On the Complexity Analysis of Static Analyses

David McAllester

AT&T Labs-Research
180 Park Ave
Florham Park NJ 07932
dmac@research.att.com
http://www.research.att.com/~dmac

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 meta-complexity 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.

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. For example, 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, i.e., the transitive closure of the input. The inference 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.

In spite of the clarity of inference rules as specifications, it is not obvious how one should view a set of inference rules as an algorithm. An algorithm has a well defined running time. But what is the running time of a set of inference rules? This paper presents two meta-theorems which state upper bounds on the running time of a bottom-up (forward chaining) execution of an arbitrary set of inference rules. By associating a set of inference rules with a running time, these theorems provide a way of viewing a set of inference rules as an algorithm. The transitivity rule given above runs in $O(n^3)$ time where n is the number of nodes in the input graph. A more efficient program for sparse inputs consists of the two rules $\mathtt{EDGE}(x,\ y) \rightarrow \mathtt{PATH}(x,\ y)$ and $\mathtt{EDGE}(x,\ y) \wedge \mathtt{PATH}(y,\ z) \rightarrow \mathtt{PATH}(x,\ z)$. 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 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 meta-complexity theorem given in section 2 implies that, for programs with at most two antecedents per rule, the running time is bounded by the size of the final data base plus the number of provable instances of the left hand sides of rules. The more efficient program runs in $O(en)$ time rather than $O(n^3)$.

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.

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

This program computes all assertions of the form $A \Rightarrow s_{i,j}$ where the nonterminal X generates the string $s_i \ldots s_j$. 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 meta-complexity theorem in section 2 states that, in general, the running time of a bottom-up logic program is bounded by the size of the final closure plus the number of "prefix firings" of the rules, i.e., the number of provable instances of prefixes of the antecedents of rules. In general there may be more provable instances of the pair of the first two antecedents of a rule than of the triple of the first three antecedents. For the above parsing program, however, the pair of the first two antecedents of the second rule has at most $|G|n^2$ provable instances and the total number of prefix firings is $O(|G|n^3)$ which gives the running time of the algorithm. This is a logic program presentation of the Cocke-Kasimi-Younger (CKY) algorithm for context-free parsing.

Bottom-up logic programming has been widely studied in the context of deductive databases [25, 24, 17]. Bottom-up logic programming has been advocated as a formalism for expressing a variety of natural language parsing algorithms [23, 5]. Bottom-up logic programming has also been advocated for program analysis algorithms used by compilers [24, 21]. The contribution of this paper lies in the two meta-complexity theorems which provide a simple characterization of the running time of logic programs. A characterization of running time seems essential if one is to view a logic program as an algorithm.

1.2 A Framework for Static Analysis

This paper is as much about static analysis in particular as about logic programming in general. Static analysis is used in compilers. 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 — properties of a program are 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 meta-complexity theorems is either the best known or within a polylog factor of the best known running time for that analysis.

This paper takes the position that the ability to easily associate logic programs with running times makes them a useful general formalism for expressing and analyzing static analysis algorithms. Two other paradigms have achieved wide recognition as useful general frameworks for static analysis — abstract interpretation [4] and set constraints [2, 9, 8]. 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, i.e., as a special case of abstract interpretation, 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.

1.3 Overview

Section 2 presents the first meta-complexity theorem and some basic examples. Sections 3, 4, and 5 present respectively liveness analysis, data flow analysis, and flow analysis (both data and control) in the lambda calculus. Section 6 presents the second main result of this paper — a meta-complexity theorem for an extended bottom-up programming language incorporating the union-find algorithm. Sections 7 and 8 present unification and congruence closure algorithms respectively. Section 9 presents Henglein's quadratic time algorithm for typability in a version of the Abadi-Cardelli object calculus [12]. 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 [18]. 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 $O(n^2)$ running time is easily derived from the union-find meta-complexity theorem.

