On the Complexity Analysis of Static Analyses

DAVID MCALLESTER

AutoReason.com

Abstract. This paper argues that for many algorithms, and static analysis algorithms in particular, bottom-up logic program presentations are clearer and simpler to analyze, for both correctness and *complexity*, than classical pseudo-code presentations. The main technical contribution consists of two theorems which allow, in many cases, the asymptotic running time of a bottom-up logic program to be determined by inspection. It is well known that a datalog program runs in $O(n^k)$ time where k is the largest number of free variables in any single rule. The theorems given here are significantly more refined. A variety of algorithms are presented and analyzed as examples.

Categories and Subject Descriptors: D3 [**Programming Languages**]; D.1.6 [**Programming Techniques**]: Logic Programming; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms; F.2 [**Analysis of Algorithms and Problem Complexity**]

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Algorithms, complexity analysis, logic programming, models of computation, program analysis, programming languages

1. *Introduction*

This paper presents two theorems that place upper bounds on the running time of bottom-up logic programs. The association of a running time with a logic program allows the program to be viewed as specifying a particular algorithm. This paper also argues that the ability to easily assign running times to bottom-up logic programs makes logic programs a useful general framework for expressing and analyzing static analysis algorithms. This position is supported through a variety of examples of static analysis algorithms expressed and analyzed as logic programs.

1.1. LOGIC PROGRAMS AS ALGORITHMS. In many cases bottom-up (or forward chaining) logic programs are clearer than programs involving classical iteration and recursion control structures. Consider transitive closure. A bottom-up logic program for transitive closure can be given with the single rule $P(x, y) \wedge P(y, z) \rightarrow P(x, z)$. We can view this rule as a program where the input is a "graph" represented as a set of assertions of the form $P(c, d)$ and the output is the set of assertions derivable from the input using the rule, that is, the transitive closure of the input. The inference

Authors' address: AutoReason.com, 85 Magnolia Dr., New Providence, NJ 07975, e-mail: mcallester@autoreason.com.

rule is arguably the clearest and most concise possible definition of the notion of transitivity—the program itself is arguably the clearest possible *specification* of the desired output.

More generally, given an initial database D of ground atomic formulas and a set R of first-order Horn clauses, we consider the deductive closure $R(D)$, that is, the set of all ground atomic formulas derivable from the "premises" in D using the "rules" in R. In general we think of a given rule set R as a program, the premise set D as the input to that program, and the deductive closure $R(D)$ as the output.

One of the most fundamental properties of an algorithm is its running time. Here we are interested in the time required to compute the output $R(D)$ as a function of the size (or other proerties) of the input D. There is one well-known result on this running time for the special case where R is range-restricted and datalog, that is, every variable in the conclusion of a rule appears in some antecedent of that rule and no rule in R involves function symbols other than constants. If R is range-restricted and datalog then $R(D)$ is always finite (for finite D) and $R(D)$ can be computed in $O(n^k)$ time where n is the number of terms in D and k is the maximum over all rules in R of the number of variables in that rule. In the range-restricted datalog case, one can compute $R(D)$ in the time given by this variable counting bound simply by constructing all instances of all rules over the terms in D and then applying the well-known linear time algorithm for the deductive closure of a set of ground Horn clauses [Downing and Gallier 1984]. The transitivity rule above is range-restricted and datalog. So the variable counting bound implies that transitive closure can be computed in $O(n^3)$ time where n is the number of nodes in the input graph.

Although the variable counting bound gives an approriate analysis for the single transitivity rule, it turns out that variable counting is too crude for most algorithms. A more efficient transitive closure algorithm for sparse inputs consists of the two rules $\text{EDGE}(x, y) \rightarrow \text{PATH}(x, y)$ and $\text{EDGE}(x, y) \wedge \text{PATH}(y, z) \rightarrow \text{PATH}(x, z)$. Note that the more efficient program has fewer ways of instantiating the antecedents of the transitivity rule. Let e be the number of input edges and n be the number of nodes. In the more efficient program there are e instances of the first antecedent of the transitivity rule and, for each such instance, at most n ways of filling in the final variable. This gives at most en ways of filling in the left-hand side of the rule. The run time theorem given in Section 3 implies that the more efficient program runs in $O(en)$ time rather than the $O(n^3)$ time predicted by variable counting.

As another example consider context free parsing. We can take the input to be a context-free grammar in Chomsky normal form and a string of terminal symbols. The grammar can be represented by a set of assertions of the form $A \Rightarrow BC$ and $A \Rightarrow a$ where A, B, and C are nonterminal symbols and a is a terminal symbol. We can represent the string by a set of assertions of the form $s_i = a$, which states that the ith symbol in the string is the terminal symbol a. Now consider the following program for context-free parsing:

$$\frac{X \Rightarrow y \qquad s_i = y}{X \Rightarrow s_{i,i}} \qquad \frac{X \Rightarrow YZ \qquad Y \Rightarrow s_{i,j} \qquad Z \Rightarrow s_{j+1,k}}{X \Rightarrow s_{i,k}}$$

This program computes all assertions of the form $A \Rightarrow s_{i,j}$ where the nonterminal X generates the string $s_i \cdots s_j$. As in the case of transitive closure, it can at least be

argued that the rules themselves are the clearest possible formal specification of the desired output. Note that if $|G|$ is the number of productions in the grammar and n is the length of the input string then there are only $O(|G|n^3)$ provable instances of the triple of antecedents in the second rule. The run time complexity theorem in Section 3 implies that the run time of this algorithm is $O(|G|n^3)$. Note that simple variable counting would yield $O(n^6)$ since the second rule involves the six variables X, Y, Z, i, j, and k. This is a logic program presentation of the Cocke-Kasimi-Younger (CKY) algorithm for context-free parsing.

1.2. APPLICATIONS TO STATIC ANALYSIS. After presenting the general run time theorems, this paper focuses on the application of these thoerems in the area of static analysis—analysis done by compilers or other tools for manipulating software that determines properties of the computer program being manipulated. For example, an optimizing compiler can often determine that, at a certain point in the program, the current value of a certain variable will not be used again. If the value of a variable is being stored in a register, and that value is no longer needed, then the register can be overwritten without storing its current value back into memory or onto the program stack. Determining that the value of a variable is no longer needed is called *liveness analysis*. Liveness analysis is "static" in the sense that it is performed at compile time rather than run time—liveness is determined by examining the (static) text of the program without relying on any (dynamic) execution. This paper presents a variety of static analysis algorithms as bottom-up logic programs. In most cases the programs (inference rules) are arguably the clearest possible specification of the computed output. Furthermore, the running time associated with these programs by virtue of the general run time theorems is either the best known or within a polylog factor of the best known.

1.3. OVERVIEW. Section 2 surveys related work from a variety of fields. Section 3 presents the first run time theorem and some basic examples. Sections 4, 5, and 6 present, respectively, liveness analysis, data flow analysis, and flow analysis (both data and control) in the lambda calculus. Section 7 presents a run time theorem for an extended bottom-up programming language incorporating the union-find algorithm. Sections 8 and 9 present unification and congruence closure algorithms, respectively. Section 10 presents a variant of Henglein's algorithm for typability in a version of the Abadi-Cardelli object calculus [Henglein 1999]. This last example is interesting for two reasons. First, the algorithm is not obvious—the first published algorithm for this problem used an $O(n^3)$ dynamic transitive closure algorithm [Palsberg 1995]. Second, Henglein's presentation of the quadratic algorithm uses classical pseudo-code and is fairly complex. Here we show that the algorithm can be presented naturally as a small set of inference rules whose running time is easily derived from a general run time theorem for logic programs.

2. *Related Work*

Bottom-up logic programming has been widely studied in the context of deductive databases [Vardi 1982; Ullman 1989; Naughton and Ramakrishnan 1991; Ullman and Ramakrishnan 1995]. The basic idea in deductive databases is to extend the notion of a database query to include recursion. For example, given a database with a parent relation one might ask for all ancestors of a given person where

an ancestor is either a parent or an ancestor of a parent. The recursive definition of "ancestor" can be viewed as the rules PARENT$(x, y) \rightarrow$ ANCESTOR(x, y) and PARENT$(x, y) \wedge$ ANCESTOR$(y, z) \rightarrow$ ANCESTOR(x, z). To evaluate the recursive query, one can compute the "intentional" ancestor relation from the "extensional" parent relation stored in the database. Computing the intensional relation is formally equivalent to computing a deductive closure as discussed in the introduction. Various optimizations for computing deductive closures have been developed in the database literature. There are three basic differences between the work presented here and optimization methods developed for deductive databases. First, databases are typically stored in large disk files and much of the work on optimization for deductive databases assumes that the deductive closure is to be computed using "set-based operations" that optimize the data access-time, that is, the time spent waiting for the disk head to reach the position on the disk where the data is held. Here we assume that data is stored in memory so that all data can be accessed efficiently. A second difference is that many of the deductive database optimizations involve rewriting the deductive rules into logically equivalent rules which are in some way more efficient. For example, rearranging the order of the antecedents of a rule can dramatically change the size of the intermediate relations representing the various prefixes of the antecedents. In the terminology used here, the number of prefix firings is sensitive to the order of the antecedents. Here we assume that the rules are being used to express an *algorithm* and that the algorithm designer is aware of the running time implication of the particular way the rules are written. Here we focus on providing the programmer with a clear conceptual model of the running time of a set of rules and then rely on the programmer to write efficient rules. A final difference is that much of the work in deductive databases concerns constructing indexes to make certain forms of data access more efficient. Here indexing is hidden in the proofs of run time theorems—the theorems provide the programmer with a model of running time which frees the programmer from having to think about indexing issues at all.

Bottom-up logic programming is also closely related to "memoing" or "tabling" for prolog programs [Tamaki and Sato 1986; Sagonas et al. 1994; Chen and Warren 1996]. In tabled prolog, inference rules are still run in a backward chaining manner but subgoals are placed in a "memoing table" so that, for example, a backward chaining interpretation of the transitivity rule is still guaranteed to terminate. It is well known that the tabled top-down execution of logic programs is closely related to the bottom-up execution of logic programs. The so-called magic-sets transformation can convert any given "top-down" logic program P into a logic program P' such that the bottom-up execution of P' simulates the tabled top-down execution of P [Bancilhon et al. 1986; Rohmer et al. 1986; Ullman 1989]. As in the case of deductive databases, the research on tabling methods for prolog execution has focused on methods for optimizing the execution of programs, while here we assume that programmers can write efficient programs provided that they have a clear model of execution time.

