Estimating the Computational Cost of Logic Programs

S. K. Debray,* P. López García,† M. Hermenegildo,† N.-W. Lin‡

Abstract

Information about the computational cost of programs is potentially useful for a variety of purposes, including selecting among different algorithms, guiding program transformations, in granularity control and mapping decisions in parallelizing compilers, and query optimization in deductive databases. Cost analysis of logic programs is complicated by nondeterminism: on the one hand, procedures can return multiple solutions, making it necessary to estimate the number of solutions in order to give nontrivial upper bound cost estimates; on the other hand, the possibility of failure has to be taken into account while estimating lower bounds. Here we discuss techniques to address these problems to some extent.

1 Introduction

Information about the computational cost of a program is potentially useful for a variety of purposes. Programmers can use such information, possibly obtained manually, to choose between different algorithmic solutions to a problem. Program transformation systems can use cost information to choose between alternative transformations. Parallezing compilers can use cost estimates for "granularity control," which tries to balance the overheads of task creation and manipulation against the benefits of additional parallelism. Information about message sizes and relative frequency of communication between different processes can be used to improve task mapping decisions on distributed memory systems. Information about the number of solutions in deductive database systems can be used for query optimization purposes. Apart from these applications

*Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA. Email: debray@cs.arizona.edu

†Department of Artificial Intelligence, Universidad Politecnica de Madrid, E-28600 Madrid, Spain. Email: pedro@dia.fi.upm.es, herme@dia.fi.upm.es

‡Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, 62107, Taiwan, R.O.C. Email: naiwei@cs.ccu.edu.tw

of cost information, the problem of cost analysis may be of some independent interest to researchers on static analysis of logic programs because (*i*) it uses a great deal of information from other kinds of analyses, such as mode and type analysis, inference of size norms, etc., so that any improvements in these analyses potentially yield improvements in cost analysis; and (*ii*) because of the rich variety of algorithms for combinatorial analysis that arise, especially when dealing with constraints. Here we discuss some of our work to date on (semi-)automatic worst-case upper and lower bound cost analysis for logic programs.

Cost analysis of functional and imperative programs has been studied by a number of researchers. A major difference between logic programming languages and other traditional languages in this regard is that logic programs are nondeterministic in general, and may produce multiple solutions, making it necessary to estimate their number to give nontrivial upper bound cost estimates. A related problem is that failure of execution is not an abnormal situation, and implicit failures have to be accounted for and dealt with explicitly if meaningful results are to be obtained. For example, in the following program to check membership of an element in a list, a naive analysis that does not take implicit failure into account will not have a base case from which to reason about its cost:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

Failure also poses problems for lower bound analyses, since any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0.

The work described here had its origins in discussions on task granularity analysis for parallel logic programs. We hoped, initially, to obtain techniques for lower bound cost estimation for arbitrary (pure) Prolog programs, but it soon became clear that on the one hand, nontrivial lower bounds were difficult to obtain because of the possibility of failure; and on the other hand, upper bound cost estimation for nondeterministic programs was difficult without information about the number of solutions a predicate could generate, which we didn't know how to estimate. We were forced to scale back our expectations, therefore, and the work reported in [5] dealt only with upper bound cost estimation for deterministic programs. Subsequently we were able to make some progress on estimating (upper bounds on) the number of solutions a procedure might produce [4]. This resulted in a framework for upper bound analysis of logic programs, discussed in Section 3 and detailed in [3, 8]. Recently, we have made some progress on our original goal of lower bound cost estimation: these ideas are discussed in Section 4, with details in [6]. Together, this provides a framework for reasoning about the computational cost of a reasonably large class of logic programs, though there is a great deal of scope for improvements in this area (a couple of directions for future research are discussed in Section 5).

