]\!]\, d(u\, H\, A\, x))),\\ \mathbf{B}[\![nil]\!]\, d\, x &= x,\\ \mathbf{B}[\![A : B]\!]\, d\, x &= \mathbf{B}[\![B]\!]\, d\, (d\, A\, x).\end{aligned}$$

An interpretation $\mathbf{I}_\mathbf{x}$ for the data-flow semantics is determined by the triple $((\mathbf{I}_\mathbf{x} X), (\mathbf{I}_\mathbf{x} c), (\mathbf{I}_\mathbf{x} u))$. We use the convention that the semantic function $\mathbf{P}$ as determined by an interpretation $\mathbf{I}_\mathbf{x}$ is denoted by $\mathbf{P}_\mathbf{x}$. For example, the standard interpretation $\mathbf{I}_{\mathbf{lax}}$ is $((\mathscr{P} Eqn), comb, unify)$, and the corresponding semantics is denoted by $\mathbf{P}_{\mathbf{lax}}$.

Since $\mathbf{C}[\![C]\!]$ is monotonic for every clause C and every interpretation, we have the following proposition:

PROPOSITION 6.2. *For every interpretation* $\mathbf{I}_\mathbf{x}$, $\mathbf{P}_\mathbf{x}$ *is well defined.*

Definition. Let $\mathbf{I} = (X, c, u)$, and let $\mathbf{I}' = (X', c', u')$ be interpretations. Then $\mathbf{I}'$ is *sound* with respect to $\mathbf{I}$ iff for some insertion (X', γ, X), c' *appr* c and u' *appr* u.

The next result follows immediately from Theorem 4.4:

COROLLARY 6.3. *If interpretation* $\mathbf{I}_\mathbf{x}$ *is sound with respect to* $\mathbf{I}_\mathbf{y}$, *then* $\mathbf{P}_\mathbf{x}$ *appr* $\mathbf{P}_\mathbf{y}$.

By Propositions 4.5 and 6.1, we therefore have the following theorem:

THEOREM 6.4. *If interpretation* $\mathbf{I}_\mathbf{x}$ *is sound with respect to* $\mathbf{I}_{\mathbf{lax}}$, *then* $\mathbf{P}_\mathbf{x}$ *appr* $\mathbf{P}_{\mathbf{bas}}$.

Developing a data-flow analysis in this framework is therefore a matter of choosing the description domain so that it captures the information required from the analysis and then defining suitable functions to approximate *comb*

and *unify*. Before giving example data-flow analyses, we identify two classes of description domain and indicate how *comb* can be approximated for these classes. This is useful because the descriptions used in most data-flow analyses belong to one of the two classes or are an orthogonal mixture of such descriptions. Thus, finding generic ways to approximate *comb* for these classes will help to give some insight into the design of practical data-flow analyses. We note that, once a suitable approximation for *comb* has been found, then, by the definition of *unify*, it may be used as the basis for developing an approximation to *unify*.

Definition. An insertion $(X, \gamma, (\mathscr{P}\ Eqn))$ is *downward closed* iff, for all $x \in X$ and $e, e' \in Eqn$, $(e \in \gamma x \wedge e' \vDash e)$ implies that $e' \in \gamma x$. An insertion $(X, \gamma, (\mathscr{P}\ Eqn))$ is *upward closed* iff, for all $x \in X$ and $e, e' \in Eqn$, $(e \in (\gamma x) \setminus \{false\} \wedge e \vDash e')$ implies that $e' \in \gamma x$.

Examples of downward closed insertions are those typically used in groundness analysis, type analysis, definite aliasing, and definite sharing analysis. Examples of upward closed insertions are those typically used in possible sharing (independence) analysis, possible aliasing analysis, and freeness analyses. Many complex analyses, such as for the determination of mode statements or the detection of and-parallelism, can be expressed as a combination of simpler analyses based on insertions that are downward or upward closed.

A notion of "substitution closure," which is similar to, but more restrictive than, downward closure, is identified by Debray [1988] as being the basis for an important class of data-flow analyses. Substitution closure is motivated by considerations of efficiency of data-flow analyses and effectively limits the precision of analyses by barring the propagation of "aliasing" or "sharing" information in analyses.

PROPOSITION 6.5. *Let* $(X, \gamma, (\mathscr{P}\ Eqn))$ *be a downward closed insertion, and let* $comb'$: $X \to X \to X$ *be given. Then* $comb'$ *appr comb iff*

$$\forall x, x' \in X.(\gamma x) \cap (\gamma x') \subseteq \gamma(comb'\ x x').$$

PROOF. We have that

$$\begin{aligned}
&comb' \text{ appr } comb \\
&\quad \Leftrightarrow \forall x, x' \in X.\forall E, E' \subseteq Eqn. \\
&E \subseteq (\gamma x) \wedge E' \subseteq (\gamma x') \Rightarrow (comb\ EE') \subseteq \gamma(comb'\ x x') \\
&\quad \Leftrightarrow \forall x, x' \in X.\forall e \in (\gamma x).\forall e' \in (\gamma x').(e \sqcap e') \in \gamma(comb'\ x x') \\
&\quad \Leftrightarrow \forall x, x' \in X.(\gamma x) \cap (\gamma x') \subseteq \gamma(comb'\ x x').
\end{aligned}$$

□

Definition. An insertion $(X, \gamma, (\mathscr{P}\ Eqn))$ is *Moore-closed* iff $\{\gamma x \mid x \in X\}$ is Moore-closed in $\mathscr{P}\ Eqn$.

COROLLARY 6.6. *Let* $(X, \gamma, (\mathscr{P}\ Eqn))$ *be a downward closed, Moore-closed insertion. Let* $\sqcap_X$ *be the meet operator on X. Then* $\sqcap_X$ *appr comb* (*and, in fact, is the best approximation*).

Let us sum up the results of this section. To define a data-flow analysis, one needs to lay down a complete (for termination preferably Noetherian)

lattice X of "descriptions." These descriptions should be designed to convey the desired kind of data-flow information. The information contents of a description should be specified by a function γ in such a way that $(X, \gamma, (\mathscr{P} Eqn))$ is an "insertion." One should also define a function c: $X \to X \to X$ that correctly emulates conjunction of ex-equations (or composition of substitutions) and a function u: $Atom \to Atom \to X \to X$ that correctly emulates unification of atoms in the presence of an ex-equation (or current substitution). That is, one must provide an "interpretation" (X, c, u) that is "sound." The result is a definition of a data-flow analysis that is automatically correct. As an example, we give in the following section a sound interpretation that specifies a very precise, nontrivial "groundness" analysis.