In addition to work in the general area of bottom-up logic programming, there is considerable work related to the particular example algorithms given in this paper. The relation between inference rules and parsing algorithms has been noted by a variety of researchers [Pereira and Warren 1983; Shieber et al. 1995; Rocio and Lopes 1998; Eisner and Satta 1999]. By providing a simple model of the running time of logic programs, the run time theorems presented here simplify the complexity analysis of a variety of parsing algorithms. The use of bottom-up logic

programming for program analysis has also been noted by several program analysis researchers [Ullman 1989; Reps 1994]. The contribution of this paper lies in the two run time theorems which provide a simple characterization of the running time of logic programs.

Two paradigms other than logic programming have achieved wide recognition as useful general frameworks for static analysis—abstract interpretation [Cousot and Cousot 1977] and set constraints [Aiken et al. 1994; Heintze and Jaffar 1990a, 1990b]. In all cases the frameworks are sufficiently flexible that it is often possible to view a single analysis, such as liveness analysis, within each of the frameworks, that is, as a special case of abstract interpretation, as a special case of set constraints, or as an algorithm expressed as a logic program. It does not seem possible to formally prove that one of these frameworks is superior to the others. As a Turing complete programming language, logic programs can in principle subsume any other programming formalism. But as a practical matter it is not immediately obvious what fraction of useful static analysis algorithms are best viewed as logic programs. This paper makes a case for bottom-up logic programs as a useful foundation for static analysis by presenting a series of examples.

3. *A First Run Time Theorem*

Formally, a bottom-up logic program is simply a set of inference rules where an inference rule is simply a first-order Horn clause, that is, a first-order formula of the form $A_1 \wedge \cdots \wedge A_n \rightarrow C$ where C and each A_i is a first-order atom, that is, a predicate applied to first-order terms (a first-order term is either a constant symbol, a first-order variable, or a function symbol applied to first-order terms). Here we consider only range-restricted rules, that is, rules in which every variable in the conclusion C appears in some antecedent A_i. We will use the term *assertion* to mean a ground atom, that is, an atom not containing variables, and use the term *database* to mean a set of assertions. For any set R of inference rules and any database D, we let $R(D)$ denote the set of assertions that can be proved from assertions in D using rules in R. This can be defined more formally with some additional terminology. A ground substitution is a mapping from a finite set of variables to ground terms. For any ground substitution σ defined on all the variables in an atom A, we let $\sigma(A)$ be the result of replacing each variable x in A by $\sigma(x)$. We say that a database E is closed under rule $A_1 \wedge \cdots \wedge A_n \rightarrow C$ if for any ground substitution σ defined on the variables in the rule, if $\sigma(A_1) \in E, \ldots, \sigma(A_n) \in E$ then $\sigma(C) \in E$. The output $R(D)$ can be defined as the least database containing D and closed under all rules in R. We view the set R as a program mapping input D to output $R(D)$.

An inference rule can be viewed as nested iterations. Consider the following:

$$P(y) \wedge Q(y, x) \wedge R(x) \rightarrow H(x, y). \tag{1}$$

Consider the case where the input is a database consisting only of assertions involving the predicates P, Q, and R. The output consists of the input plus all derivable applications of the predicate H. Intuitively, the rule iterates over assertions of the form $P(y)$ and, for each such assertion, iterates over the values of x such that $Q(y, x)$ holds and, for each such x, checks that $R(x)$ holds and, if so, asserts $H(x, y)$.

As the nested loop view might suggest, the order of the antecedents is important when viewing inferences rules as algorithms. For example, consider the following

rule which is logically equivalent to (1):

$$P(y) \wedge R(x) \wedge Q(y, x) \rightarrow H(x, y). \tag{2}$$

Rule (2) iterates over the assertions of the form $P(y)$ and then, for each such instance, iterates over all x such that $R(x)$ holds, and for each such x checks that $Q(y, x)$ holds. Now suppose there are n values of y satisfying $P(y)$ and also n values of x satisfying $R(x)$, but for any y there is at most one x satisfying $Q(y, x)$. In this case we might expect rule (1) to take $O(n)$ time and the logically equivalent rule (2) to take $O(n^2)$ time. If there were only one x such that $R(x)$ but for any y there were n values of x satisfying $Q(y, x)$ (and still n values of y satisfying $P(y)$), then (1) would take $O(n^2)$ time while rule (2) would take $O(n)$ time.

Note that for rule (2) the total number of iterations of the second loop equals the number of values of x and y such that $P(y)$ and $R(x)$ are given in the input. In general, any inference rule can be viewed as a set of nested loops where the number of iterations of the nth loop corresponds to the number of ways of instantiating the variables in the first n antecedents. This leads to the following general definition:

Definition. We define a *prefix firing* in database E to be a triple $\langle r, \sigma, i \rangle$ where r is a rule $A_1 \wedge \cdots \wedge A_n \rightarrow C$, $1 \leq i \leq n$, and where σ is a ground substitution defined on (only) the variables in $A_1, \ldots, A_i$ such that $\sigma(A_j) \in E$ for $1 \leq j \leq i$. We let $P_R(E)$ be the set of all prefix firings in E of rules in R.

Inference rules can be recursive—it is possible that a rule derives an assertion that leads to a new antecedent of that same rule. The algorithms in the introduction are all recursive in this sense. While it is natural to view nonrecursive rules as nested iterations, it is less obvious that this view is appropriate for recursive rules. The first run time theorem can be viewed as stating that the nested iteration view applies to recursive rules as well. Recall that a rule is range-restricted if every variable in the conclusion appears in some antecedent. All rules discussed in this paper are range-restricted.

THEOREM 1. *For any range-restricted rule set R there exists an algorithm for mapping any finite D to $R(D)$ which runs in time $O(|D| + |P_R(R(D))|)$ assuming unit time hash table operations.*

The theorem allows for the possibility that the rules do not terminate, that is, $R(D)$ is infinite. For range-restricted rules, the only way $R(D)$ can be infinite is if $P_R(R(D))$ is also infinite. So if $R(D)$ is infinite, the theorem holds vacuously. The more interesting case is when $R(D)$ is finite. Note that by counting prefix firings, rather than just full firings, the run time theorem captures the difference in efficiency between rules (1) and (2) above.

Before proving Theorem 1 we show how it can be used to establish the running time of some particular logic program algorithms. Consider the transitive closure algorithm defined by the inference rules $\text{EDGE}(x, y) \rightarrow \text{PATH}(x, y)$ and $\text{EDGE}(x, y) \wedge \text{PATH}(y, z) \rightarrow \text{PATH}(x, z)$. Suppose R consists of these two rules and D consists of e assertions of the form $\text{EDGE}(c, d)$ involving n constants. There are e (prefix) firings of the first rule. For the second rule there are e prefix firings for the first antecedent, and for each such firing there are at most n firings of the of the next antecedent. So the total number of prefix firings is $O(en)$. Theorem 1 now implies that the algorithm runs in time $O(en)$.

As another example, consider the CKY parsing algorithm. In the following formulation we assume that the input has been augmented with assertions of the form SUCC$(i, i+1)$ for each $1 \leq i \leq n-1$ where n is the length of the input string. Logic programming and the run time theorem can be extended to handle arithmetic, although we will not formally consider arithmetic here.

$$\frac{X \Rightarrow y \qquad s(i) = y}{X \Rightarrow s(i,i)} \qquad \frac{X \Rightarrow YZ \qquad Y \Rightarrow s(i,j) \qquad \text{SUCC}(j, j') \qquad Z \Rightarrow s(j',k)}{X \Rightarrow s(i,k)}$$

Let R be the above set of two rules, let G be a grammar in Chomsky normal form, and let S be an input string of length n. Let $D(G, S)$ consist of the assertions of the form $A \Rightarrow BC$ and $A \Rightarrow a$ in G plus the assertions $s(i) = a$ and SUCC$(i, i+1)$ for $1 \leq i \leq n$ corresponding to the input S. We have that $R(D(G, S))$ consists of $D(G, S)$ plus a set of assertions of the form $A \Rightarrow s(i, j)$ with A a nonterminal in G and $i, j \in [1, n]$. To determine the running time of this algorithm it suffices to bound the number of prefix firings. Consider the left-hand rule. There are at most $|G|$ ways of instantiating the first antecedent. Each such instantiation fixes the value of y, and there are then at most n ways of continuing with an instantiation of i. So there are $O(|G|n)$ prefix firings of the left-hand rule. Now consider the right-hand rule. Again there at most $|G|$ ways of instantiating the first antecedent. An instantiation of the first antecedent fixes the values of X, Y, and Z. Given an instantiation of Y, there are at most n^2 ways of instantiating i and j. An instantiation of j determines the instantiation of j'. Finally, there are at most n possible instantiations of k, and hence the total number of prefix firings is $O(|G|n^3)$.

Theorem 1 is proved in two stages. First the original program is transformed to a simpler program with only a constant factor expansion in the number of prefix firings. This simpler program consists of rules with only a single antecedent, plus rules of the form $P(x, y) \wedge Q(y, z) \rightarrow C$, where x, y, and z are variables. After performaning this transformation, an algorithm is given for computing deductive closures of rule sets in this restricted format.

We first consider the transformation of an arbitrary program into a program in the restricted form. If r is a rule $A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow C$ then we define the binarization $B(r)$ to be the following set of rules where $P_1, P_2, \ldots, P_n$ are fresh predicate symbols and $x_1, \ldots, x_{k_i}$ are the variables occurring in the first i antecedents. The predicate P_i represents the relation defined by the first i antecedents.

$$\begin{aligned} A_1 &\rightarrow P_1(x_1, \ldots, x_{k_1}), \\ P_1(x_1, \ldots, x_{k_1}) \wedge A_2 &\rightarrow P_2(x_1, \ldots, x_{k_2}), \\ &\vdots \\ P_{n-1}(x_1, \ldots, x_{k_{n-1}}) \wedge A_n &\rightarrow P_n(x_1, \ldots, x_{k_n}), \\ P_n(x_1, \ldots, x_{k_n}) &\rightarrow C. \end{aligned}$$

For a rule set R we define $B(R)$ to be the union of the sets $B(r)$ for $r \in R$. We assume that the predicate symbols introduced by transformations form a distinct class of symbols and we let $\pi(E)$ denote the subset of E not involving symbols introduced

by transformations. The following lemma states the semantic correctness of the binarization transformation:

LEMMA 2. *If $\pi(D) = D$, that is, the input does not use the "fresh" predicates, then $R(D) = \pi(B(R)(D))$.*

The proof of the above lemma can be done by two inductions on the length of logic program derivations—the first showing $R(D) \subseteq \pi(B(R)(D))$ and the second showing $\pi(B(R)(D)) \subseteq R(D)$. The details are omitted here.