2 The Overall Approach

In general, the cost of a procedure depends on (some measure of) the size of its input. Therefore, it is necessary to keep track of the sizes of arguments to procedures at each program point (procedure entry and exit). In addition, in order to handle nondeterministic procedures, knowledge about the number of solutions generated by each predicate is required. Not unexpectedly, the size relationships between arguments, the number of solutions, and the time complexity functions for recursive procedures are obtained in the form of difference equations. To get closed form expressions for the cost of a procedure, we have to obtain (possibly approximate) solutions to these equations.

Our approach to cost analysis of logic programs can be summarized as follows:

1. Carry out mode and data dependency analysis to identify input and output arguments to procedures and dependencies between producers and consumers of variable bindings.
2. Determine which portions of a program can be executed and therefore should be considered during cost analysis. For a worst case upper bound analysis, this step is generally trivial in that all of the program is considered to be reachable; however, for lower bound analysis, it is necessary to identify program fragments where failure can be ruled out. This may require auxiliary information, e.g., types.
3. Use data dependency information to obtain size relationships between variables in a clause. Use this to infer size relationships between input and output arguments of procedures.
4. Use argument size information to determine how many solutions a procedure can produce.
5. Use information about argument sizes and number of solutions produced by various procedures to obtain estimates of their computational costs.

The call graph of a program is a directed graph whose vertices correspond to the predicates in the program, and where there is an edge from a vertex p to a vertex q if there is a literal with predicate symbol q in the body of a clause defining p. Because of recursion, the call graph may contain cycles, but if we construct a derived graph whose vertices correspond to the strongly connected components of the call graph and where there is an edge from a vertex p to a vertex q if there is an edge from some predicate in the strong component of p to some predicate in the strong component of q, the resulting graph is acyclic. The various components of cost analysis (argument size, number of solutions, computational cost) all proceed by traversing the graph of strongly connected components of the call graph of the program in topological order starting at the leaves. At each stage, therefore, we can assume that the costs of predicates that are "lower" in the graph have been determined. To handle recursion, we symbolically represent the cost (be it size, number of solutions, or computational complexity) of the predicate being analyzed as a function of its input sizes. For recursive clauses, this expresses the cost for a particular input size (corresponding to the head of the clause) in terms of the costs for smaller inputs (corresponding to the recursive calls in the body), which can be simplified to obtain difference equations. Base cases for these difference equations are provided by the non-recursive clauses (or have to be obtained separately in the case of implicit failure as in the `member/2` predicate described earlier). By solving these equations we get an expression for the cost of the predicate under consideration.

3 Upper Bound Analyses

In worst case upper bound analyses, we assume that all goals succeed (i.e., produce at least one solution) and that all solutions computed by a predicate are needed.

3.1 Argument Sizes and Space Complexity

In general, the size of the outputs produced by a predicate depends on the size of its inputs. To simplify the analysis, we assume that the size of each output argument of a predicate is a function of the sizes of its input arguments. This works reasonably well in general, but tends to be overly conservative for divide-and-conquer programs.

Argument sizes are computed using data dependencies. Intuitively, a data dependency is a binary relation between the different argument positions in a clause (the latter may be specified, for example, by a pair of integers indicating a literal in the clause and an argument within that literal): there is a data dependency from a position a to a position b if a value defined (i.e., computed) at a is used at b.

Various measures can be used to determine the "size" of an input value, e.g., term-size, term-depth, list-length, integer-value, etc. The measure(s) appropriate in a given situation can in most cases be determined by examining the types of argument positions, the general idea being to use the "back edges" in the type graph of a predicate to determine how that predicate recursively traverses its input terms (or constructs its output terms), and thereby synthesize an appropriate measure for the predicate [7]. If we start with a program where clauses have been rewritten so that each argument in each literal is a variable, with new unification goals introduced where necessary, then for each variable x we can set up equations that specify its size in terms of the sizes of other variables and, possibly, symbolic representations of the output sizes for recursive procedures. For example, for the clause