7. GROUNDNESS ANALYSIS USING BOOLEAN FUNCTIONS

In this section we give an example data-flow analysis for groundness propagation. This analysis is based on the scheme given in the previous section. Certain Boolean functions are used as descriptions. Since the analysis is intimately dependent on the nature of variables in a logic-programming language, let us briefly consider the role of variables.

A variable in a logic-programming language is very different from a variable in an imperative or functional programming language. It is sometimes referred to as a "logical variable" and characterized as "constrain-only." This is because, as we have seen, execution of a logic program proceeds by steps that continually narrow the set of possible values that a variable may take.

This characteristic of the execution of logic programs makes data-flow analyses harder in some ways, but it also opens up new views of some analysis problems. For example, it suggests the possibility of propagating conditional invariants of the form "*From this point on*, if x has (ever gets) property p, then y has (will have) property r." Example 7.7 makes it clearer what we mean by this "projecting into the future," but first we need to explain how dependency information can be represented. A statement such as "x is ground" may be represented by a propositional variable x. Groundness (or other) dependencies may then be represented by Boolean functions, such as that denoted by $y \to x$.

Since groundness and other interesting properties are not decidable, a data-flow analysis that operates in finite time can only give approximate information, in general. The statements that it produces carry a modality, as in "x is inevitably ground" or "x is possibly ground." For this reason, only the *positive* Boolean functions are useful. It we associate the meaning "x is inevitably ground" with the formula x, then $\neg x$ would mean "x may not always be ground," and this conveys no information at all, that is, exactly the information conveyed by the function *true*.

Definition. A *Boolean function* is a function F: $Bool^n \to Bool$. We call the set of all n-ary Boolean functions $Bfun_n$ and let it be ordered by logical consequence ($\models$). The function F is *positive* iff $F(true, \ldots, true) = true$. We denote the set of positive Boolean functions of n variables by Pos_n.

For simplicity we assume that we have some fixed number n of variables and leave out subscripts and the phrase "of n variables." The set of propositional variables $\{x_1, \ldots, x_n\}$ is referred to as *Pvar*. We also use propositional formulas as representations of Boolean functions without worrying about the distinction. Thus, we may speak of a formula as if it were a function and, in any case, denote it by F. As a reminder of the fact that a propositional formula is only one of a class that all represent a given Boolean function, we put square brackets around propositional formulas and think of the result as the class of equivalent formulas. By a slight abuse of notation, we sometimes apply logical connectives to these classes of equivalent formulas. We sometimes refer to the formulas as *groundness dependency formulas*.

It is well known that *Bfun* is a Boolean lattice, and in fact, *Pos* forms a Boolean sublattice of *Bfun*. In terms of propositional formulas, meet and join are given by conjunction and disjunction, respectively. In the context of a finite set *Pvar* of propositional variables, the complementation operation on *Pos* is given by $\dot{\neg} F = F \leftrightarrow \wedge Pvar$; see Cortesi et al. [1991]. (Here and in the following, we use the notation $\wedge\{\phi_1, \ldots, \phi_n\}$ for the formula $\phi_1 \wedge \cdots \wedge \phi_n$, and similarly for $\vee$.)

Logicians have, of course, studied *Pos* under several different names. However, the history of dependency clauses for data-flow analysis is rather short. Dart used a class of dependency formulas in his work in the area of deductive databases [Dart 1988; 1991]. Dart's class is strictly less expressive than *Pos*, which was suggested by Marriott and Søndergaard [1989a] (under the less suggestive name *Prop*) and further studied by Cortesi et al. [1991].

One can imagine many types of properties of data-flow information for which the dependency formula is a useful formalism. Here we are concerned with groundness. As an example, if the constraint $x = f(y, z)$ is generated during query evaluation, the relationship $x \leftrightarrow (y \wedge z)$ is deduced. Informally, the formula says that if x is (becomes) ground then so is (does) both y and z, and vice versa. Example 7.7 shows how this kind of information may be useful.

Syntactically, *Pos* has several interesting characterizations. For example (and surprisingly), it consists of exactly those Boolean functions that can be represented by propositional formulas using only the connectives $\rightarrow$ and $\wedge$.

Example 7.1. The Boolean function $[\neg x]$ is not in *Pos*. The function $[x \rightarrow y]$ is in *Pos*. In terms of $\wedge$ and $\leftrightarrow$, we could write it as $[x \leftrightarrow (x \wedge y)]$.

It is convenient to include the (nonpositive) Boolean function *false* as an approximation to the empty set of ex-equations. So from here on we deal with the domain $Pos_\perp = Pos \cup \{false\}$, ordered by logical consequence. Figure 5 shows $Pos_\perp$ in the dyadic case. The idea with using $Pos_\perp$ is that an ex-equation e is described by $\phi \in Pos_\perp$ exactly in case that, for every unifier θ of e, the truth assignment corresponding to the variables ground by θ satisfies ϕ.

Definition. For a substitution θ, let *grounds* θ be the truth assignment that maps a variable to true if θ grounds the variable and to false otherwise.

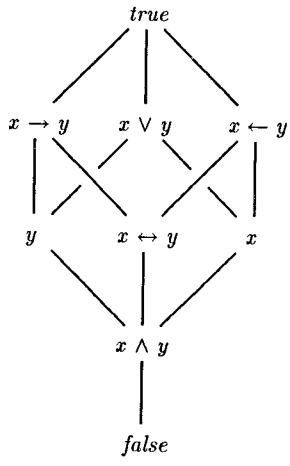

Fig. 5. $Pos_\perp$ functions of two variables.

That is, *grounds*: $Sub \to Var \to Bool$ is defined by

$$grounds\ \theta V \Leftrightarrow vars(\theta V) = \emptyset.$$

The concretization function γ: $Pos_\perp \to (\mathscr{P}\ Eqn)$ is defined by

$$\gamma\phi = \{e \in Eqn \mid \forall\theta \in unif\ e.(grounds\ \theta) \models \phi\}.$$

This concretization function maps $\phi \in Pos_\perp$ to the set of ex-equations e that have the property that, no matter how they further become constrained to some e', the groundness formula corresponding to e' satisfies ϕ.