A more interesting property of the binarization transformation is that it preserves the number of prefix firings up to a multiplicative factor. More specifically, we have the following:

LEMMA 3. *If $\pi(D) = D$ then we have the following:*

$$|P_{B(R)}(B(R)(D))| = 2|P_R(R(D))|.$$

PROOF. The assertions of the form $P_i(x_1, \ldots, x_{k_i})$ are in one-to-one correspondence with $P_R(R(D))$. For each assertion $P_i(x_1, \ldots, x_{k_i})$ there are exactly two prefix firings of $B(R)$—the firing of all antecedents in the rule that generates $P_i(x_1, \ldots, x_{k_i})$ and the prefix firing of the first antecedent when this assertion is used as an antecedent. All prefix firings in $B(R)$ are either generations of, or uses of, some assertion of the form $P_i(x_1, \ldots, x_{k_i})$. Hence there are exactly twice as many prefix firings of $B(R)$ as there are of R. □

Lemmas 2 and 3 imply that without loss of generality we can assume that all rules in R contain at most two antecedents. Now assuming that R is binary in this sense, we define an "indexing transformation" as follows. For any rule r with two antecedents $A_1 \wedge A_2 \rightarrow C$ we define $I(r)$ to be the following set of rules where $x_1, \ldots, x_n$ are all variables occurring in A_1 but not A_2, $y_1, \ldots, y_m$ are all variables that occur in both A_1 and A_2, and $z_1, \ldots, z_k$ are all variables that occur in A_2 but not A_1. The predicates P_1, P_2, and Q, and the function symbols f, g, and h are all fresh.

$$\begin{aligned} A_1 &\rightarrow P_1(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)), \\ A_2 &\rightarrow P_2(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)), \end{aligned}$$

$$P_1(x, y) \wedge P_2(y, z) \rightarrow Q(x, y, z),$$

$$Q(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)) \rightarrow C.$$

For a rule set R in which no rule has more than two antecedents, we define $I(R)$ to consist of all single-antecedent rules in R plus the union of all rule sets $I(r)$ where r is a two antecedent rule in R. We first have the following correctness lemma, whose proof we omit:

LEMMA 4. *If $\pi(D) = D$ and all rules in R have at most two antecedents, then $R(D) = \pi(I(R)(D))$.*

More significantly, we also have the following:

LEMMA 5. *If $\pi(D) = D$, and R is range-restricted, then then we have the following:*

$$\left|P_{I(R)}(I(R)(D))\right| \leq 2|R||D| + (2|R| + 3)|P_R(R(D))|.$$

Algorithm to Compute $R(D)$:

Initialize E to be the empty set. Mark every element of D and initialize the queue Q to contain D.

While Q is not empty:

1. Remove an element Φ from Q.
2. Add Φ to E.
3. For each single-antecedent rule $A \rightarrow C$ in R determine whether there is a substitution σ such that $\sigma(A) = \Phi$. If so, assert $\sigma(C)$ as described below.
4. For each two-antecedent rule $P_1(x, y) \wedge P_2(y, z) \rightarrow C$ do the following:
   4a. If Φ has the form $P_1(t_1, t_2)$ then for each t_3 such that E contains $P_2(t_2, t_3)$ assert $\sigma(C)$ where σ maps x to t_1, y to t_2, and z to t_3.
   4b. If Φ has the form $P_2(t_2, t_3)$ then for each t_1 such that E contains $P_1(t_1, t_2)$ assert $\sigma(C)$ where σ is defined as in 4a.

Procedure for Asserting Ψ:

1. If Ψ is already marked do nothing.
2. Otherwise, mark Ψ and add Ψ to Q.

FIG. 1. The algorithm underlying Theorem 1.

PROOF. Each prefix firing of $I(R)$ is either a firing of a single-antecedent rule, and hence is also a firing of a rule in R, or is a firing of a rule of the form given above in the definition of the transformation I. There can be at most $2|R||R(D)|$ firings of the rules that generate assertions of the form $P_1(x, y)$ and $P_2(y, z)$. The rules that generate $Q(x, y, z)$ have two antecedents. A firing of the first antecedent corresponds to a firing of the first antecedent in the original rule in R and a firing of both antecedents corresponds to a firing of both antecedents in the original rule in R. Hence there can be at most $|P_R(R(D))|$ prefix firings of the rules that generate the assertions $Q(x, y, z)$. Finally, each firing of a rule that uses an assertion of the form $Q(x, y, z)$ as an antecedent corresponds to a firing of an original rule. Hence the total number of prefix firings can not be larger than $3|P_R(R(D))| + 2|R||R(D)|$. For range-restricted rules, we have that $|R(D)| \leq |D| + |P_R(R(D))|$, which now yields the lemma. □

Lemmas 4 and 5 now allow us to assume without loss of generality that R consists of single-antecedent rules plus rules of the form $P_1(x, y) \wedge P_2(y, z) \rightarrow C$. Under these assumptions, we can use the algorithm shown in Figure 1 to compute $R(D)$. Theorem 1 now follows from the following two lemmas:

LEMMA 6. *If R is range-restricted and $R(D)$ is finite then the algorithm terminates with E equal to $R(D)$.*

PROOF. Since R is range-restricted, and D contains only ground atoms, the algorithm never asserts an open atom, that is, one containing variables. The algorithm maintains the invariant that all assertions in E or on Q are in $R(D)$. Since the algorithm never places the same assertion on Q twice, if $R(D)$ is finite then the algorithm must terminate. When the algorithm terminates, the final value of E must be a subset of $R(D)$. The algorithm also maintains the invariant that every atom in D or derivable in one step from E is either in E or on Q. This implies that when Q is empty E contains D plus all derivable assertions. Hence, when the algorithm terminates, E contains $R(D)$. □

LEMMA 7. *If R is range-restricted, and $R(D)$ is finite, then the algorithm can be run to completion in $O(|D| + |P_R(R(D))|)$ time assuming unit time hash table operations.*

PROOF. Throughout the proof we assume that all terms and atoms are interned—the same expression is represented by a data structure at the same location in memory—and that equality testing can be done in unit time by checking pointer equality. This allows one to determine whether a given ground term matches a given "pattern" term, possibly with repeated variables in the pattern term, in time proportional to the size of the pattern term. Interning also supports unit time marking and checking for the presence of marks. We assume constant time hash table operations so that applying a substitution to a pattern term can be done in time proportional to the size of the pattern term.

The initialization step takes time proportional to $|D|$. For range-restricted rules we have $|R(D)| \leq |D| + |P_R(R(D))|$, so it suffices to show that the running time is $O(|R(D)| + |P_R(R(D))|)$. There is one execution of steps 1 and 2 for each element of $R(D)$, and each execution takes unit time. Step 3 involves an iteration over rules in R. For a given rule $A \rightarrow C$ and a given ground atom Ψ, one must determine if there exists a σ such that $\Psi = \sigma(A)$. As mentioned above, this can be done in time proportional to the size of the atom A. If such a σ exists, it can be computed in time proportional to the size of A. The size of A is a constant determined by the rule set and independent of $|D|$ or $|P_R(R(D))|$. As mentioned above, under the assumptions used here computing $\sigma(C)$ takes time proportional to the size of C and hence is also $O(1)$. The time spent in a single call to the assert procedure is also $O(1)$. Hence the time to process a given rule in step 3 is $O(1)$. The time spent iterating over the rules in step 3 is also $O(1)$. So the total time spent in step 3 is $O(|R(D)|)$. By a similar argument, the time in step 4 outside of inner loops in 4a and 4b is also $O(|R(D)|)$. Finally, we must consider the inner loops in steps 4a and 4b. We assume that for each term t and predicate P used in an antecedent of a binary rule we maintain a list of all the terms t' such that E contains $P(t, t')$. This list must be extended each time a new assertion of the form $P(t, t')$ is added to E. The total time spent building these lists is $O(|R(D)|)$. There is an analogous list for each term t and predicate P of the terms t' such that E conatins $P(t', t)$. Given these lists, the inner loops in 4a and 4b can each be executed in time proportional to the number of iterations. It now suffices to show that the total number of iterations of the inner loops in 4a and 4b is $O(|P_R(R(D))|)$. It suffices to show that each of these loops only considers a given triple $\langle t_1, t_2, t_3 \rangle$ once. When such a triple is considered in step 4a, the assertion $P_1(t_1, t_2)$ must equal Φ. Hence this triple cannot be visited again in a later invocation of step 4a. A similar statement applies to 4b. □

4. *Liveness Analysis*

We now turn to applications of run time theorems in static analysis. Our first example is a very simple static analysis—liveness analysis. As mentioned in the introduction, most compilers rely on the ability to determine that the value of a given variable is no longer needed so that a register being used to store the variable can now be used for other purposes [Aho and Ullman 1986]. To present a simple example of liveness analysis, we first define a simple programming language. We take a program to be a sequence of instructions where each instruction has one of the following forms where x, y, and z are variables, op is an operation,

L1 $l : x = \text{op}(y, z)$

$\text{LIVE}(y, l), \text{LIVE}(z, l)$

L2 $l : x = \text{op}(y, z)$
$\text{SUCC}(l, l')$
$\text{LIVE}(w, l')$
$\text{DISTINCT}(w, x)$

$\text{LIVE}(w, l)$

L3 $l : \text{goto} l'$
$\text{LIVE}(w, l')$

$\text{LIVE}(w, l)$

L4 $l : \text{if} x \text{goto} l'$
$\text{LIVE}(w, l')$

$\text{LIVE}(W, L)$

L5 $l : \text{if} x \text{goto} l'$
$\text{SUCC}(l, l'')$
$\text{LIVE}(w, l'')$

$\text{LIVE}(w, l)$

FIG. 2. A liveness analysis algorithm.