$$
\begin{array}{l}
nrev(x_1, x_2) \ :- \\
\quad x_1 = \texttt{[}y_1 \,|\, y_2\texttt{]},\ nrev(y_2, z_1),\ z_2 = \texttt{[}y_1\texttt{]},\ append(z_1, z_2, x_2)
\end{array}
$$

assuming that $nrev/2$ has its first argument as an input argument and its second as an output argument while for $append/3$ the first two arguments are inputs and the third is an output argument, and that the size of each argument is given by the "list length" function, we get the following equations

$$
\begin{array}{lll}
x_2 = size\_nrev(x_1) & y_2 = x_1 - 1 & z_1 = size\_nrev(y_2) \\
x_2 = size\_append(z_1, z_2) & z_2 = 1 &
\end{array}
$$

These equations can be simplified to obtain size expressions for the output arguments of the procedure. For recursive clauses this yields a difference equation that expresses the output size for a given input size in terms of the output size for smaller inputs. For example, suppose that while processing the call graph in topological order, we solve the difference equations obtained for the output argument size of $append/3$ to get $size\_append(x, y) = x + y$. Then, for the clause for $nrev/2$ given above, we get the difference equation

$$size\_nrev(x_1) = size\_nrev(x_1 - 1) + 1.$$

3.2 Number of Solutions

To estimate the number of solutions a predicate can return, we estimate the number of bindings possible for each variable in the clause. We use two simple rules for this:

(i) If a variable has k occurrences in a clause, and the number of (ground) bindings possible for these occurrences are estimated as $n_1, \ldots, n_k$, then the number of bindings possible for the variable is (at most) $min(n_1, \ldots, n_k)$.

(*ii*) If a variable x is bound to a term containing a set of variables V, and for each $v \in V$ the number of bindings possible for v is given by n_v, then the number of bindings possible for x is (at most) $\prod_{v \in V} n_v$.

The second of these rules is fairly conservative in that it assumes that all variable bindings are independent, so that all possible combinations of bindings for the different variables are possible. It is possible to improve this rule to take dependences between variables into account; details are given in [3].

These rules work reasonably well for pure Horn clauses. We augment them with two approximation algorithms for dealing with arithmetic and finite-domain constraints. The first deals with linear binary constraints. Such constraint satisfaction problems can be represented as a graph where each vertex represents an assignment of a value to a variable, and where there is an edge between two vertices if the corresponding value assignments are consistent with each other. Solving a set of constraints over n variables then corresponds to finding a clique of size n in such a graph, and the number of solutions is given by the number of such cliques. A direct approach to counting the number of cliques in a consistency graph can require exponential time. Instead, we obtain upper bound approximations to the number of cliques by repeatedly simplifying a (weighted version of) the consistency graph until we obtain a bipartite graph. The number of 2-cliques in a bipartite weighted consistency graph is simply the sum of the weights of its edges, and this can be shown to provide an upper bound on the number of n-cliques in the original graph. For a set of constraints involving n variables and m domain values, the overall worst case time complexity of this algorithm is $O(n^3m^3)$.

The second approximation algorithm deals with equality and disequality constraints over a finite domain. It can be shown that the problem of estimating the number of solutions to a set of such constraints can be transformed into the problem of computing the chromatic polynomial of a graph. Since the problem of determining the chromatic number of a graph is NP-complete, that of determining chromatic polynomials is NP-hard. However, it turns out that if we can efficiently compute a lower bound on the chromatic number of a graph, then we can efficiently compute an upper bound on the chromatic polynomial of a graph. We use a result by Bondy [1] to obtain a lower bound on the chromatic number of a graph. For a set of m constraints involving n variables, this yields a procedure for computing an upper bound estimate on the number of solutions having a worst case complexity of $O(n^2 \log n + nm)$.

