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

This is a slightly revised version of a paper accompanying an invited talk at SAS-99

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 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. Most static analysis algorithms have natural representations as bottom-up logic programs, i.e., as inference rules with a forward-chaining procedural interpretation. The technical content of this paper consists of two meta-complexity theorems which allow, in many cases, the running time of a bottom-up logic logic program to be determined by inspection. This paper presents and analyzes a variety of static analysis algorithms which have natural presentations as bottom-up logic programs. For these examples the running time of the bottom-up presentation, as determined by the meta-complexity theorems given here, is either the best known or within a polylog factor of the best known.

We use the term "inference rule" to mean 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. First order Horn clauses form a Turing complete model of computation and can be used in practice as a general purpose programming language. The atoms $A_1, \ldots, A_n$ are called the antecedents of the rule and the atom C is called the conclusion. When using inference rules as a programming language one represents arbitrary data structures as first order terms. For example, one can represent terms of the lambda calculus or arbitrary formulas of first order logic as first order terms in the underlying programming language. The restriction to first order terms in no way rules out the construction of rules defining static analyses for higher order languages.

There are two basic ways to view a set of inference rules as an algorithm — the backward chaining approach taken in traditional Prolog interpreters [13,4] and the forward chaining, or bottom-up approach common in deductive databases [23,22,18]. Meta-complexity analysis derives from the bottom-up approach. As a simple example consider the rule $P(x,y) \wedge P(y,z) \rightarrow P(x,z)$ which states that the binary predicate P is transitive. Let D be a set of assertions of the form $P(c,d)$ where c and d are constant symbols. More generally we will use the term *assertion* to mean a ground atom, i.e., an atom not containing variable, 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 in the obvious way from assertions in D using rules in R. If R consists of the above rule for transitivity, and D consists of assertions of the form $P(c,d)$, then $R(D)$ is simply the transitive closure of D. In the bottom-up view a rule set R is taken to be an algorithm for computing output $R(D)$ from input D.

Here we are interested in methods for quickly determining the running time of a rule set R, i.e., the time required to compute $R(D)$ from D. For example, consider the following "algorithm" for computing the transitive closure of a predicate $\mathtt{EDGE}$ defined by the bottom-up rules $\mathtt{EDGE}(x,y) \rightarrow \mathtt{PATH}(x,y)$ and $\mathtt{EDGE}(x,y) \wedge \mathtt{PATH}(y,z) \rightarrow \mathtt{PATH}(x,z)$. If the input graph contains e edges this algorithm runs in $O(en)$ time — significantly better than $O(n^3)$ for sparse graphs. Note that the $O(en)$ running time can not be derived by simply counting the number of variables in any single rule. Section 4 gives a meta-complexity theorem which applies to arbitrary rule sets and which allows the $O(en)$ running time of this algorithm to be determined by inspection. For this simple rule set the $O(en)$ running time may seem obvious, but examples are given throughout the paper where the meta-complexity theorem can be used in cases where a completely rigorous treatment of the running time of a rule set would be otherwise tedious. The meta-theorem proved in section 4 states that $R(D)$ can be computed in time proportional to the number of "prefix firings" of the rules in R — the number of derivable ground instances of prefixes of rule antecedents. This theorem holds for arbitrary rule sets, no matter how complex the antecedents or how many antecedents rules have, provided that every variable in the conclusion of a rule appears in some antecedent of that rule.

Before presenting the first significant meta-complexity theorem in section 4, section 3 reviews a known meta-complexity theorem based on counting the number of variables in a single rule. This can be used for "syntactically local" rule sets — ones in which every term in the conclusion of a rule appears in some antecedent. Some other basic properties of syntactically local rule sets are also mentioned briefly in section 3 such as the fact that syntactically local rule sets can express all and only polynomial time decidable term languages.

Sections 4 gives the first significant meta-complexity theorem and some basic examples, including the CKY algorithm for context-free parsing. Although this paper focuses on static analysis algorithms, a variety of parsing algorithms, such as Eisner and Satta's recent algorithm for bilexical grammars [6], have simple complexity analyses based on the meta-complexity theorem given in section 4.

Section 5 gives a series of examples of program analysis algorithms expressed as bottom-up logic programs. The first example is basic data flow. This algorithm computes a "dynamic transitive closure" — a transitive closure operation in which new edges are continually added to the underlying graph as the computation proceeds. Many such dynamic transitive closure algorithms can be shown to be 2NPDA-complete [11, 17]. 2NPDA is the class of languages that can be recognized by a two-way nondeterministic pushdown automaton. A problem is 2NPDA-complete if it is in the class 2NPDA and furthermore has the property that if it can be solved in sub-cubic time then any problem in 2NPDA can also be solved in sub-cubic time. No 2NPDA-complete problem is known to be solvable in sub-cubic time. Section 5 also presents a linear time sub-transitive data flow algorithm which can be applied to programs typable with non-recursive data types of bounded size and a combined control and data flow analysis algorithm for the λ-calculus. In all these examples the meta-complexity theorem of section 4 allows the running time of the algorithm to be determined by inspection of the rule set.