2 A First Meta-Complexity 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, i.e. a first order formula of the form $A_1 \wedge \ldots \wedge A_n \rightarrow C$ where C and each A_i is a first order atom, i.e., 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). We will use *assertion* to mean a ground atom, i.e., 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 $A_1 \wedge \ldots \wedge A_n \rightarrow C$ if for any ground substitution σ defined on the variables in the rule, if $\sigma(A_1) \in E$, ..., $\sigma(A_n) \in E$ then $\sigma(C) \in E$. The output $R(D)$ can be defined as the least database contianing 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 is 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 while the logically equivalent rule (2) to take $O(n^2)$ time. If there was 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 (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 1. *We define a* prefix firing *of a rule $A_1 \wedge \ldots \wedge A_n \rightarrow C$ in database E to be a pair $\langle \sigma,\ i \rangle$ where $1 \leq i \leq n$ and where σ is a ground substitution defined on 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.*

Rule sets can be recursive — it is possible that a rules 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 non-recursive rules as nested iterations, it is less obvious that this view is appropriate for recursive rules. The first meta-complexity theorem can be viewed as stating that the nested iteration view applies to recursive rules as well.

Theorem 1. *For any rule set R there exists an algorithm for mapping D to $R(D)$ which runs in time $O(|R(D)| + |P_R(R(D))|)$.*

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 $\mathtt{EDGE}(x, y) \rightarrow \mathtt{PATH}(x, y)$ and $\mathtt{EDGE}(x, y) \wedge \mathtt{PATH}(y, z) \rightarrow \mathtt{PATH}(x, z)$. Suppose R consists of these two rules and D consists of e assertions of the form $\mathtt{EDGE}(c, d)$ 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 at most en. The closure contains at most $n^2 \leq en$ assertions. 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 $\mathtt{SUCC}(i, i+1)$ for each $1 \leq i \leq n-1$ where n is the length of the input string. Logic programming and the meta-complexity theorem can be extended to handle arithmetic, although we will not formally consider arithmetic here.

$$\frac{\begin{array}{l} X \Rightarrow y \\ s(i) = y \end{array}}{X \Rightarrow s(i,i)} \qquad\qquad \frac{\begin{array}{l} X \Rightarrow YZ \\ Y \Rightarrow s(i,j) \\ \mathtt{SUCC}(j,j') \\ Z \Rightarrow s(j',k) \end{array}}{X \Rightarrow s(i,k)}$$

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 $\mathtt{SUCC}(i, i+1)$ 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 $i, j \in [1, n]$. Hence we have that $|R(D(G, S))|$ is $O(|G|n^2)$. 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 the value of y and there are then at most n ways of continuing with an instantiation of i. So there are at most $|G|n$ 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 value 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 $O(|G|n^3)$.

The proof of theorem 1 is based on a source to source transformation of the given program R. If r is a rule $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow C$ then we define 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 1. *If $\pi(D) = D$, i.e., the input does not use the "fresh" predicates, then $R(D) = \pi(B(R)(D))$.*

The proof can be done by two inductions on proof length — 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 2. *If $\pi(D) = D$ then we have the following.*

$$|B(R)(D)| = |R(D)| + |P_R(R(D))|$$

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

Proof: The first half of the lemma follows from the observation that $B(R)(D)$ consists of $R(D)$ plus a distinct assertion of the form $P_i(x_1, \ldots, x_{k_i})$ for each element of $P_R(R(D))$. The second half also follows from the observation that the assertion 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 $P_i(x_1, \ldots, x_{k_1})$ 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 $P(x_1, \ldots, x_{k_i})$. Hence there are exactly twice as many prefix firings of $B(R)$ as there are of R. ■

Lemmas 1 and 2 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.

$$A_1 \rightarrow P_1(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m))$$
$$A_2 \rightarrow P_2(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k))$$

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

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

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

$$|I(R)(D)| \leq |R(D)| + 2|R||R(D)| + |P_R(R(D))|$$

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

Proof: First consider $|I(R)(D)|$. We have that $I(R)(D)$ contains $R(D)$ plus assertions of the form $P_1(x, y)$, $P_2(y, z)$ and $Q(x, y, z)$. Each assertion in $R(D)$ can generate at most $|R|$ assertions of the $P_1(x, y)$ and at most $|R|$ assertions of the form $P_2(y, z)$. Finally, each assertion of the form $Q(x, y, z)$ corresponds to a firing of a two antecedent rule R. Hence the total number of assertions in $I(R)(D)$ can be no larger than $|R(D)|+2|R||R(D)|+|P_R(R(D))|$. Now consider $|P_{I(R)}(I(R)(D))|$. Each prefix firing of $I(R)$ is either a firing of 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 $|R||R(D)|$ 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 $3|P_R(R(D))| + 2|R||R(D)|$. ■

Lemmas 3 and 4 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.