Example 7.2. The Boolean function $[x \leftrightarrow y]$ describes both of the ex-equations $x = a \wedge y = b$ and $\exists u'.(x = u' \wedge y = u' \wedge u = u')$, but not the ex-equation $x = a$. However, $[x \leftrightarrow y]$ is not the *best* description for $x = a \wedge y = b$. For this ex-equation, the best description is $[x \wedge y]$, which in turn has $[x \leftrightarrow y]$ as a logical consequence. That is, $[x \leftrightarrow y]$ is a less precise approximation than $[x \wedge y]$.

As a further example, let $\phi = [x \wedge (y \leftrightarrow z)]$. Then $(x = a \wedge y = f(z))$ is approximated by ϕ, but $(x = a \wedge y = f(a))$ is not.

LEMMA 7.1. *The triple* $(Pos_\perp, \gamma, (\mathscr{P}\ Eqn))$ *is a downward closed, Moore-closed insertion.*

PROOF. Clearly, γ is monotonic and costrict, and $(Pos_\perp, \gamma, (\mathscr{P}\ Eqn))$ is downward closed. Let $\Phi \subseteq Pos_\perp$. Then $\wedge\Phi \in Pos_\perp$ and $\gamma(\wedge\Phi) = \bigcap_{\phi\in\Phi}(\gamma\phi)$, so $(Pos_\perp, \gamma, (\mathscr{P}\ Eqn))$ is Moore-closed. □

It follows from Corollary 6.6 that *comb* is best approximated by conjunction.

Definition. The function $comb_{gro}$: $Pos_\perp \to Pos_\perp \to Pos_\perp$ is defined by $comb_{gro}\ \phi\phi' = \phi \wedge \phi'$.

LEMMA 7.2. $comb_{gro}$ *appr comb.*

The function *unify* is somewhat more complex to approximate. Its definition makes use of $restrict_{gro}$, a projection function on propositional formulas, and mgu_{gro}, the analogue of *mgu* for propositional formulas. The motivation for approximating *unify* in this way comes directly from its definition.

Because $Pos_\perp$ is downward closed, we can approximate existential quantification of equations by existential quantification on Boolean functions.

Definition. The function $restrict_{gro}$: $(\mathscr{P}\ Var) \to Pos_\perp \to Pos_\perp$ is defined by

$$restrict_{gro}\ U\phi = \exists\overline{U}.\phi.$$

This is well defined because positive Boolean functions are closed under existential quantification. In particular, $\exists X.\phi = \{X \mapsto true\}\phi \vee \{X \mapsto false\}\phi$ (which also explains *why* $Pos_\perp$ is closed under existential quantification).

Example 7.3. We have

$$\begin{aligned} restrict_{gro}\{x, y, z\}[x \wedge (y \leftrightarrow v) \wedge (z \leftrightarrow v)] &= [x \wedge y \wedge z] \vee [x \wedge \neg y \wedge \neg z] \\ &= [x \wedge (y \leftrightarrow z)]. \end{aligned}$$

LEMMA 7.3. $restrict_{gro}$ *appr restrict.*

PROOF. By the definition of *restrict*, we must show that if $e \in \gamma\phi$ then $(\exists\overline{U}.e) \in \gamma\ (restrict_{gro}\ U\phi)$. This holds since

$$\begin{aligned} e \in \gamma\phi &\Rightarrow \forall\theta \in unif\, e.(grounds\ \theta) \vDash \phi \\ &\Rightarrow \forall\theta \in unif\, e.(grounds\ \theta|_U) \vDash \exists\overline{U}.\phi \\ &\Rightarrow \forall\theta \in unif(\exists\overline{U}.e).(grounds\ \theta|_U) \vDash \exists\overline{U}.\phi \\ &\Rightarrow \forall\theta \in unif(\exists\overline{U}.e).(grounds\ \theta) \vDash \exists\overline{U}.\phi \\ &\Rightarrow (\exists\overline{U}.e) \in \gamma(restrict_{gro}\ U\phi). \end{aligned}$$

□

Definition. The function mgu_{gro}: $Atom \to Atom \to Pos_\perp$ is defined by

$$\begin{aligned} mgu_{gro}\ AH = &\text{if } (mgu\ AH) = \emptyset \text{ then } false \text{ else} \\ &\text{let } \{\mu\} = (mgu\ AH) \text{ in} \\ &\quad [\wedge\{V \leftrightarrow (\wedge\, vars(\mu V)) \mid V \in dom\ \mu\}]. \end{aligned}$$

Example 7.4. Let $A = p(x, y)$ and $H = p(a, f(u, v))$. Then $mgu_{gro}\ AH = [x \wedge (y \leftrightarrow (u \wedge v))]$.

LEMMA 7.4. $(mgu_{gro}\ AH)$ *appr* $\{A = H\}$.

PROOF. The case when A and H are not unifiable is immediate. Otherwise, consider some $V \in dom\ \mu$, where μ is a most general unifier of A and H. Let $T = \mu V$. Now, $\theta \in unif[\![A = H]\!]$ implies that $\theta V = \theta T$. Thus, $vars(\theta V) = \emptyset$ iff, for all $V' \in (vars\ T)$, $vars(\theta V') = \emptyset$. Thus, $[V \leftrightarrow (\wedge\, vars(\mu V))]$ approximates μ. The result follows from the fact that $Pos_\perp$ is Moore-closed. □

Definition. The function $unify_{gro}$: $Atom \to Atom \to Pos_\perp \to Pos_\perp$ is defined by

$$unify_{gro}\ HA\phi = \text{let } \rho = ren(vars\ H)((vars\ A) \cup (vars\ \phi)) \text{ in}$$
$$restrict_{gro}(vars\ H)\big((\rho\phi) \wedge (mgu_{gro}\ H(\rho A))\big).$$

LEMMA 7.5. $unify_{gro}$ *appr* $unify$.