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. Three basic applications of this meta-complexity theorem for union-find rules are presented in section 7 — a unification algorithm, a congruence closure algorithm, and a type inference algorithm for the simply typed λ-calculus. Section 8 presents Henglein's quadratic time algorithm for typability in a version of the Abadi-Cardelli object calculus [12]. This 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 [19]. 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 Terminology and Assumptions

As mentioned in the introduction, we will use the term *assertion* to mean a ground atom, i.e., atom not containing variables, and use the term *database* to mean a set of assertions. Also as mentioned in the introduction, for any rule set R and database D we let $R(D)$ be the set of ground assertions derivable from D using rules in R. We write $D \vdash_R \Phi$ as an alternative notation for $\Phi \in R(D)$. We use $|D|$ for the number of assertions in D and $||D||$ for the number of distinct ground terms appearing either as arguments to predicates in assertions in D or as subterms of such arguments.

A ground substitution is a mapping from a finite set of variables to ground terms. In this paper we consider only ground substitutions. If σ is a ground substitution defined on all the variables occurring a term t the $\sigma(t)$ is defined in the standard way as the result of replacing each variable by its image under σ. We also assume that all expressions — both terms and atoms — are represented as interned dag data structures. This means that the same term is always represented by the same pointer to memory so that equality testing is a unit time operation. Furthermore, we assume that hash table operations take unit time so that for any substitution σ defined (only) on x and y we can compute (the pointer representing) $\sigma(f(x,\ y))$ in unit time. Note that interned expressions support indexing. For example, given a binary predicate P we can index all assertions of the form $P(t, w)$ so that the data structure representing t points to a list of all terms w such that $P(t, w)$ has been asserted and, conversely, all terms w point to a list of all terms t such that $P(t, w)$ has been asserted.

We are concerned here with rules which are written with the intention of defining bottom-up algorithms. Intuitively, in a bottom-up logic program any variable in the conclusion that does not appear in any antecedent is "unbound" — it will not have any assigned value when the rule runs. Although unbound variables in conclusions do have a well defined semantics, when writing rules to be used in a bottom-up way it is always possible to avoid such variables. A rule in which all variables in the conclusion appear in some antecedent will be called *bottom-up bound*. In this paper we consider only bottom-up bound inference rules.

A datalog rule is one that does not contain terms other than variables. A syntactically local rule is one in which every term in the conclusion appears in some antecedent — either as an argument to a predicate or as a subterm of such an argument. Every syntactically local rule is bottom-up bound and every bottom-up bound datalog rule is syntactically local. However, the rule $P(x) \rightarrow P(s(x))$ is bottom-up bound but not syntactically local. Note that the converse, $P(s(x)) \rightarrow P(x)$ is syntactically local.

3 Local Rule Sets

Before giving the main results of this paper, which apply to arbitrary rule sets, we give a first "naive" meta-complexity theorem. This theorem applies only to syntactically local rule sets. Because every term in the conclusion of a syntactically local rule appears in some antecedent, it follows that a syntactically local rule can never introduce a new term. This implies that if R is syntactically local then for any database D we have that $R(D)$ is finite. More precisely, we have the following.

Theorem 1. *If R is syntactically local then $R(D)$ can be computed in $O(|D| + ||D||^k)$ time where k is the largest number of variables occurring any single rule.*

To prove the theorem one simply notes that it suffices to consider the set of ground Horn clauses consisting of the assertions in D (as unit clauses) plus all instances of the rules in R in which all terms appear in D. There are $O(||D||^k)$

such instances. Computing the inferential closure of a set of ground clauses can be done in linear time [5].

As the $O(en)$ transitive closure example in the introduction shows, theorem 1 provides only a crude upper bound on the running time of inference rules. Before presenting the second meta-complexity theorem, however, we briefly mention some addition properties of local rule sets that are not used in the remainder of the paper but are included here for the sake of completeness. The first property is that syntactically local rule sets capture the complexity class $\mathcal{P}$. We say that a rule set R *accepts* a term t if $\mathtt{INPUT}(t) \vdash_R \mathtt{ACCEPT}(t)$. The above theorem implies that the language accepted by a syntactically local rule set is polynomial time decidable. The following less trivial theorem is proved in [8]. It states the converse — any polynomial time property of first-order terms can be encoded as a syntactically local rule set.