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 $A \to C$ 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 $P_1(x,\ y) \wedge P_2(y,\ z) \to C$ 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 contains a variable then go into an infinite loop.
2. If Ψ is already marked do nothing.
3. Otherwise, mark Ψ and add Ψ to Q.

Fig. 1. The algorithm underlying theorem 1

Lemma 5. *If $R(D)$ is finite then the algorithm terminates with E equal to $R(D)$.*

Proof: Assume that $R(D)$ is finite. This implies that the algorithm never asserts an open atom, i.e., one containing variables, because otherwise any instance of that atom would be in $R(D)$ and $R(D)$ would have to be infinite. So we can assume that step 1 of the assert procedure is never executed. 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. Furthermore, when the algorithm terminates then the final value of E must be a subset of $R(D)$. The algorithm also maintains the invariant that 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 6. *The algorithm can be run to completion in $O(|R(D)| + |P_R(R(D))|)$ time.*

Proof: Through out 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. Interning also supports unit time marking and checking for the presence of marks.

The initialization step takes time proportional to $|D|$. 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 $A \to C$ and a given ground atom Ψ one must determine if there exists a σ such that $\Phi = \sigma(A)$. Given that equality testing can be unit time, 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 $|R(D)|$ or $|P_R(R(D))|$. Assuming that hash table operations take unit time, 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 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 consideredin step 4a the assertion $P_1(t_1,\ t_2)$ must equal Φ. Hence this triple can not be visited again in a later invocation of step 4a. A similar statement applies to 4b. ■

3 Liveness Analysis

We now turn to applications of meta-complexity 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 in a given variable is no longer needed so that a register being used to store the variable can now be used for other purposes. 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, e.g., addition, multiplication, or Boolean comparison, and l_i and l_j are instruction labels — a number unique to the labeled instruction.

$$
\begin{aligned}
&l_i : x = \mathrm{op}(y,\ z);\\
&l_i : \text{if } x \text{ goto } l_k;\\
&l_i : \text{goto } l_k\\
&l_i : \text{halt}
\end{aligned}
$$

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 $x = \mathrm{op}(y,\ z)$ 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 $\langle l,\ \sigma \rangle$.

L1
$$\frac{l : x = \mathrm{op}(y,\ z)}{\mathrm{live}(y,\ l),\ \mathrm{live}(z,\ l)}$$

L2
$$\frac{\begin{array}{l} l : x = \mathrm{op}(y,\ z) \\ \mathtt{SUCC}(l,\ l') \\ \mathrm{live}(w,\ l') \\ \mathtt{DISTINCT}(w,\ x) \end{array}}{\mathrm{live}(w,\ l)}$$

L3
$$\frac{\begin{array}{l} l : \mathrm{goto}\ l' \\ \mathtt{live}(w,\ l') \end{array}}{\mathtt{live}(w,\ l)}$$

L4
$$\frac{\begin{array}{l} l : \mathrm{if}\ x\ \mathrm{goto}\ l' \\ \mathtt{live}(w,\ l') \end{array}}{\mathtt{live}(w,\ l)}$$

L5
$$\frac{\begin{array}{l} l : \mathrm{if}\ x\ \mathrm{goto}\ l' \\ \mathtt{SUCC}(l,\ l'') \\ \mathtt{live}(w,\ l'') \end{array}}{\mathtt{live}(w,\ l)}$$

Fig. 2. A Liveness Analysis Algorithm

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 $\mathrm{live}(x,\ l)$. 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 $\mathrm{live}(x,\ \sigma)$ then x is not live at any state of the form $\langle l,\ \sigma \rangle$.

So if the rules do not derive 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 all antecedents involving a computed predicate P are of the form $P(x_1, \ldots, x_n)$, where each x_i is a variable and either that 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 N prefix firings. Now consider rule L2. The first antecedent determines all bindings other than w. There are N ways of instantiating the first antecedent and 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). Hence there are only $O(L)$ prefix firings of the second type. For each of the rules L3, L4, and L5 we have that each firing generates a distinct conclusion and hence the number of firings is $O(L)$.