For estimating the number of solutions for predicates in a program, we associate with each predicate a pair of values: one of these is an upper bound on the relation size for the predicate (for recursive predicates this is infinity), and the other is a function that gives an upper bound on the number of solutions that may be obtained for a single input of given size. We combine the algorithms described above as follows. When type information is available for a predicate, each of its clauses is first checked to see if it can be unfolded into a conjunction of binary disequality constraints where the variables range over the same finite set of constants. In this case, the constraint graph is constructed and the algorithm for estimating the chromatic polynomial of a graph is utilized to estimate the number of solutions possible for those variables. Otherwise, the clause is checked to see if it can be unfolded into a conjunction and/or disjunction of linear constraints over a finite domain. In this case, the algorithm for estimating the number of n-cliques of a consistency graph is employed to estimate the number of bindings possible. In other cases, the general algorithm is used. As in the case of size relationships, recursive literals are handled by using symbolic expressions to denote the number of solutions generated by them, and solving (or giving upper bound estimates to) the resulting difference equations.

The number of solutions a predicate can generate is the maximum of the number of solutions that can be generated by each mutually exclusive cluster of clauses (see [2] for a discussion of inference of mutual exclusion); the number of solutions any cluster can generate is bounded by the sum of the number of solutions that can be generated by each clause within the cluster.

3.3 Time Complexity

The worst-case upper bound time complexity of a clause, for a single procedure call to that clause, can be obtained as the time taken for head unification together with the time to execute its body.

If we express the time complexity in terms of the number of resolution steps, or procedure calls, then head unification involves just a single resolution. If complexity is expressed in terms of the number of unification operations, then the cost of head unification is given by the number of arguments. If we want to estimate the number of instructions executed, then we have to examine the unification algorithm being used in detail, to obtain a precise expression for its worst-case cost for inputs of a given size, and use this to express the cost of head unification. An intermediate solution is to "peel" head unification to "normal form," i.e., represent it as a sequence of atomic Herbrand domain constraints and count the number of such constraints, or even estimate the cost of each of them. Mode and type information can help estimate such cost.

The cost of executing the body of the clause can be obtained from the costs of executing each body literal (as mentioned earlier, recursive literals are handled using symbolic representations). The input size for each body literal is obtained as a function of the input sizes for the clause head from argument size analysis. Number of solutions analysis is used to determine how many times each body literal is executed: given Prolog's left-to-right execution strategy, for example, the number of times a body literal is executed is (bounded above by) the product of (upper bounds on) the number of solutions produced by the literals to its left.

As in the case of estimating the number of solutions, the clauses are partitioned into mutually exclusive clusters. The time complexity for each such cluster can be obtained by summing the time complexity for each of its clauses. In addition to that, however, we also need to take into account the failure cost introduced by trying to solve the clauses in other clusters. The failure cost from solving a clause in another cluster can be estimated by considering the sources leading to the mutual exclusion among clauses. This information can be easily produced by mutual exclusion analysis [2]. After the failure costs are added into the time complexity for each cluster, the time complexity of a predicate is then obtained as the maximum of the time complexities of these clusters.

4 Lower Bound Cost Analyses

The main problem with the inference of lower bounds on the computational cost of logic programs is the possibility of failure of execution. Any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0. An obvious solution would be to try and rule out "bad" argument values by considering the types of predicates. However, most existing type analyses provide upper approximations, in the sense that the type of a predicate is a superset of the set of argument values that are actually encountered at runtime. Unfortunately, straightforward attempts to address this issue, for example by trying to infer lower approximations to the calling types of predicates, fail to yield nontrivial lower bounds for most cases. We take a different approach where, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail, using the notion of a set of tests "covering" the type of a variable.