Theorem 2 (Givan & McAllester). *If $\mathcal{L}$ is a polynomial time decidable term language then there exists a syntactically local rule set which accepts exactly the terms in $\mathcal{L}$.*

The second subject we mention briefly is what we will call here semantic locality. A rule set R will be called *semantically local* if whenever $D \vdash_R \Phi$ there exists a derivation of Φ from assertions in D using rules in R such that every term in that derivation appears in D. Every syntactically local rule set is semantically local. By the same reasoning used to prove theorem 1, if R is semantically local then $R(D)$ can be computed in $O(|D| + ||D||^k)$ time where k is the largest number of variables in any single rule. In many cases it possible to mechanically show that a given rule set is semantically local even though it is not syntactically local [15, 2]. However, semantic locality is in general an undecidable property of rule sets [8].

4 A Second Meta-Complexity Theorem

We now prove our second meta-complexity theorem. We will say that a database E is closed under a rule set R if $R(E) = E$. It would seem that determining closedness would be easier than computing the closure in cases where we are not yet closed. The meta-complexity theorem states, in essence, that the closure can be computed quickly — it can be computed in the time needed to merely check the closedness of the final result. Consider a rule $A_1 \wedge \ldots \wedge A_n \rightarrow C$. To check that a database E is closed under this rule one can compute all ground substitutions σ such that $\sigma(A_1)$, ... $\sigma(A_n)$ are all in E and then check that $\sigma(C)$ is also in E. To find all such substitutions we can first match the pattern A_1 against assertions in the database to get all substitutions σ_1 such that $\sigma_1(A_1) \in D$. Then given σ_i such that $\sigma_i(A_1)$, ..., $\sigma_i(A_i)$ are all in E we can match $\sigma_i(A_{i+1})$ against the assertions in the database to get all extensions σ_{i+1} such that $\sigma_{i+1}(A_1)$, ..., $\sigma_{i+1}(A_{i+1})$ are in E. Each substitution σ_i determines a "prefix firing" of the rule as defined below.

Definition 1. *We define a* prefix firing *of a rule $A_1, \ldots, A_n \to C$ in a rule set R under database E to be a ground instance $B_1, \ldots, B_i$ of an initial sequence $A_1, \ldots, A_i$, $i \leq n$, such that $B_1, \ldots, B_n$ are all contained in D. We let $P_R(E)$ be the set of all prefix firings of rules in R for database E.*

Note that the rule $P(x, y) \wedge P(y, z) \wedge R(z) \to P(x, z)$ might have a large number of firings for the first two antecedents while having no firings of all three antecedents. The simple algorithm outlined above for checking that E is closed under R requires at least $|P_R(E)|$ steps of computation. As outlined above, the closure check algorithm would actually require more time because each step of extending σ_i to σ_{i+1} involves iterating over the entire database. The following theorem states that we can compute $R(D)$ in time proportional to $|D|$ plus $|P_R(R(D))|$.

Theorem 3. *For any set R of bottom-up bound inference rules there exists an algorithm for mapping D to $R(D)$ which runs in $O(|D| + |P_R(R(D))|)$ time.*

Before proving theorem 3 we consider some simple applications. Consider the transitive closure algorithm defined by the inference rules $\texttt{EDGE}(x, y) \to \texttt{PATH}(x, y)$ and $\texttt{EDGE}(x, y) \wedge \texttt{PATH}(y, z) \to \texttt{PATH}(x, z)$. If R consists of these two rules and D consists of e assertions of the form $\texttt{EDGE}(c, d)$ involving n constants then we immediately have that $|P_R(R(D))|$ is $O(en)$. So theorem 3 immediately implies that the algorithm runs in $O(en)$ time.

$$\frac{\begin{array}{l} U \to a \\ \texttt{INPUT(CONS(}a,\ j\texttt{))} \end{array}}{\texttt{PARSES(}U,\ \texttt{CONS(}a,\ j\texttt{)},\ j\texttt{)}} \qquad \frac{\begin{array}{l} A \to BC \\ \texttt{PARSES(}B,\ i,\ j\texttt{)} \\ \texttt{PARSES(}C,\ j,\ k\texttt{)} \end{array}}{\texttt{PARSES(}A,\ i,\ k\texttt{)}}$$

$$\frac{\texttt{INPUT(CONS(X, Y))}}{\texttt{INPUT(Y)}}$$

Fig. 1. The Cocke-Kasimi-Younger (CKY) parsing algorithm. $\texttt{PARSES(}u,\ i,\ j\texttt{)}$ means that the substring from i to j parses as nonterminal u.

As a second example, consider the algorithm for context free parsing shown in figure 1. The grammar is given in Chomsky normal form and consists of a set of assertions of the form $X \to a$ and $X \to YZ$. The input sting is represented as a "lisp list" of the form $\texttt{CONS}(a_1,\ \texttt{CONS}(a_2,\ \ldots\ \texttt{CONS}(a_n,\ \texttt{NIL})))$ and the input string is specified by an assertion of the form $\texttt{INPUT}(s)$. Let g be the number of productions in the grammar and let n be the length of the input string. Theorem 3 immediately implies that this algorithm runs in $O(gn^3)$ time. Note that there is a rule with six variables — three string index variables and three grammar nonterminal variables.