4 Data Flow Analysis

Some programming languages, such as Common Lisp and Scheme, use type tags on data values and generate "graceful" run time errors if a run-time type violation occurs, e.g., an attempt is made to extract a slot from a non-structure. 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. 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. As in most static analyses, data flow analysis is a conservative approximation to an undecidable problem.

Here we formulate data flow in a simple but 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 langauges (e.g., Scheme) it is syntactically impossible to write a program that uses a variable before assigning it some initial value. In such langauges 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 $y = \Pi_1(x)$.

D1 $$\frac{x = \mathtt{k}}{x \Rightarrow \mathtt{INT}}$$

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

D3 $$\frac{\begin{array}{l} y = \Pi_j(x) \\ x \Rightarrow \langle z_1,\ z_2\rangle \end{array}}{y \Rightarrow z_j}$$

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

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.

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 on 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 $w \Rightarrow^* v$. 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 $x \Rightarrow$ `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 $x \Rightarrow$ `INT` is derivable from a given set of assignments using the rules in figure 3 is 2NPDA complete [11, 15]. 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 sub-cubic time if it can be solved in $O(n^k)$ time for $k < 3$. If a 2NPDA-hard problem can be solved in sub-cubic time then all problems in 2NPDA can be solved in sub-cubic time. The data flow problem is 2NPDA-complete in the sense that it is in the class 2NPDA and is 2NPDA-hard. No sub-cubic procedure is known for any 2NPDA-complete problem and it seems reasonable to conjecture that no such procedure exists for computing the information specified by figure 3.

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 [10]. This more efficient algorithm can also be stated and analyzed as a set of inference rules, although we will not do so here.

5 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. 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($e,\ \sigma$) and $\langle e,\ \sigma\rangle \Rightarrow^* v$. Intuitively, the assertion compute($e,\ \sigma$) states that 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 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($w,\ \sigma$). Note that the rules maintain the invariant that in all derivable assertions of the form compute($w,\ \sigma$) 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 the analysis in figure 5 is conservative, i.e., to establish its correctness, we view each assertions 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 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($e,\ \sigma$), 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 checks 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

E1 $$\frac{\text{compute}((f\ w),\ \sigma)}{\texttt{compute}(f,\ \sigma),\ \ \texttt{compute}(w,\ \sigma)}$$

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

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

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

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

E6 $$\frac{\begin{array}{l}p \Rightarrow q\\ q \Rightarrow^* v\end{array}}{p \Rightarrow^* v}$$

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

E8 $$\frac{\begin{array}{l}\text{compute}(\langle e_1,\ e_2\rangle,\ \sigma)\\ \langle e_1,\ \sigma\rangle \Rightarrow^* v_1\\ \langle e_2,\ \sigma\rangle \Rightarrow^* v_2\end{array}}{\langle\langle e_1,\ e_2\rangle,\ \sigma\rangle \Rightarrow^* \langle v_1,\ v_2\rangle}$$

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

E10 $$\frac{\begin{array}{l}\text{compute}(\Pi_j(u),\ \sigma)\\ \langle u,\ \sigma\rangle \Rightarrow^* \langle v_1,\ v_2\rangle\end{array}}{\langle \Pi_j(u),\ \sigma\rangle \Rightarrow^* v_j}$$

Fig. 4. An algorithm for evaluating lambda terms.

F1 $$\frac{\mathrm{compute}((f\ w))}{\mathrm{compute}(f),\ \ \mathrm{compute}(w)}$$

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

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

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

F7 $$\frac{\mathrm{compute}(\langle e_1,\ e_2\rangle)}{\begin{array}{l}\mathrm{compute}(e_1),\ \ \mathrm{compute}(e_2),\\ \langle e_1,\ e_2\rangle \Rightarrow^* \langle e_1,\ e_2\rangle\end{array}}$$

F9 $$\frac{\mathrm{compute}(\Pi_j(u))}{\mathrm{compute}(u)}$$

F10 $$\frac{\begin{array}{l}\mathrm{compute}(\Pi_j(u))\\ u \Rightarrow^* \langle e_1,\ e_2\rangle\end{array}}{\Pi_j(u) \Rightarrow e_j}$$

Fig. 5. Flow analysis for the λ-Calculus with pairing.