PROOF. This follows from the definition of *unify*, Theorem 4.4, and Lemmas 7.2–7.4 □

Example 7.5. Let $A = append(x, y, z)$, $H = append(nil, y, y)$, and $H' = append(u : x, y, u : z)$. Then

$$\begin{aligned} unify_{gro}\ HA[true] &= [true], \\ unify_{gro}\ AH[true] &= [x \wedge (y \leftrightarrow z)], \\ unify_{gro}\ H'A[true] &= [true], \\ unify_{gro}\ AH'[false] &= [false], \\ unify_{gro}\ AH'[x \wedge (y \leftrightarrow z)] &= [(x \wedge y) \leftrightarrow z]. \end{aligned}$$

To see how the last result comes about, consider the most general unifier of A and H' (after application of a name toggle): $\{x \mapsto x^\#, y \mapsto y^\#, z \mapsto z^\#, u \to u^\#, x \to x^\#, y^\# \to y, z^\# \to z, u^\# \to u\}$. We have that

$$mgu_{gro}\ AH' = [(x \leftrightarrow (u^\# \wedge x^\#)) \wedge (y \leftrightarrow y^\#) \wedge (z \leftrightarrow (u^\# \wedge z^\#))].$$

Conjoining the formula $[x \wedge (y \leftrightarrow z)]$ (after name toggling), we get

$$[(x \leftrightarrow (u^\# \wedge x^\#)) \wedge (y \leftrightarrow y^\#) \wedge (z \leftrightarrow (u^\# \wedge z^\#)) \wedge x^\# \wedge (y^\# \leftrightarrow z^\#)].$$

We need to restrict this to the set $\{x, y, z\}$. "Projecting away" $x^\#$ yields

$$[(x \leftrightarrow u^\#) \wedge (y \leftrightarrow y^\#) \wedge (z \leftrightarrow (u^\# \wedge z^\#)) \wedge (y^\# \leftrightarrow z^\#)].$$

Projecting $y^\#$ away yields $[(x \leftrightarrow u^\#) \wedge (y \leftrightarrow z^\#) \wedge (z \leftrightarrow (u^\# \wedge z^\#))]$, and projecting $u^\#$ away then yields $[(y \leftrightarrow z^\#) \wedge (z \leftrightarrow (x \wedge z^\#))]$. Finally, projecting $z^\#$ away yields $[z \leftrightarrow (x \wedge y)]$.

Definition. The *groundness analysis* $\mathbf{P_{gro}}$ is given by instantiating the data-flow semantics with the interpretation $(Pos_\perp, comb_{gro}, unify_{gro})$.

THEOREM 7.6. $\mathbf{P_{gro}}$ *appr* $\mathbf{P_{bas}}$.

PROOF. This follows immediately from Lemmas 7.1, 7.2, and 7.5, and from Theorem 6.4. □

Example 7.6. Let P be the *append* program

$append(nil, y, y).$
$append(u : x, y, u : z) \leftarrow append(x, y, z).$

Consider the query $A = \leftarrow append(x, y, z)$. To compute $\mathbf{P}_{\mathbf{gro}}[\![P]\!]\, A[true]$, the analysis proceeds as follows: Let $d_P = \sqcup_{C \in P}(\mathbf{C}_{\mathbf{gro}}[\![C]\!])$. We then have

$$\begin{aligned}
&(d_P \uparrow 0)A[true] = [false],\\
&(d_P \uparrow 1)A[true] = [x \wedge (y \leftrightarrow z)] \vee [false] \qquad = [x \wedge (y \leftrightarrow z)],\\
&(d_P \uparrow 2)A[true] = [x \wedge (y \leftrightarrow z)] \vee [(x \wedge y) \leftrightarrow z] = [(x \wedge y) \leftrightarrow z],
\end{aligned}$$

and $(d_P \uparrow 3) = (d_P \uparrow 2) = lfp\ d_P$. Thus, $\mathbf{P}_{\mathbf{gro}}[\![P]\!]\, A[true] = [(x \wedge y) \leftrightarrow z]$. Similarly, one can show that $\mathbf{P}_{\mathbf{gro}}[\![P]\!]\ A[z] = [x \wedge y \wedge z]$. In the "extended" version of $\mathbf{P}_{\mathbf{gro}}$, we find that, provided this holds in the initial call, all calls to *append* have a third argument that is inevitably ground.

Example 7.7. Consider the following (naive) program R to reverse lists:

$rev(nil, nil).$

$rev(x : y, z) \leftarrow append(u, x : nil, z), \textcircled{a}\ rev(y, u).$

Consider the query $A' = rev(x, y)$, and assume that we are interested in queries of that form with x being ground. A first approximation, $[x \wedge y]$, is obtained by considering the fact $rev(nil, nil)$:

$$(d_R \uparrow 0)A'[x] = [false],$$
$$(d_R \uparrow 1)A'[x] = [x \wedge y].$$

That is, so far it looks as if the query will result only in answers with x and y ground, but this may well be revised in a subsequent approximation step.

In processing the recursive clause, we use the approximation already obtained for *append*. In terms of the variables in R, the approximation is $[(u \wedge x) \leftrightarrow z]$. Conjoining this with the "current constraint" $[x \wedge y]$, we arrive (after simplification) at the formula

$$\phi\colon [x \wedge y \wedge (z \leftrightarrow u)]$$

as the approximation that holds at the program point marked ⓐ. It expresses that x and y will be ground and that z is (or will become) ground iff u is (will). This relation between z and u is what we had in mind when saying that invariants may be "projected into the future."

We now see the value of this: The (current) formula for the last atom in the clause is $[y \wedge u]$ (from $d_R \uparrow 1$), and conjoining this with ϕ, we get, after simplification, $[x \wedge y \wedge z \wedge u]$. In terms of the variables in the query, this translates (after restriction) to $x \wedge y$; that is,

$$(d_R \uparrow 2)A'[x] = [x \wedge y].$$

We conclude that $[x \wedge y]$ is a fixpoint; that is, y will indeed become ground in every answer, assuming that x was. This information, in turn, can be translated into information about calls. It says that if *rev* is queried with a ground first argument, all of the generated calls to *rev* will have a ground first argument.

The details of implementing the groundness analysis are beyond the scope of this paper. The important choice is how to represent Boolean functions in a

way that supports their efficient manipulation. Le Charlier and Van Hentenryck [1993] used ordered binary-decision diagrams [Bryant 1992] for this purpose and reported very good results, in both precision and efficiency of the analysis. Marriott and Søndergaard [1993] showed that a groundness analysis using *Pos* has the property that queries need only be analyzed in their most general form; analysis of instances can be done simply by conjoining elements in *Pos*, speeding up the analysis (this is related to Jacobs and Langen's "condensing" [Jacobs and Langen 1992]). Codish and Demoen [1993] reported good results with their implementation of *Pos*, which is based on an "abstract compilation" approach, that is, on the generation of a constraint program whose execution, in turn, corresponds to the data-flow analysis.