We now give a proof of theorem 3. The proof is based on a source to source transformation of the given program. We note that each of the following source to source transformations on inference rules preserve the quantity $|D|+|P_R(R(D))|$ (as a function of D) up to a multiplicative constant. In the second transformation note that there must be at least one element of D or $P_r(R(D))$ for each assertion in $R(D)$. Hence adding any rule with only a single antecedent and with a fresh predicate in the conclusion at most doubles the value of $|D|+|P_R(R(D))|$. The second transformation can then be done in two steps — first we add the new rule and then replace the antecedent in the existing rule. A similar analysis holds for the third transformation.

$A_1,\ A_2,\ A_3,\ \ldots A_n \rightarrow\ C \quad \Rightarrow$

$$A_1,\ A_2 \rightarrow P(x_1,\ \ldots,\ x_n), \quad P(x_1,\ \ldots,\ x_n),\ A_3,\ \ldots,\ A_n \rightarrow C$$

where $x_1,\ \ldots, x_n$ are all free variables in A_1 and A_2.

$A_1,\ \ldots,\ P(t_1,\ \ldots,\ t_n),\ \ldots, A_n \rightarrow C \quad \Rightarrow$

$$P(t_1,\ \ldots,\ t_n) \rightarrow Q(x_1,\ \ldots,\ x_m), \quad A_1,\ \ldots,\ Q(x_1,\ \ldots,\ x_m),\ \ldots, A_n \rightarrow C$$

where at least one of t_i is a non-variable and $x_1,\ \ldots,\ x_m$ are all the free variables in $t_1,\ \ldots,\ t_n$.

$P(x_1,\ \ldots,\ x_n),\ Q(y_1,\ \ldots,\ y_m) \rightarrow C \quad \Rightarrow$

$$P(x_1,\ \ldots,\ x_n) \rightarrow P'(f(z_1,\ \ldots,\ z_k),\ g(w_1,\ \ldots\ w_h))$$

$$Q(y_1,\ \ldots,\ y_n) \rightarrow Q'(g(w_1,\ \ldots,\ w_h),\ u(v_1,\ \ldots\ v_j))$$

$$P'(x,y),\ Q'(y,z) \rightarrow R(x,y,z)$$

$$R(f(x_1,\ \ldots,\ x_n),\ g(w_1,\ \ldots,\ w_h),\ u(v_1,\ \ldots,\ v_m)) \rightarrow C$$

where $z_1,\ \ldots,\ z_k$ are those variables among the x_is which are not among the y_is; $w_1,\ \ldots,\ w_h$ are those variables that occur both in the x_is and y_is; and $v_1,\ \ldots\ v_i$ are those variables among the y_is that are not among the x_is.

These transformations allow us to assume without loss of generality that the only multiple antecedent rules are of the form $P(x,\ y), Q(y,\ z) \rightarrow R(x,\ y,\ z)$. For each such multiple antecedent rule we create an index such that for each y we can enumerate the values of x such that $P(x,\ y)$ has been asserted and also enumerate the values of z such that $Q(y,\ z)$ has been asserted. When a new assertion of the form $P(x,\ y)$ or $Q(y,\ z)$ is derived we can now iterate over the possible values of the missing variable in time proportional to the number of such values.

5 Basic Examples

Figure 2 gives a simple first-order data flow analysis algorithm. The algorithm takes as input a set of assignment statements of the form $\mathtt{ASSIGN}(x,\ e)$ where x is a program variable and e is a either a "constant expression" of the form $\mathtt{CONSTANT}(n)$, a tuple expression of the form $\langle y,\ z\rangle$ where y and z are program variables, or a projection expression of the form $\Pi_1(y)$ or $\Pi_2(y)$ where y is a program variable. Consider a database D containing e assignment assertions involving n program variables and pair expressions. Clearly the first rule (upper left corner) has at most e firings. The transitivity rule has at most n^3 firings. The other two rules have at most en firings. Since e is $O(n^2)$, theorem 3 implies that the algorithm given in figure 2 runs in $O(n^3)$ time.

It is possible to show that determining whether a given value can reach a given variable, as defined by the rules in figure 2, is 2NPDA complete [11, 17]. 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.

Cubic time is impractical for many applications. If the problem is changed slightly so as to require that the assignment statements are well typed using types of a bounded size, then the problem of determining if a given value can reach a given variable can be solved in linear time. This can be done with sub-transitive data flow analysis [10]. In the first-order setting of the rules in figure 2 we use the types defined by the following grammar.

$$\tau ::= \mathtt{INT} \mid \langle \tau,\ \tau \rangle$$

Note that this grammar does not allow for recursive types. The linear time analysis can be extended to handle list types and recursive types but giving an analysis weaker than that of figure 2. For simplicity we will avoid recursive types here. We now consider a database containing assignment statements such as those described above but subject to the constraint that it must be possible to assign every variable a type such that every assignment is well typed. For example, if the database contains $\mathtt{ASSIGN}(x,\ \langle y, z\rangle)$ then x must have type $\langle \tau,\ \sigma\rangle$ where τ and σ are the types of y and z respectively. Similarly, if the database contains $\mathtt{ASSIGN}(y,\ \Pi_i(x))$ then x must have a type of the form $\langle \tau,\ \sigma\rangle$ where y has type τ. Under these assumptions we can use the inference rules given in figure 3.