for example, addition, multiplication, or boolean comparison, and l_i and l_j are instruction labels—a number unique to the labeled instruction:

$$l_i\colon\ x = \text{op}(y, z);$$
$$l_i\colon\ \text{if } x \text{ goto } l_k;$$
$$l_i\colon\ \text{goto } l_k;$$
$$l_i\colon\ \text{halt}.$$

We assume a successor relation on labels—each label that labels an instruction other than a halt instruction has a successor label which is the next instruction to be executed. A program state is a pair $\langle l, \sigma \rangle$ where l is the instruction label of the next instruction to be executed and σ is a "store" mapping variables to values. A single step of computation converts a given program state into the next program state. For example, if l labels the instruction $x = +(y, z)$ then a single execution step converts the the state $\langle l, \sigma \rangle$ to the successor state $\langle l', \sigma' \rangle$ where l' is the successor label of l and σ' is identical to σ except that $\sigma'(x)$ is $\sigma(y) + \sigma(z)$. We say that an instruction of the form $x = \text{op}(y, z)$ writes x and reads y and z. We say that a variable x is *live* in state $\langle l, \sigma \rangle$ if the computation starting in that state reads x without having written x in an earlier instruction. For example, if l labels the instruction $x = +(x, y)$ then x is live at $\langle l, \sigma \rangle$ because it is about to be read. If l labels $x = +(y, z)$ and $\langle l, \sigma \rangle$ has successor state $\langle l', \sigma' \rangle$, and a variable w different from x is live at $\langle l', \sigma' \rangle$, then w is live at $\langle l, \sigma \rangle$.

It is undecidable to determine whether x is live at $\langle l, \sigma \rangle$—in the general case this would require determining if a given loop halts, which is equivalent to deciding the halting problem. A static analysis generally computes a conservative approximation to an undecidable problem. An algorithm for liveness analysis is defined by the rules shown in Figure 2. The rules are conservative in the sense that, for any state $\langle l, \sigma \rangle$ and variable x, if x is live at $\langle l, \sigma \rangle$ then the rules derive $\text{LIVE}(x, l)$. This can be proved by induction on the number of steps of computation it takes for the computation starting at $\langle l, \sigma \rangle$ to read x. This implies that if the rules *do not* derive $\text{LIVE}(x, l)$ then x is not live at any state of the form $\langle l, \sigma \rangle$. So if the rules do not derive $\text{LIVE}(x, l)$ then the compiler can reuse the register storing the value of x when it reaches program label l.

The rules assume that for each pair of distinct variables x and y the database contains the assertion DISTINCT(x, y). In practice the predicate DISTINCT can be computed on demand rather than stored in the input database. One can prove that Theorem 1 holds with computed predicates in rule antecedents provided that two conditions are satisfied. First, antecedents involving computed predicates must be of the form $P(x_1, \ldots, x_n)$, where each x_i is a variable. Second, in any computed-predicate antecedent, either all x_i occur in some earlier antecedent (as is the case in Figure 2) or the bindings for the variables not occurring in earlier antecedents can be computed in time proportional to the number of such bindings.

We now analyze the running time of the algorithm given in Figure 2. Let N be the number of instructions in the program and let V be the number of variables. Since all derived assertions are of the form LIVE(x, l), we have that $|R(D)|$ is $O(NV)$. Rule L1 is actually an abbreviation for two rules—one concluding LIVE(y, l) and one concluding LIVE(z, l). These rules each have at most N prefix firings. Now consider rule L2. The first antecedent determines all bindings other than w. There are at most N ways of instantiating the first antecedent and at most V ways of instantiating w so we get $O(NV)$ prefix firings. A similar analysis holds for rules L3, L4, and L5. So the algorithms runs in $O(NV)$ time.

Actually a tighter analysis is possible. Let L be the total number of assertions of the form LIVE(x, l) contained in $R(D)$. Let $\overline{V}$ be L/N. Intuitively, $\overline{V}$ is the average over all instructions of the number of live variables at that instruction. It is possible to show that the algorithm actually runs in time $O(N + N\overline{V})$. In practice $\overline{V}$ remains bounded even for very large programs and so, in practice, the analysis runs in time linear in the size of the program. To see that the algorithm runs in time $O(N + N\overline{V})$ note that $N + N\overline{V}$ equals $N + L$. To show that this bound holds, it suffices to divide the prefix firings into two sets, one of which has size $O(N)$ and one of which has size $O(L)$. There are only $O(N)$ prefix firings of L1. We divide the prefix firings of L2 into those in which w is x and those in which w and x are distinct. There are $O(N)$ prefix firings of the first type. Each prefix firing of the second type generates a distinct assertion of the form LIVE(w, l). So the number of prefix firings of this sort can be no larger than the number of assertions of the form LIVE(w, l) which is, by definition, the quantity L. Hence there are only $O(L)$ prefix firings of the second type. By a similar argument, each firing of the rules L3, L4, and L5 generates a distinct conclusion, and hence the number of firings of each of these rules is $O(L)$.

5. *Data Flow Analysis*

Some programming languages, such as Common Lisp and Scheme, use type tags on data values and generate graceful run time exceptions if a run-time type violation occurs, for example, if an attempt is made to extract a slot from a nonstructure. Such languages do not use static-type checking but are still guaranteed never to segment-fault. In some cases it is possible for the compiler to statically determine that a particular pointer variable is guaranteed to be a structure of a certain type [Shivers 1991]. In that case the run time safety check can be omitted from the compiled code. Data flow analysis provides one way of determining that a variable is guaranteed to be a structure of a certain type. More generally, data flow analysis intuitively determines what kind of values a given variable can have at a given program point. Data flow analysis has a variety of applications in compilers

D1 $\dfrac{x = \mathrm{k}}{x \Rightarrow \mathrm{INT}}$

D2 $\dfrac{x = \langle y, z\rangle}{x \Rightarrow \langle y, z\rangle}$

D3 $\dfrac{y = \Pi_j(x) \quad x \Rightarrow \langle z_1, z_2\rangle}{y \Rightarrow z_j}$

D4 $\dfrac{u \Rightarrow w, \quad w \Rightarrow v}{u \Rightarrow v}$

FIG. 3. A data flow analysis algorithm. The rule involving Π_j is an abbreviation for two rules—one with Π_1 and one with Π_2.

[Aho and Ullman 1986; Appel 1997]. As in most static analyses, data flow analysis is a conservative approximation to an undecidable problem.

Here we formulate data flow in a simple abstract setting. We extract from the program the assignment statements of the form $x = e$. Here we consider only assignments of the form $x = k$, $x = \langle y, z\rangle$, $x = \Pi_1(y)$, and $x = \Pi_2(y)$, where k is an integer constant, $\langle x, y\rangle$ is the abstract pair of x and y, $\Pi_1(x)$ is the first component of the pair x, and $\Pi_2(x)$ is the second component of the pair x. We take a store to be a mapping from a finite subset of the variables to values where a value is either an integer or a pair of values. In many programming languages (e.g., Scheme), it is syntactically impossible to write a program that uses a variable before assigning it some initial value. In such languages an assighment $x = e$ is guaranteed not to be executed until all variables in e have values. An assignment $x = e$ will be called *executable in store* σ if σ assigns a value to all variables in e and e is not of the form $\Pi_1(x)$ or $\Pi_2(x)$ where $\sigma(x)$ is not a pair. If $x = e$ is executable in σ then e has a well-defined value in σ which we denote as $\sigma(e)$. A set of assignment statements define a nondeterministic transition relation on stores. We say that σ' is a possible successor of σ if there is an executable assignment $x = e$ such that σ' is identical to σ except that $\sigma'(x) = \sigma(e)$. We say that σ' is *reachable from* σ if either it is σ or there is a possible successor σ'' of σ such that σ' is reachable from σ''. A store is called *reachable* if it is reachable from the empty store (the store that does not assign any values to any variables). We are interested in the set of values assigned to x in reachable stores. If the value of x is guaranteed to be a pair, then the run time safety test can be omitted from the compilation of an instruction of the form $y = \Pi_1(x)$.

Figure 3 gives a simple data flow analysis algorithm. The analysis algorithm generates assertions of the form $x \Rightarrow e$, where x is a program variable and e is either INT (as in rule D1), an expression $\langle y, z\rangle$ that occurs somewhere in the right-hand side of some assignment statement (as in rule D2), or a program variable (as in rule D3). If there are N input assignment statements then there are only $O(N^2)$ possible assertions of the form x $\Rightarrow$ e.

The derivable assertions of the form $x \Rightarrow$ INT and $x \Rightarrow \langle y, z\rangle$ should be viewed as defining a grammar for generating values. We write $x \Rightarrow^* v$ to mean either that v is an integer and $x \Rightarrow$ INT is generated by the rules, or v is a pair $\langle u, w\rangle$ where the rules derive $x \Rightarrow \langle y, z\rangle$ and we have $y \Rightarrow^* u$ and $w \Rightarrow^* w$. We now prove that if a store σ is reachable (in the sense defined above) then for any x assigned a value by σ we have that $x \Rightarrow^* \sigma(x)$. The proof is by induction on the number of assignments needed to reach σ starting from the empty store. The result is immediate for the empty store. Now assume the result for σ and let σ' be the

result of executing $x = k$. We need to show the result for $\sigma'(y)$ for all y on which σ' is defined. If y is x then the result follows by rule D1. If y is not x, the result follows by the induction hypothesis. A similar analysis holds when σ' is generated by an execution of $x = \langle y, z \rangle$ where the argument relies on the existence of rule D2. In the case where σ' is generated by $y = \Pi_i(x)$, the argument involves a combination of rules D3 and D4. Note that if the rules fail to generate $x \Rightarrow \text{INT}$ then x must be a pair and run time checks can be omitted from the compilation of $y = \Pi_i(x)$.

By counting prefix firings, one can show that the running time of the algorithm in Figure 3 is dominated by the number of prefix firings of D4, which is $O(N^3)$. It is possible to show that determining whether $x \Rightarrow \text{INT}$ is derivable from a given set of assignments using the rules in Figure 3 is 2NPDA complete [Heintze and McAllester 1997b; Melski and Reps 1997]. 2NPDA is the class of languages recognizable by a two-way nondeterministic pushdown automaton. A language $\mathcal{L}$ will be called 2NPDA-hard if any problem in 2NPDA can be reduced to $\mathcal{L}$ in n polylog n time. We say that a problem can be solved in *subcubic time* if it can be solved in $O(n^k)$ time for $k < 3$. If a 2NPDA-hard problem can be solved in subcubic time then all problems in 2NPDA can be solved in subcubic time. The data flow problem is 2NPDA-complete in the sense that it is in the class 2NPDA and is 2NPDA-hard. No subcubic procedure is known for any 2NPDA-complete problem.