8. IMPLEMENTATION ISSUES

The denotational equations given in this paper can be considered *definitions* of logic-program semantics ($\mathbf{P}_{\mathbf{bas}}$) and of data-flow analyses (the instances of **P**), presented in the same formal language. What is being defined is orthogonal to the ways in which it may be computed, and the equations themselves contain little commitment as to how one could implement the data-flow analyses. Indeed, one of the purposes of this paper is to show how the orthogonal issues of approximation and implementation can (and should) be kept separate, as depicted in Figure 3.

Read naively, the equations specify a highly redundant way of computing certain mathematical objects. On the other hand, the denotational definitions can be given a "call-by-need" reading that guarantees that the same partial result is not repeatedly recomputed and only computed at all if it is needed for the final result. Such readings are independent of *what* is being defined, and a reader is free to choose whichever is desired. In fact, with such a call-by-need reading the definition of **P** is, modulo syntactic rewriting, a working implementation of a generic data-flow analyzer written in a functional programming language [Errington and Søndergaard 1992]. In programming languages that do not support a call-by-need semantics, implementation is somewhat harder.

The direct implementation obtained from "running the definition" is interpretive and not very efficient. Hermenegildo et al. [1992] suggested that analysis can be sped up by specializing an abstract interpreter to a given source program, thus in effect performing "abstract compilation." The same idea is implicit in earlier work on abstract interpretation. For example, Mycroft [1981] performed strictness analysis by generating functional programs that compute Boolean values that, in turn, provide the desired strictness information.

In the remainder of this section, we sketch a traditional interpretive implementation.[7] This is instructive because it shows the close relationship

[7] Since it is really the collecting semantics that must be approximated, a data-flow analysis is seen as a fixpoint computation over a domain of *annotations* of a given program. An annotation is a mapping from the set of program points to the lattice of descriptions used, and the computation continues as long as a new description is found to obtain at some program point.

between our semantic definition and other generic analyzers (written in C or PROLOG) based on the AND/OR tree framework of Bruynooghe et al. [Bruynooghe 1987; Bruynooghe et al. 1987].

To avoid redundant computations, the result of invoking atom A in the context of description x should be recorded. Such memoing can be implemented using function graphs. The function graph for a function f is the set of pairs $\{(x, f(x)) \mid x \in dom\ f\}$, where $dom\ f$ denotes the domain for f. The computation of a function graph is best done in a demand-driven fashion: We only compute as much of it as is necessary to answer a given query. This corresponds to the "minimal function graph" semantics used by Jones and Mycroft [1986].

However, matters are complicated by the fact that we are performing a fixpoint computation. Recall that we want to compute a partial function d: $Atom \to X \to X$. Thus, for each atom A in the program or query, we must keep a function graph mapping each "input" description x to the corresponding "output" description $x' = dAx$. However, as the function definition is recursive, we must compute the result by means of the function's Kleene sequence. This means that the (partial) function graph that we compute does not simply correspond to a function that becomes more and more defined; the result corresponding to input x may well be revised several times as we go. An example is given by the computation of dAx in Example 7.6, where $A = append(x, y, z)$ and $x = true$: At one stage the value is assumed to be $[x \wedge (y \leftrightarrow z)]$, but later this is revised to $[(x \wedge y) \leftrightarrow z]$.

This means that the function graph cannot be used (as might have been hoped) to simply look up values dAx. Rather, we must organize the computation such that a request for dAx is always taken as a request to *recompute* this value, and only if that computation leads to a recursive request for dAx will the current value be used; that is, a simple lookup is performed.

Another, less important, way to improve efficiency is only to compute the function graph "modulo variable renaming." The reason we can do this is that the meaning of a program is independent of variable names in the following sense:

PROPOSITION 8.1. *Let* ρ *be a name toggle. Then* $\rho(\mathbf{P}_{\mathbf{bas}}[\![P]\!]\, A\, E) = \mathbf{P}_{\mathbf{bas}}[\![P]\!](\rho A)(\rho E)$.

Thus, in an implementation we do not want to distinguish, for example, the calls $(p(x, f(y, x)), \{x \mapsto y\})$ and $(p(u, f(v, u)), \{u \mapsto v\})$. The simplest solution is to annotate variables with numbers as they are met, starting from 1, and to refer to them through these numbers. For example, the two calls above are both referred to as $(p(1, f(2, 1)), \{1 \mapsto 2\})$.

Other avenues are open for improving the efficiency of a generic data-flow analyzer:

—A certain amount of partial evaluation can be performed to speed up the fixpoint computation. For example, a given body atom will be unifiable with a fixed set of clause heads, and the relevant clauses should not be looked up repeatedly. Rather, each atom should be annotated with the

clauses it may successfully call and the corresponding most general unifiers.

—Le Charlier et al. [1990] suggest maintaining a (dynamic) dependency graph for the following reason: Updating an entry in the function graph will not necessarily mean that every other entry has to be recomputed, and it is, in fact, possible to keep track of which other entries will be affected. If a functional programming language is used, the graph need not even be explicit; proper use of continuations can achieve the same effect with less effort on the implementor's part, though possibly less efficiently [Errington and Søndergaard 1992]. More specifically, rather than yielding a result description x' only, dAx should return a pair (x', k), where k is a continuation. The idea is that k is invoked whenever x' is revised, automatically causing entries that depend on dAx to be recomputed.

—The monotonicity of the mapping dA can be utilized in the following way: Whenever the value of dAx gets updated to x', we know that the values of dAx'' for $x \sqsubseteq x''$ must be at least x', and we can therefore update each such entry to $x' \sqcup y$, where y is its current value. Similarly, when it is discovered that a new entry dAx is needed (i.e., its value is not $\bot$), we may look up all of the values of dAx' for $x' \sqsubseteq x$ and use their least upper bound as the initial value for dAx. To make these kinds of consequential updates fast, Le Charlier et al. arrange input descriptions as a Hasse graph, that is, a structure that directly reflects the partial ordering of the descriptions.

9. RELATED WORK

An indication of the usefulness and wide applicability of abstract interpretation for logic programming is the amount of work published on the topic in recent years. A recent special issue of *Journal of Logic Programming* (1992) has been devoted to the topic, and we refer readers to the extensive bibliography of Cousot and Cousot [1992].

In particular, obtaining information about calls to clauses by approximating the SLD semantics (as studied in this paper) seems useful, as many kinds of code improvement that can be performed automatically by a compiler depend on information about calls that may take place at run time, and other kinds of program transformation make use of that information as well.