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 [2, 7]. The rules can also be used to determine if the given term is typable by recursive types with function, pairing, and union types [14] using arguments similar to those relating control flow analysis to partial types [13, 19]. It is possible to give a sub-transitive 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 [10]. The sub-transitive analysis algorithm can also be presented as a bottom-up logic program whose running time can be analyzed using theorem 1.

6 A Union-Find Meta-Complexity 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 [3], the quadratic type inference algorithm for an Abadi-Cardelli object calculus given by Henglein [12], and the dramatic improvement in empirical performance due to equality reported by Fahndrich et al. in [6]. 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 meta-complexity theorem for such union find rule sets.

We let `UNION`, `FIND`, and `MERGE` be three distinguished binary predicate symbols. The predicate `UNION` can appear in rule conclusions but not in rule antecedents. The predicates `FIND` and `MERGE` can appear in rule antecedents but not in rule conclusions. A bottom-up bound 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. An assertion of the form `MERGE`$(u,\ w)$ means that at some point a union operation was applied to u and w and, at the time of that union operation, u and w were not equivalent. An assertion `FIND`$(u,\ f)$ means that at some point the find of u was the value f.

For any given database we define the merge graph to be the undirected graph containing an edge between s and w if either `MERGE`$(s,\ w)$ or `MERGE`$(w,\ s)$ is in the database. If there is a path from s to w in the merge graph then we say that s and w are equivalent. We say that a database is union-find consistent if for every term s whose equivalence class contains at least two members there exists a unique term f such that for every term w in the equivalence class of s the database contains `FIND`$(w,\ f)$. This unique term is called the find of s. Note that a database not containing any `MERGE` or `FIND` assertions is union-find consistent. We now define the result of performing a union operation on the terms s and t in a union-find consistent database. If s and t are already equivalent then the union operation has no effect. If s and t are not equivalent then the union operation adds the assertion `MERGE`$(s,\ t)$ plus all assertions of the form `FIND`$(w,\ f)$ where w is equivalent to either s or t and f is the find of the larger equivalence class if either equivalence class contains more than one member — otherwise f is the term t. The fact that the find value is the second argument if both equivalence classes are singleton is significant for the complexity analysis of the unification and congruence-closure algorithms. Note that if either class contains more than one member, and w is in the larger class, then the assertion `FIND`$(w,\ f)$ does not need to be added. With appropriate indexing the union operation can be run in time proportional to number of new assertions added, i.e., the size of the smaller equivalence class. Also note that whenever the find value of term changes the size of the equivalence class of that term at least doubles. This implies that for a given term s the number of terms f such that E contains `FIND`$(s,\ f)$ is at most log (base 2) of the size of the equivalence class of s.

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, the erasure process does not improve the bound given in theorem 2 below. In fact, such erasure makes the theorem more difficult to state. In order to allow for a relatively simply meta-complexity theorem we do not erase obsolete `FIND` assertions.

We define an clean database to be one not containing `MERGE` 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 $\mathtt{UNION}(u, w)$, $\mathtt{UNION}(s, w)$, and $\mathtt{UNION}(u, s)$ then the merge graph will contain only two arcs and the graph depends on the order in which the union operations are done. If rules are used to derived other assertions from the `MERGE` 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 meta-complexity theorem for union-find rule sets.

Theorem 2. *For any 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 $R(D)$.*

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 `MERGE`. 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 non-redundant operation is proportional to the number of new `FIND` assertions added.

7 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 $\sigma = \sigma' \circ \gamma$, i.e., $\sigma(u) = \sigma'(\gamma(u))$ for all terms u. Unification is used in logic programming when one allows the database to contain assertions with variables.

To give a unification algorithm as a set of inference rules we assume that the input to the algorithm contains the single assertion $\mathtt{UNIFY!}(t'_1,\ t'_2)$ where t'_1 and t_2 prime are ground terms (data structures) representing the input terms t_1 and t_2. 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 and a single pairing function when we write the pair of e_1 and e_2 as $\langle e_1,\ e_2\rangle$. An elegant but inefficient unification algorithm is defined by the following rule plus the reflexivity, symmetry rules for the predicate $=$.