Note that the rules in figure 3 are not syntactically local. The inference rule at the lower right contains a term in the conclusion, namely $\Pi_j(e_2)$, which is not contained in any antecedent. This rules does introduce new terms. However, it is not difficult to see that the rules maintain the invariant that for every derived assertion of the form $e_1 \Leftarrow e_2$ we have that e_1 and e_2 have the same

$$\frac{\mathtt{ASSIGN}(x,\ \mathtt{CONSTANT}(n))}{x \Leftarrow \mathtt{CONSTANT}(n)}$$

$$\frac{\mathtt{ASSIGN}(x,\ \langle y,\ z\rangle)}{x \Leftarrow \langle y,\ z\rangle}$$

$$\frac{\begin{array}{l}\mathtt{ASSIGN}(y,\ \Pi_j(x)) \\ x \Leftarrow \langle z_1,\ z_2\rangle\end{array}}{y \Leftarrow z_j}$$

$$\frac{u \Leftarrow w,\quad w \Leftarrow v}{u \Leftarrow v}$$

Fig. 2. 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.

$$\frac{\mathtt{ASSIGN}(x,\ \mathtt{CONSTANT}(n))}{x \Leftarrow \mathtt{CONSTANT}(n)}$$

$$\frac{\mathtt{ASSIGN}(x,\ \langle y,\ z\rangle)}{\Pi_1(x) \Leftarrow y,\quad \Pi_2(x) \Leftarrow z}$$

$$\frac{\mathtt{ASSIGN}(y,\ \Pi_j(x))}{y \Leftarrow\ \Pi_j(x)}$$

$$\frac{e \Leftarrow\ \Pi_j(x)}{\mathtt{COMPUTE\text{-}}\Pi_j(x)}$$

$$\frac{e_1 \Leftarrow e_2,\quad \mathtt{COMPUTE\text{-}}\Pi_j(e_1)}{\Pi_j(e_1) \Leftarrow \Pi_j(e_2)}$$

Fig. 3. Sub-transitive data flow analysis. A rule with multiple conclusions represents multiple rules — one for each conclusion.

$$\frac{\mathtt{SOURCE}(x)}{\mathtt{REACHES}(x)}$$

$$\frac{z \Leftarrow y,\quad \mathtt{REACHES}(y)}{\mathtt{REACHES}(z)}$$

Fig. 4. Determining the existence of a path from a given source.

$$\frac{\texttt{INPUT}((f\ w))}{\texttt{INPUT}(f),\ \ \texttt{INPUT}(w)}$$

$$\frac{\texttt{INPUT}(\lambda x.e)}{\texttt{INPUT}(e),\ \ \lambda x.e \rightarrow \lambda x.e}$$

$$\frac{\texttt{INPUT}(\langle e_1,\ e_2\rangle)}{\texttt{INPUT}(e_1),\ \ \texttt{INPUT}(e_2),\ \ \langle e_1,\ e_2\rangle \rightarrow \langle e_1,\ e_2\rangle}$$

$$\frac{\texttt{INPUT}(\Pi_j(u))}{\texttt{INPUT}(u)}$$

$$\frac{\begin{array}{l}\texttt{INPUT}((f\ w))\\ f \rightarrow \lambda x.u\end{array}}{x \rightarrow w,\ \ (f\ w) \rightarrow u}$$

$$\frac{\begin{array}{l}\texttt{INPUT}(\Pi_j(u))\\ u \rightarrow \langle e_1,\ e_2\rangle\end{array}}{\Pi_j(u) \rightarrow e_j}$$

$$\frac{u \rightarrow w,\ \ w \rightarrow v}{u \rightarrow v}$$

Fig. 5. A flow analysis algorithm for the λ-Calculus with pairing. The rules are intended to be applied to an initial database containing a single assertion of the form $\texttt{INPUT}(e)$ where e is a closed λ-calculus term which has been α-renamed so that distinct bound variables have distinct names. Note that the rules rules are syntactically local — every term in a conclusion appears in some antecedent. Hence all terms in derived assertions are subterms of the input term. The rules compute a directed graph on the subterms of the input.