A number of papers have been concerned with generic frameworks for abstract interpretation of logic programs. By this is usually meant a general setting that allows one to express a number of data-flow analyses in a uniform way, just as we have done in the present paper through our "data-flow semantics." Almost all of these frameworks approximate the SLD semantics. We now compare our semantic definitions to other denotational SLD-based definitions and our framework to other generic analysis frameworks.

Early work on SLD-based semantic definitions for logic programs was done by Jones and Mycroft [1984], who addressed both operational and denotational semantics. Debray and Mishra [1988] gave a thorough exposition of a denotational definition, including a proof of its correctness with respect to

SLD. Both Jones and Mycroft and Debray and Mishra assumed a left-to-right computation rule and a depth-first search of SLD trees (as in Prolog), and both definitions captured nontermination (unlike ours). Both used *sequences* of substitutions as denotations, rather than sets, which gave the definitions a rather different flavor. The definition used by Jones and Søndergaard [1987] achieves certain simplifications by assuming a parallel search rule and, consequently, manipulates *sets* of substitutions. The use of substitutions forces it to employ an elaborate renaming technique that complicates semantic definitions somewhat. Winsborough [1988a; 1988b] and Jacobs and Langen [1992] suggested denotational definitions along similar lines. Marriott and Søndergaard [1989b] gave a uniform presentation of both "bottom-up" (i.e., T_P-style [Apt and van Emden 1982]) and "top-down" (SLD-based) definitions by expressing both in terms of operations on lattices of substitutions, as far as this is possible. In particular, they showed that operations such as unification and composition of substitutions can be adequately dissolved into lattice operations to simplify definitions, an idea that has formed the point of departure for the definitions presented here.

Abstract interpretation of logic programs was first mentioned by Mellish [1981], who suggested it as a way to formalize mode analysis. An occur check analysis that was formalized as a nonstandard semantics was given by Søndergaard [1986]. Some of the techniques used in the present paper can be traced back to that work. This is the case with the principle of performing unification both on call and return so as to facilitate that only local variables need be manipulated at any stage (this was referred to as a principle of "locality"). The work also established the principle of binding information to *variables* in a program throughout computations, rather than to argument positions, as is more usual in other frameworks [Bruynooghe 1987; Mannila and Ukkonen 1987; Mellish 1987].

Other things being equal, this improves precision. For example, consider a mode analysis of the program

$\leftarrow p(f(x))$.
$p(f(u))$.

using the two modes "free" (unbound or bound to a variable) and "any." The "argument position" methods will funnel mode information about the variables x and u through the argument position of p and assign x the mode "any," while clearly x could be more precisely deemed "free." To counteract such behavior, an "argument position" method must use more fine-grained descriptions and pay the price of more expensive "abstract operations."

A framework for the abstract interpretation of logic programs was first given by Mellish [1987]. Mellish's semantics is an operational parallel to our "lax" semantics with the imprecision that this implies: Success patterns are not associated with their corresponding call patterns, so success information is propagated back, not only to the atom that actually called the clause, but to all atoms that unify with the clause's head. The application that had Mellish's interest in particular was mode analysis. Debray [1986] subsequently investigated this application in more detail and pointed to a problem in Mellish's

application (the so-called aliasing problem, which may manifest itself as either a soundness or a completeness problem, depending on the particular data-flow analysis).

A framework for the abstract interpretation of logic programs based on a denotational definition of SLD was given by Jones and Søndergaard [1987]. This was the first denotational approach to abstract interpretation of logic programs, and the first paper in the area to apply the idea of a generic data-flow algorithm with a few basic operations being parameters. The framework allowed even the base (or standard) semantics to be expressed as an instance of the data-flow semantics. This has the advantage of providing a very clean cut between a semantic definition that is *precise* (unlike our lax and data-flow semantics) and interpretations in which *all* introduced imprecision resides. In the present paper, we have abandoned this approach only to simplify our presentation. Jones and Søndergaard used operations "call" and "return," which in the present approach have been replaced by "unify" and "comb." We find this conceptually cleaner.

Kanamori and Kawamura [1993] suggested a framework based on OLDT resolution [Tamaki and Sato 1986], which essentially is SLD resolution extended with memoing, so as to avoid redundant computation. Bruynooghe et al. [Bruynooghe 1987; Bruynooghe et al. 1987] suggested an AND/OR tree-based framework that expresses the ideas behind the Jones–Søndergaard scheme at a lower level of abstraction. Efficient implementation of data-flow analysis engines based on the AND/OR tree framework are described in Le Charlier et al. [1990].

The framework used by Winsborough [1988a; 1988b] is rather close to ours. In particular, one semantic definition (Winsborough's "total function graph semantics" [Winsborough 1988b]) is almost identical to our base semantics, the difference being that, where we employ ex-equations, it works with substitutions that are "canonized" to bar variants of a substitution from introducing redundancy. (Mellish [1987] used the same idea.)

Debray [1988] studied a framework for data-flow analysis with the point of departure that analyses must be *efficient*. He identified a property of description domains ("substitution closure") and gave a complexity analysis to support the claim that the corresponding class of data-flow analyses can be implemented efficiently. Our groundness analysis falls outside Debray's class, as does any data-flow analysis that attempts to maintain information about possible aliasing (see also below).

In other studies of logic program analysis [Marriott and Søndergaard 1989b], we have found it useful to distinguish between "bottom-up" and "top-down" analysis. This distinction is not clear-cut, but we think of a top-down semantics as one that allows for extraction of information about the SLD tree that corresponds to the execution of a program given some query. Bottom-up analysis is not based on such a semantics to begin with, but on a T_P-style semantics [Apt and van Emden 1982], and therefore, it cannot provide information about calls that will take place at run time. Bottom-up analysis suffices for several applications, though. It is not only the conceptu-

ally simplest of the two, it also allows for efficient derivation of *query-independent* information about a program.

It has been suggested that top-down analysis can be done by first applying the so-called magic set transformation to the source program/query pair and then computing call patterns in a bottom-up fashion from the transformed program. See, for example, Debray and Ramakrishnan [1991] and Nilsson [1991], and the closely related Alexander template approach [Kanamori 1993].