Cubic time is impractical for many applications. However, if we only consider programs in which the assignment statements are well typed using types of a bounded size, then a more efficient algorithm is possible [Heintze and McAllester 1997a]. This more efficient algorithm can also be stated and analyzed as a set of inference rules, although we will not do so here.

6. *Flow Analysis in the Lambda Calculus*

As a final example of an application of Theorem 1, we consider flow analysis in the lambda calculus with pairing. This flow analysis includes control flow analyses, that is, determining what program points a given jump instruction can branch to. A jump instruction corresponds to a procedure call in the lambda calculus. Control flow analysis in the lambda calculus corresponds to determining what procedures are possible values of a given procedure variable. The flow analysis algorithm for the lambda calculus given here is very similar to the data flow analysis given in the previous section. However, lambda calculus formulations are common in the literature, and it seems important to give an example of a static analsysis phrased directly as inference rules on the lambda calculus.

The lambda calculus can be viewed as an abstract functional programming language where a program is a term and executing the program corresponds to computing the value of a term. The terms of the pure lambda calculus with pairing are defined by the following grammar:

$$e ::= x \mid \langle e_1, e_2 \rangle \mid \Pi_1(e) \mid \Pi_2(e) \mid (e_1, e_2) \mid \lambda x.e.$$

We define the operational semantics of the lambda calculus in Figure 4. The semantics is itself written as a bottom-up logic program evaluator. The evaluation rules manipulate assertions of the form compute(e, σ) and $\langle e, \sigma \rangle \Rightarrow^* v$. Intuitively, the assertion compute(e, σ) states that the evaluator should compute the value of term e under the variables bindings given by σ. The assertion $\langle e, \sigma \rangle \Rightarrow^* v$ states that v is the resulting value. The initial database consists of a single assertion

E1 $\dfrac{\text{compute}((f\,w),\sigma)}{\text{compute}(f,\sigma),\quad \text{compute}(w,\sigma)}$

E2 $\dfrac{\text{compute}(\lambda x.e,\sigma)}{\langle \lambda x.e,\sigma\rangle \Rightarrow^* \langle \lambda x.e,\sigma\rangle}$

E3 $\dfrac{\text{compute}(x,\sigma)}{\langle x,\sigma\rangle \Rightarrow^* \sigma(x)}$

E4 $\dfrac{\text{compute}((f\,w),\sigma)\quad \langle f,\sigma\rangle \Rightarrow^* \langle \lambda x.e,\sigma'\rangle\quad \langle w,\sigma\rangle \Rightarrow^* v}{\langle (f\,w),\sigma\rangle \Rightarrow \langle e,\sigma'[x := v]\rangle}$

E5 $\dfrac{p \Rightarrow q}{\text{compute}(q)}$

E6 $\dfrac{p \Rightarrow q\quad q \Rightarrow^* v}{p \Rightarrow^* v}$

E7 $\dfrac{\text{compute}(\langle e_1,e_2\rangle,\sigma)}{\text{compute}(e_1,\sigma),\quad \text{compute}(e_2,\sigma)}$

E8 $\dfrac{\text{compute}(\langle e_1,e_2\rangle,\sigma)\quad \langle e_1,\sigma\rangle \Rightarrow^* v_1\quad \langle e_2,\sigma\rangle \Rightarrow^* v_2}{\langle\langle e_1,e_2\rangle,\sigma\rangle \Rightarrow^* \langle v_1,v_2\rangle}$

E9 $\dfrac{\text{compute}(\Pi_j(u),\sigma)}{\text{compute}(u,\sigma)}$

E10 $\dfrac{\text{compute}(\Pi_j(u),\sigma)\quad \langle u,\sigma\rangle \Rightarrow^* \langle v_1,v_2\rangle}{\langle \Pi_j(u),\sigma\rangle \Rightarrow^* v_j}$

FIG. 4. An algorithm for evaluating lambda terms.

of the form compute$(e,\emptyset)$, where e is a closed term and $\emptyset$ is the empty binding environment. Rules E4 and E5 derive other assertions of the form compute(w,σ). The expression $\sigma(x)$ in the conclusion of E3 represents the term derived by applying the subsitution σ to the term x (with renaming of bound variables in x to avoid the capture of free variables in σ). The expression $\sigma'[x := v]$ in the conclusion of rule E4 represents the substitution that is identical to σ' except that it maps x to the value v. Note that the rules maintain the invariant that in all derivable assertions of the form compute(w,σ) we have that w is a subterm of the original top-level term. We can think of the term w as the program counter and the σ as the program store.

Figure 5 gives an algorithm for both control and data flow analysis for the λ-calculus with pairing. The rules are numbered so as to suggest alignment with the rules in Figure 4. The input to the analysis is a single assertion of the form compute(e), where e is a closed term. Rules F1, F2, F7, and F9 derive all assertions of the form compute(w), where w is a subterm of e. The rules also derive assertions of the form $e \Rightarrow w$ and $e \Rightarrow^* w$, where e and w are subterms of the input. All assertions of the form $e \Rightarrow^* v$ have the property that the "value" v is either a lambda expression or a pairing expression.

To verify that the analysis in Figure 5 is conservative, that is, to establish its correctness, we view each assertion of the form $e \Rightarrow^* w$ as a production in a grammar for generating values. To maintain consistency with Figure 4, we define a value to be either a pair $\langle \lambda x.e,\sigma\rangle$, where σ maps the free variables of $\lambda x.e$ to

F1 $\dfrac{\text{compute}((f\,w))}{\text{compute}(f),\ \ \text{compute}(w)}$

F2 $\dfrac{\text{compute}(\lambda x.e)}{\lambda x.e \Rightarrow^* \lambda x.e,\ \text{compute}(e)}$

F4 $\dfrac{\text{compute}((f\,w)),\ f \Rightarrow^* \lambda x.u}{x \Rightarrow w,\ \ (f\,w) \Rightarrow u}$

F6 $\dfrac{u \Rightarrow w,\ \ w \Rightarrow^* v}{u \Rightarrow^* v}$

F7 $\dfrac{\text{compute}(\langle e_1, e_2\rangle)}{\text{compute}(e_1),\ \ \text{compute}(e_2),\ \langle e_1, e_2\rangle \Rightarrow^* \langle e_1, e_2\rangle}$

F9 $\dfrac{\text{compute}(\Pi_j(u))}{\text{compute}(u)}$

F10 $\dfrac{\text{compute}(\Pi_j(u)),\ u \Rightarrow^* \langle e_1, e_2\rangle}{\Pi_j(u) \Rightarrow e_j}$

FIG. 5. Flow analysis for the lambda calculus with pairing.

values, or a pair of values. Note that the base case is given by closed lambda expressions and empty substitutions. The rules in Figure 4 generate assertions of the form compute(e, σ), where e is a subterm of the input term and σ maps variables to values, plus assertions of the form $\langle e, \sigma\rangle \Rightarrow^* v$, where v is a value. We now formally treat the output of Figure 5 as defining a grammar. For any subterm e of the input term and value v, we define $e \rightharpoonup^* v$ to mean that either $e \Rightarrow^* \lambda x.u$ and v is $\langle \lambda x.u, \sigma\rangle$, where σ is a substitution satisfying $y \rightharpoonup^* \sigma(y)$ for all y in the domain of σ, or v is a pair $\langle v_1, v_2\rangle$ such that $e \Rightarrow^* \langle w_1, w_2\rangle$ with $w_1 \rightharpoonup^* v_1$ and $w_2 \rightharpoonup^* v_2$. The rules in Figure 5 are conservative in the sense that if Figure 4 generates $\langle e, \sigma\rangle \Rightarrow^* v$ then Figure 5 generates a grammar yielding $e \rightharpoonup^* v$. The proof is by computational induction on the inference rules in Figure 4 and is omitted here. Note, however, that if, for a given subterm e, Figure 5 does not generate any assertion of the form $e \Rightarrow^* \lambda x.e$, then it follows that all values of e are pairs and run time safety checks in the compilation of $\Pi_j(e)$ can be omitted. By counting prefix firings in the rules in Figure 5, we get that the running time of this analysis is $O(N^3)$, where N is the number of subterms of the input term.

The analysis defined in Figure 5 can be viewed as a form of set based analysis [Heintze 1994; Aiken et al. 1994]. The rules can also be used to determine if the given term is typable by recursive types with function, pairing, and union types [McAllester 1996] using arguments similar to those relating control flow analysis to partial types [Kozen et al. 1994; Palsberg and O'Keefe 1995]. It is possible to give a subtransitive flow algorithm which runs in linear time under the assumption that the input expression is well typed and that every type expression has bounded size [Heintze and McAllester 1997a]. The subtransitive analysis algorithm can also be presented as a bottom-up logic program whose running time can be analyzed using Theorem 1.

7. *A Union-Find Run Time Theorem*

A variety of program analysis algorithms exploit equality. Perhaps the most fundamental use of equality in program analysis is the use of unification in type inference for simple types. Other examples include the nearly linear time flow analysis

algorithm of Bondorf and Jorgensen [1993], the quadratic type inference algorithm for an Abadi-Cardelli object calculus given by Henglein [1999], and the improvement in empirical performance due to equality reported by Fähndrich et al. [1998]. Here we formulate a general approach to the incorporation of union-find methods into algorithms defined by bottom-up inference rules. In this section we give a general run time theorem for such union-find rule sets.