U1 $\quad \dfrac{\texttt{UNIFY!}(t_1,\ t_2)}{t_1 = t_2}$ $\qquad$ U2 $\quad \dfrac{\langle t_1,\ t_2\rangle = \langle u_1,\ u_2\rangle}{t_1\ = u_1, \quad t_2 = u_2}$

Before presenting a more efficient union-find based algorithm we consider the correctness of the above rules as an implementation of unification. The rules define an equivalence relation on the subterms of the input terms. The constant symbols in the input terms are divided into two types — those representing constants and functions of the original terms and those representing variables of the original terms. If two different non-variable constants become equated, or if a non-variable constant is equated with a pair then we say that a "clash" has occurred. In this case the input terms are not unifiable. We can also define a "subterm" relation that takes into account the equivalence relation. More specifically, we say that e is a virtual subterm of w if either e is a subterm of w or w is equivalent to a term w' such that e is a virtual subterm of w'. If the virtual subterm relation contains a cycle then we say that the unification results in an occurs-check violation (some term occurs inside itself). If there is no clash and no occurs-check violation then a most general unifier can be constructed as follows. First one selects an element of each equivalence class where we give preference to non-variable elements — if the class contains a non-variable then the canonical element must be a non-variable. Then we define σ_e to be the canonical representative of the equivalence class of e if that canonical representative is not a pair, and to be $\langle \sigma_u,\ \sigma_w\rangle$ if the canonical representative is the pair $\langle u,\ v\rangle$. If the virtual subterm relation is acyclic then σ_e is a finite term. The most general unifier is the substitution mapping x to σ_x. The details of the correctness proof for this unification algorithm are beyond the scope of this paper. Here we focus on finding a simple presentation of a more efficient algorithm for constructing the equivalence relation defined by rules U1 and U2.

The algorithm defined by rules U1 and U2 uses explicit rules for equality (which are omitted above) rather a union-find data structure. The running time of U1 and U2 can be analyzed using theorem 1. Let N be the number of subterms of the input term. The size of $R(D)$ is $O(N^2)$ and the number of prefix firings is dominated by the number of prefix firings of the transitivity rule for equality and is $O(N^3)$. A more efficient algorithm for computing the same equivalence relation is defined by the following two rules.

U3 $\quad \dfrac{\texttt{UNIFY!}(x,\ y)}{\texttt{UNION}(x,\ y)}$ $\qquad$ U4 $\quad \dfrac{\texttt{FIND}(\langle x,\ y\rangle, f)}{\texttt{UNION}(\Pi_1(f),\ x),\ \ \texttt{UNION}(\Pi_2(f),\ y)}$

We first note that U3 and U4 effectively implement U1 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 $\Pi_i(f)$ 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 be using the second argument as the source of the find value). This implies that every union operation is either of the form $\texttt{UNION}(s,\ w)$ or $\texttt{UNION}(\Pi_i(w),\ s)$ where s and w appear in 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 merge graph. For a given term s the number of assertions of the form $\texttt{FIND}(s,\ f)$ 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 2 now implies that the closure can be computed in $O(N \log N)$ time. The best known unification algorithm runs in $O(N)$ time [20] and the best on-line unification algorithm runs in $O(N\alpha(N))$ time where α is the inverse of Ackermann's function. The application of theorem 2 to rules U3 and U4 yields a slightly worse running time for what is, perhaps, a simpler presentation.

8 Congruence Closure

The congruence closure problem is to determine whether an equation $s = t$ between ground terms is provable form 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 can not 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 6 gives 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 $\texttt{EQUAL!}(u,\ v)$ and the "goal equation" stated as $\texttt{EQUAL?}(s,\ t)$. Figure 6 assumes the reflexivity, symmetry and transitivity rules for equality.

Rules C1, C2, and C3 generate assertions of the form $\texttt{INPUT}(e)$ for all terms e appearing in the input problem. Rule C4 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. It is possible to prove that running the congruence rule on only the input terms suffices [22].

Now we consider the congruence closure algorithm given in figure 7. These rules compute the same equivalence relation on the terms in the input as do the rules in figure 6. In particular, if $\langle u_1,\ u_2\rangle$ and $\langle w_1,\ w_2\rangle$ are both input terms

C1 $\dfrac{\texttt{EQUAL?}(x,\ y)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y)}$

C2 $\dfrac{\texttt{EQUAL!}(x,\ y)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y),\ x = y}$

C3 $\dfrac{\texttt{INPUT}(\langle x,\ y\rangle)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y)}$

C4 $\dfrac{\texttt{INPUT}(\langle x_1,\ x_2\rangle),\ \texttt{INPUT}(\langle y_1,\ y_2\rangle) \quad x_1 = y_1,\ \ x_2 = y_2}{\langle x_1,\ x_2\rangle = \langle y_1,\ y_2\rangle}$

Fig. 6. A cubic congruence closure algorithm.