For instance, assume that we are given the *append* program from before:

$append(nil, y, y).$
$append(u : x, y, u : z) \leftarrow append(x, y, z).$

and the query $\leftarrow append(x, y, z)$. The program/query combination is transformed to the following:

$append(nil, y, y) \leftarrow call\_append(nil, y, y).$
$append(u : x, y, u : z) \leftarrow call\_append(u : x, y, u : z), append(x, y, z).$
$call\_append(x, y, z) \leftarrow call\_append(u : x, y, u : z).$
$call\_append(x, y, z).$

A bottom-up evaluation of the transformed program reflects what happens in a top-down evaluation of the original program. In our example, the unit clause $call\_append(x, y, z)$ expresses that *append* will be called with arguments (x, y, z). The first clause in the transformed program says that if an instance of $append(nil, y, y)$ is called then it will succeed, and so on.

In general, each clause $H \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$ in P gives rise to $n + 1$ clauses, namely,

$$H \leftarrow call\_H, A_1, \ldots, A_n$$

and, for $i \in \{1, \ldots, n\}$,

$$call\_A_i \leftarrow call\_H, A_1, \ldots, A_{i-1}.$$

A query $\leftarrow A_1, \ldots, A_m$ is replaced by m clauses $(i \in \{1, \ldots, m\})$:

$$call\_A_i \leftarrow A_1, \ldots, A_{i-1}.$$

Applying bottom-up abstract interpretation to the transformed program then yields call pattern information about the original program. However, whether this approach is better than what we have suggested in this paper is not clear, as the magic set transformation is not free and the resulting program is typically much larger than the original.

10. CONCLUSION

One contribution of this paper is a new framework for the abstract interpretation of definite logic programs based on SLD resolution. It captures the essence of a major class of data-flow analyses used in many different logic-programming tools, in particular, compilers. It is based on *simple* semantic definitions for logic programs and data-flow analyses. The simplicity is partly due to the use of sets of existentially quantified equations, rather than

sequences of substitutions. There is a high degree of congruence between the standard semantics and the data-flow analyses, emphasizing their close relationship.

We have demonstrated the usefulness of our framework by developing a nontrivial groundness analysis and proving its correctness. The groundness analysis uses positive Boolean functions to capture groundness dependencies among variables in a program. This results in a simple and clean, yet highly precise, data-flow analysis.

However, we feel that the most important contribution of our paper is a theory of "language-independent abstract interpretation," which is suitable for logic-programming-style languages and other nondeterministic programming languages. Our theory is a modification of the denotational approach developed by Nielson in the context of deterministic programming languages. The key is to formalize abstract interpretation in terms of a powerful metalanguage such as that of denotational semantics. This allows for generality at different levels. First, it allows for comfortable reasoning at exactly the level of abstraction called for by any particular class of applications or data-flow analyses. Second, the proof of correctness of a particular data-flow analysis becomes simpler, since parts of it can be conducted at the level of the metalanguage once and for all. Finally, most of the theory is independent of any particular programming language, since it is expressed in terms of the metalanguage only.

Our theory facilitates development of analyses in other related programming-language paradigms. It has been applied successfully to develop a generic framework for the analysis of logic programs where the semantics is based on a fixpoint characterization of the least model [Marriott and Søndergaard 1992], rather than the SLD-based semantics considered here. Our theory has also been used to formalize data-flow analysis for constraint logic-programming languages [Marriott and Søndergaard 1990; Marriott and Stuckey 1993], logic-programming languages that allow the use of "safe" negation [Marriott et al. 1990], and even concurrent constraint programming languages [Falaschi et al. 1993]. Other related languages in which our theory should be useful include logic-programming languages with delay mechanisms ("freeze," "wait," "when," etc.), as well as deductive databases.

ACKNOWLEDGMENTS

We would like to thank Will Winsborough for his insightful comments. Very thorough referees have reviewed a previous version of this paper, and we thank them for their constructive critique.

REFERENCES

ABRAMSKY, S., AND HANKIN, C., EDS. 1987. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, Chichester, U.K.

APT, K., AND VAN EMDEN, M. 1982. Contributions to the theory of logic programming. *J. ACM 29*, 3 (July), 841–862.

BRUYNOOGHE, M. 1991. A practical framework for the abstract interpretation of logic programs, *J. Logic Program. 10*, 2, 91–124.

BRUYNOOGHE, M. 1987. A framework for the abstract interpretation of logic programs. Rep. CW 62, Dept. of Computer Science, Univ. of Leuven, Belgium.

BRUYNOOGHE, M., AND JANSSENS, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Program. 13*, 2–3 (July), 205–258

BRUYNOOGHE, M., JANSENNS, G., CALLEBAUT, A., AND DEMOEN, B. 1987. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 4th International Symposium on Logic Programming* (San Francisco, Calif.). IEEE, New York, 192–204

BRYANT, R. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24*, 3 (Sept.), 293–318.

CODISH, M., AND DEMOEN, B. 1993. Analyzing logic programs using "Prop"-ositional logic programs and a magic wand. In *Logic Programming: Proceedings of the 1993 Symposium*, D. Miller, Ed. MIT Press, Cambridge, Mass., 114–129.

CORTESI, A., FILÉ, G., AND WINSBOROUGH, W. 1991. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science* (Amsterdam, The Netherlands). IEEE, New York, 322–327.

COUSOT, P., AND COUSOT, R 1992. Abstract interpretation and application to logic programs. *J. Logic Program. 13*, 2–3 (July), 103–179.

COUSOT, P., AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages* (San Antonio, Tex.). 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 Annual ACM Symposium on Principles of Programming Languages* (Los Angeles, Calif.). ACM, New York, 238–252.

DART, P. 1991. On derived dependencies and connected databases. *J. Logic Program. 11*, 2 (Aug.), 163–188.

DART, P. 1988. Dependency analysis and query interfaces for deductive databases. Ph.D. thesis, Dept. of Computer Science, The Univ. of Melbourne, Australia.

DEBRAY, S. K. 1988. Efficient dataflow analysis of logic programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, Calif.). ACM, New York, 260–273.

DEBRAY, S. K. 1986. Global optimization of logic programs. Ph.D. thesis, Dept. of Computer Science, State Univ. of New York at Stony Brook, New York.

DEBRAY, S. K., AND MISHRA, P. 1988. Denotational and operational semantics for Prolog. *J. Logic Program. 5*, 1 (Mar.), 61–91.

DEBRAY, S. K., AND RAMAKRISHNAN, R. 1991. Canonical computation of logic programs. Draft.