We let UNION, FIND, and FLINK be three distinguished binary predicate symbols. The predicate UNION can appear in rule conclusions but not in rule antecedents. The predicates FIND and FLINK can appear in rule antecedents but not in rule conclusions. A rule set satisfying these conventions will be called a *union-find rule set*. Intuitively, an assertion of the form UNION(u, w) in the conclusion of a rule means that u and w should be made equivalent. The UNION assertions cause assertions of the form FLINK(u, w) and FIND(u, w) to be added to the database. The mechanism described below maintains the invariant that for any given u there is at most one w such that the database contains FLINK(u, w). Furthermore, the relation FLINK is acyclic. Hence this relation forms a tree and, for any given node u, we can follow the FLINK relation from that node until we reach a node that has no FLINK successor. This terminal node is the find value of u. Two nodes are considered equivalent if they have the same find value. The predicate FIND is maintained as the transitive closure of the predicate FLINK; intuitively the relation FIND contains all possible path-compressions of the find data tree. Note that two terms u and w are considered equivalent if and only if there exists an f such that the database contains both FIND(u, f) and FIND(v, f). Of course in practice one should erase obsolete FIND assertions so that for any term s there is at most one assertion of the form FIND(s, f). However, because FIND assertions can generate conclusions before they are erased, it seems difficult to formulate the erasure process so as to improve the statement of Theorem 8 below.

When an assertion of the form MERGE(u, w) is added to the database and u and w are already equivalent, no modification is made to the relations FLINK or FIND. If u and v are not equivalent then a new FLINK assertion is added from the find of the smaller equivalence class to the find of the larger equivalence class and the relation FIND is updated to again be the transitive closure of FLINK. If the two classes are the same size then the FLINK arc goes from the find of u (the first argument to merge) to the find of v (the second argument). This tie-breaking convention can have algorithmic significance as discussed in later sections.

We define a clean database to be one not containing FLINK or FIND assertions. Given a union-find rule set R and a clean database D, we say that a database E is an R-closure of D if E can be derived from D by repeatedly applying rules in R—including rules that result in union operations—and no further application of a rules in R changes E. Unlike the case of traditional inference rules, a union-find rule set can have many possible closures—the set of derived assertions depends on the order in which the rules are used. For example, if we derive the three union operations UNION(u, w), UNION(s, w), and UNION(u, s) then the FLINK relation will contain only two arcs and the choice of the two arcs depends on the order in which the union operations are done. If rules are used to derive other assertions from the FLINK assertions then arbitrary relations can depend on the order of inference. For most algorithms, however, the correctness analysis and running time analysis can be done independently of the order in which the rules are run. We now present a general run time theorem for union-find rule sets:

THEOREM 8. *For any range-restricted union-find rule set R, there exists an algorithm mapping D to an R-closure of D, denoted as $R(D)$, that runs in time $O(|D| + |P_R(R(D))| + |F(R(D))|)$, where $F(R(D))$ is the set of* FIND *assertions in $R(D)$. Furthermore, $|F(R(D))| \leq N\lceil \log_2 N \rceil$, where N is the number of distinct terms that appear as an argument of a* UNION *assertion in $R(D)$.*

The proof is essentially identical to the proof of Theorem 1. The same source-to-source transformation is applied to R to show that without loss of generality we need only consider single antecedent rules plus rules of the form $P(x, y) \wedge Q(y, z) \rightarrow R(x, y, z)$, where x, y, and z are variables and P, Q, and R are predicates other than UNION, FIND, or FLINK. For all the rules that do not have a UNION assertion in their conclusion, the argument is the same as before. Rules with union operations in the conclusion are handled using the union operation which has unit cost for each prefix firing leading to a redundant union operation and where the cost of a nonredundant operation is proportional to the number of new FIND assertions added. Each time two equivalences classes are merged, the find value changes on the smaller of the two classes. This implies that every time the find value of a term changes the size of that term's class at least doubles. So the find value of a term can change at most $\lceil \log_2 N \rceil$ times. This implies that $|F(R(D))| \leq N\lceil \log_2 N \rceil$.

8. *Unification*

Given two first-order terms t_1 and t_2, unification is the problem of determining if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. If such a substitution exists, then one is interested in finding the most general substitution, the substitution γ such that if σ satisfies $\sigma(t_1) = \sigma(t_2)$ then we have that there exists a σ' such that $\sigma = \sigma' \circ \gamma$, that is, $\sigma(u) = \sigma'(\gamma(u))$ for all terms u. Unification is used in logic programming when one allows the database to contain assertions with variables.

This paper assumes that the input to logic programs consists of ground terms, that is, terms not containing variables. The input to a unification problem consists of open terms, that is, terms with variables. However, there is no problem in representing the variables of the input terms with a set of constants that represent input term variables. To give a unification algorithm as a set of inference rules, we assume that the input to the algorithm contains the single assertion UNIFY!(t_1', t_2'), where t_1' and t_2' are ground terms (data structures) representing the input terms t_1 and t_2. In the remainder of this section we will use the term *constant* to mean a constant representing an input constant and the term *variable* to mean a constant representing an input variable. It is possible to represent first-order terms using constants and a single pairing function. So we can assume without loss of generality that the input terms are constructed from constants representing input variables, constants representing input constants, and a single pairing function where we write the pair of e_1 and e_2 as $\langle e_1, e_2 \rangle$.

A unification problem determines an equivalence relation on the subterms of the input terms. More specifiically, two subterms s and w of the input terms t_1' and t_2' are equivalent if for any unifying substitution σ, that is, for any σ with $\sigma(t_1') = \sigma(t_2')$, we have that $\sigma(s) = \sigma(w)$. In the case where a unifying substitution exists, this equivalence relation can be computed using the rules in Figure 6. Multiple conclusions in a rule represent multiple rules—one for each conclusion—and rules R1 and R2 implement the reflexivity property of equality. A clash occurs if the equivalence relation generated by the rules of Figure 6 contains an equivalence

U1 $\dfrac{\text{UNIFY!}(t_1, t_2)}{\text{INPUT}(t_1), \text{INPUT}(t_2), t_1 = t_2}$

U2 $\dfrac{\langle t_1, t_2\rangle = \langle u_1, u_2\rangle}{t_1 = u_1, \quad t_2 = u_2}$

S $\dfrac{x = y}{y = x}$

T $\dfrac{\begin{array}{l} x = y \\ y = z \end{array}}{x = z}$

R1 $\dfrac{\text{INPUT}(\langle x, y\rangle)}{\text{INPUT}(x), \text{INPUT(y)}}$

R2 $\dfrac{\text{INPUT}(x)}{x = x}$

FIG. 6. An $O(n^3)$ algorithm for computing the unification equivalence relation.

U3 $\dfrac{\text{UNIFY!}(x, y)}{\text{UNION}(x, y)}$

U4 $\dfrac{\text{FIND}(\langle x, y\rangle, f)}{\text{UNION}(\Pi_1(f), x), \quad \text{UNION}(\Pi_2(f), y)}$

FIG. 7. An $O(N \log N)$ algorithm for the unification equivalence relation.

class with two distinct constants or a class containing both a pair and a constant. We assume that the variables are associated with integer indexes so we can talk about the variable of least index for any set of variables. If no clash occurs, we can define a particular substitution in terms of the computed equivalence relation as follows: For any term s that is a subterm of the input terms, we define $\sigma(s)$ to be the (unique) constant in the equivalence class of s if such a constant exists, to be the pair $\langle\sigma(u), \sigma(v)\rangle$ if the equivalence class of s contains a pair $\langle u, v\rangle$, and otherwise to be the variable of least index in the equivalence class of s. The term defined in this way is independent of the choice of the pair in the equivalence class if the class contains more than one pair—if the class contains both $\langle u, v\rangle$ and $\langle u', v'\rangle$ then the equivalence class of u must be the same as the equivalence class of u' and similarly for v and v' so $\langle\sigma(u), \sigma(v)\rangle$ is the same as $\langle\sigma(u'), \sigma(v')\rangle$. It is possible, however, that $\sigma(u)$ is an infinite term. For example, starting with UNIFY!$(x, \langle a, x\rangle)$, the rules in Figure 6 generate a two equivalence classes, one for the constant a and one for both the variable x and the pair $\langle a, x\rangle$. In this case $\sigma(x)$ is the infinte terms $\langle a, \langle a, \langle a, \ldots\rangle\rangle\rangle$. In general, if some term s contained in the input is such that $\sigma(s)$ is infinite, then we say that the equivalence relation *contains an occurs-check violation.* Given a union-find representation of the equivalence relation, as constructed by the rules in Figure 6, one can determine whether there is an occurs-check violation in linear time using the linear time algorithm for determining the existince of cycles in a directed graph. See Martelli and Montanari [1982] for details. If there is no clash or occurs-check violation then the two input terms are unifiable, and the substitution mapping x to $\sigma(x)$ is a most general unifying substiution. The rules in Figure 6 contain explicit rules for equality, and the running time of these rules is dominated by the transitivity rule T, which is $O(N^3)$ where N is the number of subterms of the original term.

A more efficient algorithm for computing the same equivalence relation is defined by the rules in Figure 7. We first note that U3 and U4 effectively implement U1

C1 EQUAL!(x, y)

INPUT(x), INPUT(y), $x = y$

C2 INPUT$(\langle x_1, x_2\rangle)$, INPUT$(\langle y_1, y_2\rangle)$
$x_1 = y_1, \quad x_2 = y_2$

$\langle x_1, x_2\rangle = \langle y_1, y_2\rangle$

C3 EQUAL?(x, y)

INPUT(x), INPUT(y)

FIG. 8. An $O(n^3)$ congruence closure algorithm (assuming rules R1, R2, S, and T of Figure 6).

and U2. In particular, if $\langle u_1, u_2\rangle$ is in the same equivalence class as $\langle w_1, w_2\rangle$ then they must both have the same find value f, and both u_1 and w_1 must be equivalent to $\Pi_1(f)$ and hence equivalent to each other.

To analyze the running time of the rules U3 and U4, we first note that the rules maintain the invariant that all find values are terms appearing in the input problem (the union operation breaks ties by using the second argument as the source of the find value). This implies that every union operation is either of the form UNION(s, w) or UNION$(\Pi_i(w), s)$ where s and w appear in the input problem. Let N be the number of distinct terms appearing in the input. We now have that there are only $O(N)$ terms involved in the equivalence relation defined by the FLINK graph. For a given term s, the number of assertions of the form FIND(s, f) is at most the log (base 2) of the size of the equivalence class of s. So we now have that there are only $O(N \log N)$ FIND assertions in the closure. This implies that there are only $O(N \log N)$ prefix firings. Theorem 8 now implies that the closure can be computed in $O(N \log N)$ time. The best-known unification algorithm runs in $O(N)$ time [Paterson and Wegman 1978]. The application of Theorem 8 to rules U3 and U4 yields a slightly worse running time for what is, perhaps, a simpler presentation.