C1 $\dfrac{\texttt{EQUAL?}(x,\ y)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y)}$

C2' $\dfrac{\texttt{EQUAL!}(x,\ y)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y),\ \texttt{UNION}(x,\ y)}$

C3 $\dfrac{\texttt{INPUT}(\langle x,\ y\rangle)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y)}$

C5 $\dfrac{\texttt{INPUT}(x)}{\texttt{ID-OR-FIND}(x,\ x)}$

C6 $\dfrac{\texttt{FIND}(x,\ y)}{\texttt{ID-OR-FIND}(x,\ y)}$

C7 $\dfrac{\texttt{INPUT}(\langle x,\ y\rangle) \quad \texttt{ID-OR-FIND}(x,\ x') \quad \texttt{ID-OR-FIND}(y,\ y')}{\texttt{UNION}(\langle x',\ y'\rangle,\ \langle x,\ y\rangle)}$

Fig. 7. An $O(N \log^3 N)$ algorithm for congruence closure.

where u_1 and w_1 are 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 7 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 $\texttt{FIND}(s,\ f)$ is $O(\log n)$. So the number of firings of the congruence rule 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 2 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.

9 Henglein's Quadratic Algorithm

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 [16]. Here we consider Henglein's quadratic time algorithm for determining typability in a variant of the Abadi-Cardelli object calculus [12, 1]. This algorithm is interesting because the first algorithm published for the problem was a classical dynamic transitive closure algorithm requiring $O(N^3)$ time [18] and because Henglein's presentation of the quadratic algorithm is given in classical pseudo-code and is fairly complex.

The type inference problem solved by Henglein's algorithm is for object-oriented programs under a certain type system for objects. An object can be viewed as a record with fields or slots. An object type specifies types for fields. For example, the type $[\ell_1 = \texttt{INT},\ \ell_2 = \texttt{INT}]$ denotes set of all objects in which the fields ℓ_1 and ℓ_2 are both integers. Note that the type $[\ell_1 = \texttt{INT},\ \ell_2 = \texttt{INT}]$ is a subtype (a subset) of the type $[\ell_1 = \texttt{INT}]$ — 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 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.

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

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 language objects compute the values for slots 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 set of objects α 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, e.g., the expression $\mu\alpha[\ell_1 = \alpha]$ is closed.

A presentation the Abadi-Cardelli 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 problem of determining whether there exists type expressions satisfying a certain set of constraints. 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 type expression as defined by the above grammar. The problem is to find 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 $\texttt{ACCEPTS}([\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n],\ \ell_i)$ and $[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n].\ell_i = \sigma_i$ with $1 \leq i \leq n$. Note that this preprocessing can be done in linear time. The cubic algorithm can be given as a bottom-up logic program consisting of the following rules plus the reflexivity, symmetry, and transitivity rules for =.

$$\frac{\sigma = \tau}{\sigma \leq \tau}$$

$$\frac{\sigma \leq \tau,\ \tau \leq \gamma}{\sigma \leq \gamma}$$

$$\frac{\begin{array}{l}\texttt{ACCEPTS}(\tau,\ \ell)\\ \texttt{ACCEPTS}(\sigma,\ \ell)\\ \sigma \leq \tau\end{array}}{\sigma.\ell = \tau.\ell}$$

We say that the input is rejected by the rules if the rules derive an assertion of the form $\sigma \leq \tau$ where τ accepts a field not accepted by σ. One can show that the rules are "sound" in the sense that if the rules reject the input (derive a contradiction) then the constraints are unsatisfiable. Conversely, if the rules do not reject the input then it is possible to show that one can construct a solution to the constraints although the proof is beyond the scope of this paper. By counting prefix firings we get that this algorithm runs in $O(N^3)$ time.