DONZEAU-GOUGE, V. 1978. Utilisation de la sémantique dénotationelle pour l'étude d'interprétations non-standard. Res. Rep. 273, IRIA, Le Chesnay, France.

ERRINGTON, D. L., AND SØNDERGAARD, H. 1992. Absinth: A generic dataflow analyser for logic programs. Tech. Rep. 92/19, Dept. of Computer Science, Univ. of Melbourne, Australia.

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1993. Compositional analysis for concurrent constraint programming. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (Montreal, Canada). IEEE, New York, 210–221.

HECHT, M. 1977. *Flow Analysis of Computer Programs*. North-Holland, Amsterdam, The Netherlands.

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.

JACOBS, D., AND LANGEN, A. 1992. Static analysis of logic programs for independent and-parallelism. *J. Logic Program. 13*, 2–3 (July), 291–314.

JENSEN, J. 1965. Generation of machine code in Algol compilers. *BIT 5*, 235–245.

JONES, N. D. 1981. Flow analysis of lambda expressions. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming*, S. Even and O. Kariv, Eds. Lecture Notes in Computer Science, vol. 115. Springer-Verlag, New York, 114–128.

JONES, N. D., AND MUCHNICK, S. S. 1981. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, N J., 102–131.

JONES, N. D., AND MYCROFT, A. 1986. Dataflow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla.). ACM, New York, 296–306.

JONES, N. D., AND MYCROFT, A. 1984. A stepwise development of operational and denotational semantics for Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*. (Atlantic City, N.J.). IEEE, New York, 289–298.

JONES, N. D., AND SØNDERGAARD, H. 1987. A semantics-based framework for the abstract interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood, Chichester, U.K., 123–142.

KANAMORI, T. 1993. Abstract interpretation based on Alexander templates. *J. Logic Program. 15*, 1–2 (Jan.), 31–54.

KANAMORI, T., AND KAWAMURA, T. 1993. Abstract interpretation based on OLDT resolution. *J. Logic Program. 15*, 1–2 (Jan.), 1–30.

KLEENE, S. C. 1952. *Introduction to Metamathematics*. North-Holland, Amsterdam, The Netherlands. (Originally published by Van Nostrand, New York.)

LE CHARLIER, B., AND VAN HENTENRYCK, P. 1993. Groundness analysis for Prolog. In *Proceedings of the 3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Copenhagen, Denmark). ACM, New York, 99–110.

LE CHARLIER, B., MUSUMBU, K., AND VAN HENTENRYCK, P. 1990. A generic abstract interpretation algorithm and its complexity analysis. In *Logic Programming: Proceedings of the 8th International Conference*, K Furukawa, Ed. MIT Press, Cambridge, Mass., 64–78.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag, New York.

MANNILA, H., AND UKKONEN, E. 1987. Flow analysis of Prolog programs. In *Proceedings of the 4th Symposium on Logic Programming*. (San Francisco, Calif.). IEEE, New York, 205–214.

MARRIOTT, K. 1993. Frameworks for abstract interpretation. *Acta Inf. 30*, 2, 103–129.

MARRIOTT, K., AND SØNDERGAARD, H. 1993. Precise and efficient groundness analysis. *ACM Lett. Program. Lang. Syst. 2*, 1–4, 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. 1990. Analysis of constraint logic programs. In *Logic Programming: Proceedings of the North American Conference 1990*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 531–547.

MARRIOTT, K., AND SØNDERGAARD, H. 1989a. Notes for a tutorial on abstract interpretation of logic programs. Presented to North American Conference Logic Programming (Cleveland, Ohio).

MARRIOTT, K., AND SØNDERGAARD, H. 1989b. Semantics-based dataflow analysis of logic programs. In *Information Processing 89*, G. X. Ritter, Ed. North-Holland, Amsterdam, The Netherlands, 601–606.

MARRIOTT, K., AND STUCKEY, P. 1993. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S.C.) ACM, New York, 334–344.

MARRIOTT, K., SØNDERGAARD, H., AND DART, P. 1990. A characterization of non-floundering logic programs. In *Logic Programming: Proceedings of the North American Conference 1990*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 661–680.

MELLISH, C. S. 1987. Abstract interpretation of Prolog programs. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood, Chichester, U.K., 181–198.

MELLISH, C. S. 1981. The automatic generation of mode declarations for Prolog programs. DAI Res. Pap. 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland.

MYCROFT, A. 1981. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh, Scotland.

NAUR, P. 1963. The design of the Gier Algol compiler, part II. *BIT 3*, 145–166.

NIELSON, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comput. Sci. 69*, 2 (Dec.), 117–242.

NIELSON, F. 1988. Strictness analysis and denotational abstract interpretation. *Inf. Comput. 76*, 1 (Jan.), 29–92.

NIELSON, F. 1982. A denotational framework for data flow analysis. *Acta Inf. 18*, 265–287.

NILSSON, U. 1991. Abstract interpretation: A kind of magic. In *Programming Language Implementation and Logic Programming*, J. Małuszyński and M Wirsing, Eds. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, New York, 299–309.

REYNOLDS, J. C. 1969. Automatic computation of data set definitions. In *Information Processing 68*, A Morrell, Ed. North-Holland, Amsterdam, The Netherlands, 456–461

SCHMIDT, D. 1986. *Denotational Semantics: A Methodology for Language Development* Allyn and Bacon, Boston, Mass.

SINTZOFF, M. 1972. Calculating properties of programs by valuation on specific models. In *Proceedings of the ACM Conference on Proving Assertions about Programs. SIGPLAN Not. 7*, 1 (Jan.), 203–207.

SØNDERGAARD, H. 1989. Semantics-based analysis and transformation of logic programs. Ph.D. thesis, Dept. of Computer Science, Univ. of Copenhagen, Denmark.

SØNDERGAARD, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of ESOP 86*, B. Robinet and R. Wilhelm, Eds. Lecture Notes in Computer Science, vol. 213. Springer-Verlag, New York, 327–338

TAMAKI, H., AND SATO, T. 1986. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, E. Shapiro, Ed Lecture Notes in Computer Science, vol. 240. Springer-Verlag, New York, 84–98.

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

WINSBOROUGH, W. 1988b. Source-level transforms for multiple specialization of Horn clauses (extended abstract). Tech. Rep. 88-15, Dept. of Computer Science, Univ. of Chicago, Illinois.

Received March 1990; revised February 1993; accepted June 1993