9. *Congruence Closure*

The congruence closure problem is to determine whether an equation $s = t$ between ground terms is provable from a given set of equations between ground terms using the reflexivity, symmetry, transitivity, and congruence rules for equality. As with unification, we will assume that expressions are represented using constants and a single pairing function. The congruence property of equality states that if $u_1 = w_1$ and $u_2 = w_2$ then $\langle u_1, u_2\rangle = \langle w_1, w_2\rangle$. The congruence rule cannot be used directly in a bottom-up logic program because it generates an infinite number of conclusions and hence a bottom-up procedure using this rule directly would fail to terminate.

Figure 8 plus the equality rules R1, R2, S, and T of Figure 6 provide a cubic time algorithm for congruence closure. We take the input to consists of the set of given equations represented by assertions of the form EQUAL!(u, v) and the "goal equation" stated as EQUAL?(s, t). Rules C1, C3, and R1 generate assertions of the form INPUT(e) for all terms e appearing in the input problem. Rule C2 is a variant of the congruence rule restricted so that it can only generate assertions involving input terms. This algorithm terminates in $O(N^3)$ time (dominated by the transitivity rule for equality), where N is the number of input terms. Shostak [1978] proved that running rule C2 on only the input terms suffices.

C1' EQUAL!(x, y) / INPUT(x), INPUT(y), UNION(x, y)

C2' INPUT$(\langle x, y\rangle)$, ID − OR − FIND(x, x'), ID − OR − FIND(y, y') / UNION$(\langle x', y'\rangle, \langle x, y\rangle)$

C3 EQUAL?(x, y) / INPUT(x), INPUT(y)

R1 INPUT$(\langle x, y\rangle)$ / INPUT(x), INPUT(y)

C4 INPUT(x) / ID − OR − FIND(x, x)

C5 FIND(x, y) / ID − OR − FIND(x, y)

FIG. 9. An $O(N \log^3 N)$ algorithm for congruence closure.

Now we consider the congruence closure algorithm given in Figure 9. These rules compute the same equivalence relation on the terms in the input as do the rules in Figure 8. In particular, if $\langle u_1, u_2\rangle$ and $\langle w_1, w_2\rangle$ are both input terms where u_1 and w_1 have been made equivalent, and u_2 and w_2 have been made equivalent, then u_1 and w_1 must have the same find f_1 and w_1 and w_2 must have the same find f_2 and both $\langle u_1, u_2\rangle$ and $\langle w_1, w_2\rangle$ are made equivalent to $\langle f_1, f_2\rangle$. To analyze the complexity of the rules in Figure 9 we first note that, since the union operation breaks ties by selecting the find value from the second argument, the rules maintain the invariant that every find value is an input term. Given this, one can see that all terms involved in the equivalence relation are either input terms or pairs of input terms. This implies that there are at most $O(N^2)$ terms involved in the equivalence relation, where N is the number of distinct terms in the input. So we have that for any given term s the number of assertions of the form FIND(s, f) is $O(\log N)$. So the number of firings of the rule C2′ is $O(N \log^2 N)$. But this implies that the number of terms involved in the equivalence relation is actually only $O(N \log^2 N)$. Since each such term can appear in the left-hand side of at most $O(\log N)$ FIND assertions, there can be at most $O(N \log^3 N)$ FIND assertions. Theorem 8 now implies that the closure can be computed in $O(N \log^3 N)$ time. It is possible to show that by erasing obsolete FIND assertions the algorithm can be made to run in $O(N \log N)$ time—the best-known running time for congruence closure.

10. *Henglein's Algorithm for Object Type Inference*

Type inference is the problem of taking a program without type declarations and inferring types for program variables. For many languages and type systems, it is possible to determine, for a given program without type declarations, whether or not there exist type declarations under which the program is well typed. In this case, we say that the type inference problem is decidable. Perhaps the most fundamental type inference algorithm is for the Hindley-Milner type system used in the programming language ML [Milner 1978]. Here we present an inference rule version of Henglein's quadratic time algorithm for determining typability in a variant of the Abadi-Cardelli object calculus [Henglein 1999; Abadi and Cardelli 1996]. This algorithm is interesting because the first algorithm published for the

problem was a classical dynamic transitive closure algorithm requiring $O(N^3)$ time [Palsberg 1995] and because Henglein's presentation of the quadratic algorithm is given as in classical pseudo-code and is fairly complex. The algorithm presented here is given as a set of union-find inference rules for which Theorem 8 yields a run time of $O(N^2 \log N)$.

The type inference problem solved by Henglein's [1999] algorithm is for object-oriented programs under a certain type system for objects. An object can be viewed as a record with fields. An object type specifies types for fields. For example, the type $[\ell_1 = \text{INT}, \ell_2 = \text{INT}]$ denotes set of all objects in which the fields ℓ_1 and ℓ_2 are both integers. Note that the type $[\ell_1 = \text{INT}, \ell_2 = \text{INT}]$ is a subtype (a subset) of the type $[\ell_1 = \text{INT}]$—anything in which both fields ℓ_1 and ℓ_2 are integers is something where the field ℓ_1 is an integer. In the "pure" object calculus of Abadi and Cardelli [1996], there are only objects—there are no integers, procedures, or other data types. The pure calculus is of theoretical interest because it isolates and simplifies the nature of the objects and object types. In the pure object calculus, type expressions are defined by the following grammar where α represents type variables:

$$\sigma ::= \alpha | [\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n] | \mu\alpha.\sigma.$$

This grammar allows for the universal type [] that places no constraints on an object and hence represents the set of all objects. In the Abadi-Cardelli [1996] language, objects compute the values for fields on demand (rather than storing the value in the slot). On-demand computation of slot values allows objects to be "infinitely deep." In particular, recursive types such as $\mu\alpha[\ell_1 = \sigma, \ell_2 = \alpha]$ are meaningful and denote the type α of objects where slot ℓ_1 has type σ and in which ℓ_2 (recursively) has type α. A type expression is closed if all type variables in that expression are bound in μ expressions, for example, the expression $\mu\alpha[\ell_1 = \alpha]$ is closed.

A presentation of the Abadi-Cardelli [1996] programming language is beyond the scope of this paper. Here we simply note that the problem of determining the existence of acceptable type declarations can be converted to a problem of determining whether there exist type expressions satisfying a certain set of constraints [Henglein 1999]. More specifically, we can take the input to be a set of inequalities of the form $\sigma_1 \leq \sigma_2$, where σ_1 and σ_2 are finite nonrecursive type expressions (as defined by the first two cases of the above grammar). The problem is to find (possibly recursive) closed type expressions for the type variables such that the constraints are satisfied. To define this problem precisely, one must define the inequality relation $\sigma_1 \leq \sigma_2$ for closed type expressions σ_1 and σ_2. Here we are interested in an "invariant" interpretation of type inequality—a closed type $[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n]$ is a subtype of a closed type $[m_1 = \tau_1; \ldots; m_k = \tau_k]$ if each m_i is equal to some ℓ_j where σ_j *equals* τ_i. Equality on (recursive) types is defined to mean that the (possibly infinite) type expressions that result from unrolling all recursive definitions are equal.

Although superficially the type inference problem may seem quite complex, there is a very simple cubic time decision procedure. We assume that the input has been preprocessed so that, for each type expression $[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n]$ appearing in the input (either at the top level or as a subexpression of a top-level type expression), the database also includes all assertions of the form ACCEPTS($[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n], \ell_i$) and $[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n].\ell_i = \sigma_i$ with $1 \leq i \leq n$. We also assume that, rather than use rule R1 of Figure 6, the input is preprocessed so that for every

P1 $\dfrac{\sigma = \tau}{\sigma \leq \tau, \tau \leq \sigma}$

P2 $\dfrac{\sigma \leq \tau, \tau \leq \gamma}{\sigma \leq \gamma}$

P3 $\dfrac{\text{INPUT}(\tau)}{\tau \leq \tau}$

P4 $\dfrac{\begin{array}{l}\text{ACCEPTS}(\tau, \ell) \\ \text{ACCEPTS}(\sigma, \ell) \\ \gamma \leq \tau \\ \gamma \leq \sigma\end{array}}{\sigma.\ell = \tau.\ell}$

FIG. 10. Palsberg's $O(N^3)$ object type inference algorithm.

type expression τ appearing in the input the initial database contains the assertion INPUT(τ). Note that this preprocessing can be done in linear time. Palsberg's [1995] cubic algorithm can be given as a bottom-up logic program consisting of the rules in Figure 10.

The rules are sound in the sense that, under any interpretation of the type variables as type expressions, if the premises are true under the above mentioned notions of type equality and subtyping, then the conclusion is true. If the rules derive $[\ell_1 = \tau_1, \ldots, \ell_k = \tau_k] \leq [m_1 = \sigma_1, \ldots, m_n = \sigma_n]$, where there exists an m_i that is not equal to any ℓ_j, then we have a contradiction and the input constraints are not satisfiable, that is, there is no interpretation of the type variables as types under which all the input assertions are simultaeously true. If an assertion of this form is derived then we say that *the input constraints are rejected*. If the input constraints are not rejected then one can construct a variable interpretation satisfying the constraints as follows: For each type expression γ appearing in the input, we define the type $\sigma(\gamma)$ recursively to be the type expression $[\ell_1 = \sigma(\tau_1.\ell_1), \ldots, \ell_k = \sigma(\tau_k.\ell_k)]$, where the pairs $\langle \ell_i, \tau_i \rangle$ are all pairs such that the rules derive $\gamma \leq \tau_i$ where we have ACCEPTS(τ_i, ℓ_i). Rule P4 ensures that, for a given field ℓ_i, the type $\sigma(\tau_i.\ell_i)$ is independent of the choice of τ_i. The recursion need not terminate and the type expression $\sigma(\gamma)$ may in fact be infinite. However, in the case where it is infinite, the number of distinct subexpressions can be no larger than the number of distinct type expressions in the input. Since the number of distinct (infinite) type expressions is finite, these types can be represented by finite recursive type expressions.