type. This implies that every newly introduced term must be well typed. For example, if the rules construct the expression $\Pi_1(\Pi_2(x))$ then x must have a type of the form $\langle\tau,\ \langle\sigma,\ \eta\rangle\rangle$. Since the type expressions are finite, there are only finitely many such well typed terms. So the inference process must terminate. In fact if no variable has a type involving more than b syntax nodes then the inference process terminates in linear time. To see this it suffices to observe that the rules maintain the invariant that for every derived assertion involving $\Leftarrow$ is of the form $\Pi j_1(\ldots \Pi_{j_n}(e_1)) \Leftarrow \Pi j_1(\ldots \Pi_{j_n}(e_2))$ where the assertion $e_1 \Leftarrow e_2$ is derived directly from an assignment using one of the rules on the left hand side of the figure. If the type of x has only b syntax nodes then an input assignment of the form $\mathtt{ASSIGN}(x,\ e)$ can lead to at most b derived $\Leftarrow$ assertion. So if there are n assignments in the input database then there are at most bn derived assertions involving $\Leftarrow$. It is now easy to check that each inference rule has at most bn firings. So by theorem 3 we have that the algorithm runs in $O(bn)$ time.

It is possible to show that these rules construct a directed graph whose transitive closure includes the graph constructed by the rules in figure 2. So to determine if a given source value flows to a given variable we need simply determine if there is a path from the source to the variable. It is well known that one can determine in linear time whether a path exists from a given source node to any other node in a directed graph. However, we can also note that this computation can be done with the algorithm shown in figure 4. The fact that the algorithm in figure 4 runs in linear time is guaranteed by Theorem 3.

As another example, figure 5 gives an algorithm for both control and data flow in the λ-calculus extended with pairing and projection operations. These rules implement a form of set based analysis [1, 9]. The rules can also be used to determine if the given term is typable by recursive types with function, pairing, and union types [16] using arguments similar to those relating control flow analysis to partial types [14, 20]. A detailed discussion of the precise relationship between the rules in figure 5, set based analysis, and recursive types is beyond the scope of this paper. Here we are primarily concerned with the complexity analysis of the algorithm. All rules other than the transitivity rule have at most n^2 prefix firings and the transitivity rule has at most n^3 firings. Hence theorem 3 implies that the algorithm runs in $O(n^3)$ time.

It is possible to give a sub-transitive flow algorithm analogous to the rules in figures 5 which runs in linear time under the assumption that the input expression is well typed and that every type expression has bounded size [10]. However, the sub-transitive version of figure 5 is beyond the scope of this paper.

6 Algorithms Based on Union-Find

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 dramatically improvement in empirical performance due to equality reported by Fahndrich et al. in [7]. 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 4 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 $\texttt{UNION}(u, w)$, $\texttt{UNION}(s, w)$, and $\texttt{UNION}(u, s)$ 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 4. *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 3. 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 Basic Union-Find Examples

Figure 6 gives a unification algorithm. The essence of the unification problem is that if a pair $\langle s,\ t\rangle$ is unified with $\langle u,\ w\rangle$ then one must recursively unify s with u and t with w. The rules guarantee that if $\langle s,\ t\rangle$ is equivalent to $\langle u,\ w\rangle$ then s and u are both equivalent to the term $\Pi_1(f)$ where f is the common find of the two pairs. Similarly, t and w must also be equivalent. So the rules compute the appropriate equivalence relation for unification. However, the rules do not detect clashes or occurs-check failures. This can be done by performing appropriate linear-time computations on the final find map.

To analyze the running time of the rules in figure 6 we first note that the rules maintain the invariant that all find values are terms appearing in the input problem. 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

$$\frac{\texttt{EQUATE!}(x,\ y)}{\texttt{UNION}(x,\ y)} \qquad \frac{\texttt{FIND}(\langle x,\ y\rangle, f)}{\texttt{UNION}(\Pi_1(f),\ x),\ \ \texttt{UNION}(\Pi_2(f),\ y)}$$

Fig. 6. A unification algorithm. The algorithm operates on "simple terms" defined to be either a constant, a variable, or a pair of simple terms. The input database is assumed to be a set of assertions of the form `EQUATE!`$(s,\ w)$ where s and w are simple terms. The rules generate the appropriate equivalence relation for unification but do not generate clashes or occurs-check failures (see the text). Because `UNION`$(x,\ y)$ selects y as the find value when both arguments have singleton equivalence classes, these rules maintain the invariant that all find values are terms in the original input.

$$\frac{\texttt{EQUATE!}(x,\ y)}{\texttt{INPUT}(x),\ \ \texttt{INPUT}(y)} \qquad \frac{\texttt{INPUT}(x)}{\texttt{ID-OR-FIND}(x,\ x)}$$

$$\frac{\texttt{INPUT}(\langle x,\ y\rangle)}{\texttt{INPUT}(x),\ \texttt{INPUT}(y)} \qquad \frac{\texttt{FIND}(x,\ y)}{\texttt{ID-OR-FIND}(x,\ y)}$$

$$\frac{\texttt{EQUATE!}(x,\ y)}{\texttt{UNION}(x,\ y)} \qquad \frac{\begin{array}{l}\texttt{INPUT}(\langle x,\ y\rangle)\\ \texttt{ID-OR-FIND}(x,\ x')\\ \texttt{ID-OR-FIND}(y,\ y')\end{array}}{\texttt{UNION}(\langle x',\ y'\rangle,\ \langle x,\ y\rangle)}$$