$$\frac{\tau \leq \sigma}{\tau \Rightarrow \sigma} \qquad \frac{\tau = \sigma}{\mathtt{UNION}(\tau,\ \sigma)}$$

$$\frac{\mathtt{MERGE}(\tau,\ \sigma)}{\tau \Rightarrow \sigma,\quad \sigma \Rightarrow \tau} \qquad \frac{\begin{array}{l}\mathtt{ACCEPTS}(\tau,\ \ell)\\ \mathtt{ACCEPTS}(\sigma,\ \ell)\\ \sigma \Rightarrow^* \tau\end{array}}{\mathtt{UNION}(\sigma.\ell,\ \tau.\ell)}$$

$$\frac{\sigma \Rightarrow \tau}{\tau \Rightarrow^* \sigma}$$

$$\frac{\sigma \Rightarrow \tau,\quad \tau \Rightarrow^* \gamma}{\sigma \Rightarrow^* \gamma}$$

Fig. 8. Henglein's type inference algorithm.

Figure 8 gives a quadratic union-find algorithm which computes the same closure as the cubic time rules. The equality relation is stored in the union-find data structure. The inequality relation is stored in the relation $\Rightarrow$ and its transitive closure $\Rightarrow^*$. Recall that a merge assertion is generated for each non-redundant union operation. There can be at most $O(N)$ merge assertions. Hence the base ordering $\Rightarrow$ as only $O(N)$ edges. The version of transitive closure implemented in these rules is $O(en)$ for an input graph of e edges and n nodes — we get only $O(N^2)$ prefix firings in the transitive closure rules. The number of prefix firings in the remaining rules is also $O(N^2)$ and the number of find assertions is $O(N \log N)$ so the total running time $O(N^2)$.

10 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 algorithm final checks were performed by a post-processing pass. It is possible to extend the logic programming language in ways that allow more algorithms to be fully expressed as rules? Stratified negation by failure would allow a natural way of inferring NOT(ACCEPTS(σ, ℓ)) in Henglein's algorithm while preserving the truth of theorems 1 and 2. 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 meta-complexity theorems allowing natural and efficient renderings of Dijkstra's shortest path algorithm and the inside algorithm for computing the probability of a given string in a probabilistic context free grammar.

References

1. M. Abadi and L. Cardelli. *A Theory Of Objects.* Springer-Verlag, 1996.
2. A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages*, pages 163–173. Association for Computing Machinery, 1994.
3. A. Bondorf and A. Jorgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of functional Programming*, 3(3), 1993.
4. P. Cousot and R. Cousot. 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*, pages 238–252, 1977.
5. Jason Eisner and Giorgio Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *ACL-99*, pages 457–464, 1999.
6. Manual Fähndrich, Jeffrey Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI98*, 1998.
7. N. Heintze. Set based analysis of ml programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
8. N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51. IEEE Computer Society Press, 1990.
9. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languages*, pages 197–209. Association for Computing Machinery, 1990.

10. Nevin Heintze and David McAllester. Linear time subtransitive control flow analysis. In *PLDI-97*, 1997.
11. Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings, Twelvth Annual IEEE Symposium on Logic in Computer Science*, pages 342–361. IEEE Computer Society Press, 1997.
12. Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. *Theory and Practice of Object Systems (TAPOS)*, 5(1):57–72, 1999. A Preliminary Version appeared in FOOL4.
13. Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *J. Comput. Syst. Sci.*, 49(2):306–324, October 1994.
14. David McAllester. Inferring recursive types. Available at http://www.research.mit.edu/~dmac, 1996.
15. D. Melski and T. Reps. Intercovertability of set constraints and context free language reachability. In *PEPM'97*, 1997.
16. Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.
17. Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.
18. J. Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.
19. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *POPL95*, pages 367–378, 1995.
20. M. S. Paterson and M. N. Wegman. Linear unification. *JCSS*, 16:158–167, 1978.
21. Thomas Reps. *Demand interprocedural program analysis using logic databases*, pages 163–196. Kluwer Academic Publishers, 1994.
22. R. Shostak. An algorithm for reasoning about equality. *Comm. ACM.*, 21(2):583–585, July 1978.
23. Stuart M. Shieber snf Yves Schabes and Fernando Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1-2):3–36, July/August 1995.
24. J. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eigth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 140–149, March 1989.
25. M. Vardi. Complexity of relational query languages. In *14th Symposium on Theory of Computation*, pages 137–146, 1982.