Figure 11 gives union-find inference rules which perform the same analysis. The equality relation is stored in the union-find data structure. The inequality relation is stored in the relation $\Rightarrow$ and its transitive closure $\Rightarrow^*$. Rule P1 of Figure 10 is implemented by rules H5 and H2 of Figure 11. Rule P4 of Figure 10 is implemented by two applications of rule H6 of Figure 11. Note that the convention of breaking ties in favor of the second argument in a UNION assertion maintains the invariant that find values are always type expressions appearing in the input. This implies that the equivalence relation on the expressions appearing in the input is determined by the set of FLINK assertions between input expressions. This observation implies that rule P2 of Figure 10 is faithfully implemented by H3 and H4 of Figure 11. Rule P3 of Figure 10 is not required in the formulation in Figure 11.

We now consider the running time of the rules in Figure 11. Let N be the number of input assertions after the preprocessing described above. There are clearly only

H1 $\dfrac{\tau \leq \sigma}{\tau \Rightarrow \sigma}$

H2 $\dfrac{\text{INPUT}(\tau)\quad \text{FLINK}(\tau, \sigma)}{\tau \Rightarrow \sigma, \sigma \Rightarrow \tau}$

H3 $\dfrac{\sigma \Rightarrow \tau}{\tau \Rightarrow^* \sigma}$

H4 $\dfrac{\sigma \Rightarrow \tau,\ \ \tau \Rightarrow^* \gamma}{\sigma \Rightarrow^* \gamma}$

H5 $\dfrac{\tau = \sigma}{\text{UNION}(\tau, \sigma)}$

H6 $\dfrac{\text{ACCEPTS}(\tau, \ell)\quad \sigma \Rightarrow^* \tau}{\text{UNION}(\sigma.\ell, \tau.\ell)}$

FIG. 11. On $O(N^2 \log N)$ object type inference algorithm.

$O(N)$ distinct type expressions in the input. This implies that there can be at most $O(N)$ FLINK assertions between type assertions appearing in the input—every such FLINK assertion reduces the number of equivalence classes by 1 and therefore there cannot be more such assertions than input type expressions. Hence the total number of prefix firings of rule H2 is $O(n)$. This implies that there are only $O(N)$ $\Rightarrow$ assertions, all of which are between input type expressions. The fact that there are only $O(N)$ $\Rightarrow$ assertions implies that there are at most $O(N^2)$ prefix firings of the transitivity rule H4. Furthermore, there are at most $O(N)$ prefix firings of H1, H3, and H5 and at most $O(N^2)$ prefix firings of H6. So the total number of prefix firings of the rules in Figure 11 is $O(N^2)$. It remains only to consider the number of FIND assertions. The number of FIND assertions is at most $m \log m$, where m is the number of nodes involved in the FIND assertions. Rule H6 introduces new nodes of the form $\sigma.\ell$, where σ is an input expression and ℓ is a field name used in other input type expressions. There can be $O(N^2)$ such new type expressions introduced in this way. The number of FIND assertions of the form FIND($\sigma.\ell$, τ), where $\sigma.\ell$ is one of the newly created nodes, is no more than $O(N^2 \log N^2) = O(N^2 \log N)$. By Theorem 8, the run time required is $O(N^2 \log N)$. By only computing find compressions on terms appearing in the input, the algorithm can be modified to run in $O(N^2)$ time. However, this improved run time is not given directly by the general run time theorem for union-find rule sets.

11. *Conclusions*

This paper has argued that many algorithms have natural presentations as bottom-up logic programs and that such presentations are clearer and simpler to analyze, both for correctness and for complexity, than classical pseudo-code presentations. A variety of examples have been given and analyzed. These examples suggest a variety of directions for further work.

In the case of unification and Henglein's [1999] algorithm, final checks were performed by a postprocessing pass. In unification, the postprocessing involves checking for clashes and occurs-check violations. In Henglein's [1999] algorithm, one must check that one has not derived an assertion of the form $\sigma \leq \tau$ where τ accepts a field not acepted by σ. We might consider extensions to logic programming that allow these postprocessing steps to be naturally expressed as rules. Stratified

negation by failure would allow a natural way of inferring NOT(ACCEPTS(σ, ℓ)) in Henglein's [1999] algorithm while preserving the truth of Theorems 1 and 8. This would allow the acceptability check to be done with rules. A simple extension of the union-find formalism would allow the detection of an equivalence between distinct "constants" and hence allow the rules for unification to detect clashes. It might also be possible to extend the language to improve the running time for cycle detection and strongly connected component analysis for directed graphs.

Another direction for further work involves aggregation. It would be nice to have language features and run time theorems allowing natural and efficient renderings of Dijkstra's [1959] shortest-path algorithm and the inside algorithm for computing the probability of a given string in a probabilistic context-free grammar [Lari and Young 1990].

REFERENCES

ABADI, M., AND CARDELLI, L. 1996. *A Theory Of Objects*. Springer-Verlag, Berlin, Germany.

AHO, A. V., AND ULLMAN, J. 1986. *Compilers: Principles Techniques and Tools*. Addison Wesley, Reading, MA.

AIKEN, A., WIMMERS, E., AND LAKSHMAN, T. K. 1994. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, pp. 163–173.

APPEL, A. W. 1997. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, U.K.

BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. D. 1986. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGMODSIGACT Symposium on Principles of Database Systems*. ACM Press, New York, pp. 1–15.

BONDORF, A., AND JORGENSEN, A. 1993. Efficient analysis for realistic off-line partial evaluation. *J. Funct. Program. 3*, 3, 315–346.

CHEN, W., AND WARREN, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 1, 20–74.

COUSOT, P., AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, pp. 238–252.

DIJKSTRA, E. W. 1959. A note on two problems in connection with graphs. *Numer. Math. 1*, Oct., 269–271.

DOWNING, W., AND GALLIER, J. W. 1984. Linear time algorithms for testing the satisfiability of propositional horn formulae. *J. Logic Program. 1*, 3, 267–284.

EISNER, J., AND SATTA, G. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the Annual Conference of the Association for Computational Linguistics* (ACL-99). Morgan Kaufman, Menlo Park, Calif., pp. 457–464.

FÄHNDRICH, M., FOSTER, J., SU, Z., AND AIKEN, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Programming Language Design and Implementation* (PLDI 98). ACM Press, New York, pp. 85–96.

HEINTZE, N. 1994. Set based analysis of ml programs. In *ACM Conference on Lisp and Functional Programming*. ACM Press, New York, pp. 306–317.

HEINTZE, N., AND JAFFAR, J. 1990a. A decision procedure for a class of set constraints. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 42–51.

HEINTZE, N., AND JAFFAR, J. 1990b. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, pp. 197–209.

HEINTZE, N., AND MCALLESTER, D. 1997a. Linear time subtransitive control flow analysis. In *Conference on Programming Language Design and Implementation* (PLDI 97). ACM Press, New York, pp. 26–272.

HEINTZE, N., AND MCALLESTER, D. 1997b. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings, Twelvth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 342–361.

HENGLEIN, F. 1999. Breaking through the n^3 barrier: Faster object type inference. *Theor. Pract. Obj. Syst. 5*, 1, 57–72. A Preliminary Version appeared in FOOL4.

KOZEN, D., PALSBERG, J., AND SCHWARTZBACH, M. I. 1994. Efficient inference of partial types. *J. Comput. Syst. Sci. 49*, 2 (Oct.), 306–324.

LARI, K., AND YOUNG, S. J. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Comput. Speech Lang. 4*, 1, 35–56.

MARTELLI, A., AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2, 258–282.

MCALLESTER, D. 1996. Inferring recursive types. Available online at http://www.research.mit.edu/~dmac.

MELSKI, D., AND REPS, T. 1997. Intercovertability of set constraints and context free language reachability. In *ACM SIGPLAN Symposium of Partial Evaluation and Semantic-Based Program Manipulation* (PEPM'97). ACM Press, New York, pp. 74–89.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3, 348–375.

NAUGHTON, J., AND RAMAKRISHNAN, R. 1991. Bottom-up evaluation of logic programs. In *Computational Logic*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, Mass.

PALSBERG, J. 1995. Efficient inference of object types. *Inform. Comput. 123*, 2, 198–209.

PALSBERG, J., AND O'KEEFE, P. 1995. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst. 17*, 4, 576–599.

PATERSON, M. S., AND WEGMAN, M. N. 1978. Linear unification. *J. Comput. Syst. Sci. 16*, 2 (April), 158–167.

PEREIRA, F., AND WARREN, D. 1983. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*. Morgan Kaufman, Menlo Park, Calif., pp. 137–144.

REPS, T. 1994. *Demand Interprocedural Program Analysis Using Logic Databases*. Kluwer Academic Publishers, Norwell, Mass., pp. 163–196.

ROCIO, V., AND LOPES, J. G. 1998. Partial parsing, deduction and tabling. In *Proceedings of Tabulation in Parsing and Deduction* (TAPD'98) (Paris, France, April 1998), pp. 52–61.

ROHMER, J., LESCOEUR, R., AND KERISIT, J. M. 1986. The Alexander method—a technique for the processing of recursive axioms in deductive database queries. *New Gen. Comput. 4*, 3, 273–285.

SAGONAS, K., SWIFT, T., AND WARREN, D. S. 1994. Xsb as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (SIGMOD'94, Minneapolis, Minn.). ACM Press, New York, pp. 442–453.

SHIVERS, O. 1991. Data flow analysis and type recovery in scheme. In *Topics in Advanced Language Implementation*, P. Lee, Ed. MIT Press, Cambridge, Mass.

SHOSTAK, R. 1978. An algorithm for reasoning about equality. *Comm. ACM 21*, 2 (July), 583–585.

SHIEBER, S. M., SCHABES, Y., AND PEREIRA, F. 1995. Principles and implementation of deductive parsing. *J. Logic Program. 24*, 1-2 (July/Aug.), 3–36.

TAMAKI, H., AND SATO, T. 1986. Old resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 225. Springer-Verlag, London, England, pp. 84–96.

ULLMAN, J. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems* (March 1998). ACM Press, New York, pp. 140–149.

ULLMAN, J., AND RAMAKRISHNAN, R. 1995. A survey of research in deductive database systems. *J. Logic Program. 23*, 2, 125–150.

VARDI, M. 1982. Complexity of relational query languages. In *Proceedings of the 14th ACM SIGACT Symposium on Theory of Computating* (San Francisco, Calif.). ACM Press, New York, pp. 137–146.

RECEIVED OCTOBER 1999; REVISED MAY 2002; ACCEPTED JUNE 2002