Fig. 7. A congruence closure algorithm. The input database is assumed to consist of a set of assertions of the form `EQUATE!`$(s,\ w)$ and `INPUT`(s) where s and w are simple terms (as defined in the caption for figure 6). As in figure 6, all find values are terms in the original input.

$$\frac{\texttt{INPUT}((f,\ u))}{\texttt{INPUT}(f),\ \ \texttt{INPUT}(u)}$$

$$\frac{\texttt{INPUT}(\lambda x.u)}{\texttt{INPUT}(u)}$$

$$\frac{\texttt{INPUT}((f,\ u))}{\begin{array}{l}\texttt{UNION}(\texttt{DOM}(\texttt{TYPE}(f)),\ \texttt{TYPE}(u))\\ \texttt{UNION}(\texttt{RAN}(\texttt{TYPE}(f)),\ \texttt{TYPE}((f\ u)))\end{array}}$$

$$\frac{\texttt{INPUT}(\lambda x.u)}{\begin{array}{l}\texttt{UNION}(\texttt{TYPE}(x),\ \texttt{DOM}(\texttt{TYPE}(\lambda x.u)))\\ \texttt{UNION}(\texttt{TYPE}(u),\ \texttt{RAN}(\texttt{TYPE}(\lambda x.u)))\end{array}}$$

Fig. 8. Type inference for simple types. The input database is assumed to consist of a single assertion of the form `INPUT(e)` where e is closed term of the pure λ-calculus and where distinct bound variables have been α-renamed to have distinct names. As in the case of the unification algorithm, these rules only construct the appropriate equivalence relation on types. An occurs-check on the resulting equivalence relation must be done elsewhere.

$O(n)$ 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 4 now implies that the closure can be computed in $O(n \log n)$ time. The best known unification algorithm runs in $O(n)$ time [21] 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 4 to the rules of figure 6 yields a slightly worse running time for what is, perhaps, a simpler presentation.

Now we consider the congruence closure algorithm given in figure 7. First we consider its correctness. The fundamental property of congruence closure is that if s is equivalent to s' and t is equivalent to t' and the pairs $\langle s,\ t\rangle$ and $\langle s',\ t'\rangle$ both appear in the input, then $\langle s,\ t\rangle$ should be equivalent to $\langle s',\ t'\rangle$. This fundamental property is guaranteed by the lower right hand rule in figure 7. This rule guarantees that if $\langle s,\ t\rangle$ and $\langle s',\ t'\rangle$ both occur in the input and s is equivalent to s' and t to t' then both $\langle s,\ t\rangle$ and $\langle s',\ t'\rangle$ are equivalent to $\langle f_1,\ f_2\rangle$ where f_1 is the common find of s and s' and f_2 is the common find of t and t'. So the algorithm computes the congruence closure equivalence relation.

To analyze the complexity of the rules in figure 7 we first note that, as in the case of unification, 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 $\mathtt{FIND}(s,\ f)$ 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 4 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.

We leave it to the reader to verify that the inference rules in figure 8 define the appropriate equivalence relation on the types of the program expressions and that the types can be constructed in linear time from the find relation output by the procedure. It is clear that the inference rules generate only $O(n)$ union operations and hence the closure can be computed in $O(n \log n)$ time.

8 Henglein's Quadratic Algorithm

We now consider Henglein's quadratic time algorithm for determining typability in a variant of the Abadi-Cardelli object calculus [12]. This algorithm is interesting because the first algorithm published for the problem was a classical dynamic transitive closure algorithm requiring $O(n^3)$ time [19] and because Henglein's presentation of the quadratic algorithm is given in classical pseudo-code and is fairly complex.

$$\frac{\tau \leq \sigma}{\tau \rightarrow \sigma,\quad \tau \Rightarrow \sigma}$$

$$\frac{\sigma \Rightarrow \tau,\quad \tau \rightarrow \gamma}{\sigma \rightarrow \gamma}$$

$$\frac{\mathtt{EQUAL}(\tau,\ \sigma)}{\mathtt{UNION}(\tau,\ \sigma)}$$

$$\frac{\begin{array}{l}\mathtt{ACCEPTS}(\tau,\ \ell)\\ \mathtt{ACCEPTS}(\sigma,\ \ell)\\ \sigma \rightarrow \tau\end{array}}{\mathtt{UNION}(\sigma.\ell,\ \tau.\ell)}$$

$$\frac{\mathtt{MERGE}(\tau,\ \sigma)}{\begin{array}{l}\tau \rightarrow \sigma,\quad \tau \Rightarrow \sigma\\ \sigma \rightarrow \tau,\quad \sigma \Rightarrow \tau\end{array}}$$

Fig. 9. Henglein's type inference algorithm.

A simple union-find rule set for Henglein's algorithm is given in figure 9. First we define type expressions with the grammar $\sigma ::= \alpha \mid [\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n]$

where α represents a type variable and $\ell_i \neq \ell_j$ for $i \neq j$. Intuitively, an object o has type $[\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n]$ if it provides a slot (or field) for each slot name ℓ_i and for each such slot name we have that the slot value $o.\ell_i$ of o for slot ℓ_i has type σ_i. The algorithm takes as input a set of assertions (type constraints) of the form $\sigma_1 \leq \sigma_2$ where σ_1 and σ_2 are type expressions. We take [] to be the type of the object with no slots. Note that, given this "null type" as a base type, there are infinitely many closed type expressions, i.e., type expressions not containing variables. The algorithm is to decide whether there exists an interpretation γ mapping each type variable to a closed type expression such that for each constraint $\sigma_1 \leq \sigma_2$ we have that $\gamma(\sigma_1)$ is a subtype of $\gamma(\sigma_2)$. The subtype relation is taken to be "invariant", i.e., 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.

The rules in figure 9 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 `EQUAL`$([\ell_1 = \sigma_1; \ldots; \ell_n = \sigma_n].\ell_i,\ \sigma_i)$ with $1 \leq i \leq n$. Note that this preprocessing can be done in linear time. The invariance property of the subtype relation justifies the final rule (lower right) in figure 9. A system of constraints is rejected if the equivalence relation forces a type to be a subexpression of itself, i.e., an occurs-check on type expressions fails, or the final database contains `ACCEPTS`$(\sigma,\ \ell)$ and $\tau \to \sigma$, but not `ACCEPTS`$(\tau,\ \ell)$.

To analyze the complexity of the algorithm in figure 9 note that all terms involved in the equivalence relation are type expressions appearing in the processed input — each such expression is either a type expression of the original unprocessed input or of the form $\sigma.\ell$ where σ is in the original input and ℓ is a slot name appearing at the top level of σ. Let n be the number assertions in the processed input. Note that the preprocessing guarantees that there is at least one input assertion for each type expression so the number of type expressions appearing in the input is also $O(n)$. Since there are $O(n)$ terms involved in the equivalence relation the rules can generate at most $O(n)$ `MERGE` assertions. This implies that the rules generate only $O(n)$ assertions of the form $\sigma \Rightarrow \tau$. This implies that the number of prefix firings is $O(n^2)$. Since there are $O(n)$ terms involved in the equivalence relation there are $O(n \log n)$ `FIND` assertions in the closure. Theorem 4 now implies that the running time is $O(n^2 + n \log n) = O(n^2)$.

9 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 $\texttt{NOT}(\texttt{ACCEPTS}(\sigma,\ \ell))$ in Henglein's algorithm while preserving the truth of theorems 3 and 4. 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. 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.
2. David Basin and Herald Ganzinger. Automated complexity analysis based on ordered resolution. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 456–465. IEEE Computer Society Press, 1996.
3. A. Bondorf and A. Jorgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of functional Programming*, 3(3), 1993.
4. Keith L. Clark. Logic programming schemes and their implementations. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.
5. William Downing and Jean H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.
6. Jason Eisner and Giorgio Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *ACL-99*, pages 457–464, 1999.
7. Manual Fähndrich, Jeffrey Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI98*, 1998.
8. Robert Givan and David McAllester. New results on local inference relations. In *Principles of Knowlwedge Representation and Reasoning: Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, October 1992. internet file ftp.ai.mit.edu:/pub/users/dam/kr92.ps.
9. N. Heintze. Set based analysis of ml programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
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. R. A. Kowalski. Predicate logic as a programming language. In *IFIP 74*, 1974.
14. Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *J. Comput. Syst. Sci.*, 49(2):306–324, October 1994.
15. D. McAllester. Automatic recognition of tractability in inference relations. *JACM*, 40(2):284–303, April 1993. internet file ftp.ai.mit.edu:/pub/users/dam/jacm2.ps.
16. David McAllester. Inferring recursive types. Available at http://www.research.mit.edu/~dmac, 1996.
17. E. Melski and T. Reps. Intercovertability of set constraints and context free language reachability. In *PEPM'97*, 1997.
18. Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.
19. J. Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.
20. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *POPL95*, pages 367–378, 1995.
21. M. S. Paterson and M. N. Wegman. Linear unification. *JCSS*, 16:158–167, 1978.
22. 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.
23. M. Vardi. Complexity of relational query languages. In *14th Symposium on Theory of Computation*, pages 137–146, 1982.