4.1 Coverings and Non-Failure Analysis

The basic idea behind the notion of covering is very simple. We can think of a test $\tau(x)$ as denoting a set of terms $succ(\tau(x))$, namely, the terms for which the test succeeds (tests that take more than one argument can be thought of as unary predicates operating on tuples of the appropriate size). This extends in the obvious way to sets of tests: a set of tests $S = \{\tau_1(x), \ldots, \tau_n(x)\}$, which represents the disjunction $\tau_1(x) \vee \cdots \vee \tau_n(x)$, denotes the set of terms $succ(S) = \cup_{i=1}^{n} succ(\tau_i(x))$. Now suppose that the variable x has type T, where a type intuitively denotes a set of terms. If $T \subseteq succ(S)$, then it must be the case that for any possible value that x can take on (this value must lie in the set of terms T) at least one of the tests in the set S will succeed. In this case, we say that the set of tests S *covers* the type T of x. The idea can be generalized easily to type assignments on multiple variables.

We can think of each clause in a program as consisting of an "input test", which is a conjunction of head unifications and tests on the input arguments, followed by a sequence of output unifications and calls to other predicates. The basic idea behind our approach is to determine whether the set of input tests of a predicate cover the type of its input arguments: if it does, we can guarantee that at least one of these tests will succeed for any call to that predicate, and therefore that if such a call fails it must be due to the failure of a body goal. Information about the possible failure of body goals is obtained by processing the strongly connected components of the call graph in topological order.

The main technical problem here, then, is that of determining whether a set of tests covers a type assignment. It turns out that in the presence of arbitrary arithmetic operations the problem is undecidable in general, even if the set of tests under consideration is a singleton (the proof is a straightforward reduction from Matijasevič's proof of the unsolvability of Hilbert's tenth problem [9]), and is co-NP-hard even if we restrict ourselves to finite types. We therefore have to resort to sound (but obviously incomplete) algorithms for checking coverings. A linear time algorithm for this is described in [6].

Using the notion of coverings, it is straightforward to identify the non-failing goals and predicates in a program. This simply involves a depth-first traversal of a graph derived from the the call graph of the program, starting from literals whose types are not covered by the input tests of the called predicate, marking each visited node (literal as well as predicate) as "possibly failing." When this is over the unmarked predicates and literals are guaranteed to be non-failing.

4.2 Argument Size

After non-failing goals have been identified, lower-bound argument size analysis proceeds essentially as described in Section 3.1, with two obvious differences: first, the output sizes for a clause that may fail are 0;[1] and second, the output size of a predicate is obtained by taking the *min* of the output sizes of its clauses.

[1] This works because we consider only the non-failing literals to the left of the first possibly-failing literal when estimating the computational cost of a clause (see Section 4.4).

4.3 Number of Solutions

It is tempting to try and estimate a lower bound on the number of solutions generated by a clause '$H\ :-\ B_1, \ldots, B_n$' from lower bounds on the number of solutions generated by each of the body literals B_i, in a manner analogous to the estimation of upper bounds on the number of solutions. Unfortunately, this does not work. For example, given a clause '$p(X)\ :-\ q(X), r(X)$', where X is an output variable, suppose that q and r generate n_q and n_r bindings, respectively, for X, then $min(n_q, n_r)$ is not a lower bound on the number of solutions the clause can generate. To see this, suppose that q can bind X to either a or b, while r can bind X to either b or c: thus, $min(n_q, n_r) = min(2, 2) = 2$, but the number of solutions for the clause is 1. The problem in this case is that the goals $q(X)$ and $r(X)$ "interfere" with each other in terms of the bindings they allow for X. We can give more restrictive rules that essentially rule out such interference, for example by requiring that output variables be distinct and unaliased and occur at most once in the clause body. The approach can be extended to handle equality and disequality constraints by computing a lower bound on the chromatic polynomial of the associated graph.

Once we have computed lower bounds on the number of solutions a single clause can yield, we can estimate lower bounds on the number of solutions produced by a set of clauses. In general, this is given by the *min* of the number of solutions due to the individual clauses. However, if we can show that a set of clauses have "equivalent" input tests, so that if one of them succeeds for a call then all of them do, then we can improve our lower bound estimate to be the sum of the lower bounds of the individual clauses in that set.

These restrictions may seem serious, but they nevertheless allow us to infer interesting lower bounds for a reasonably large class of programs. For example, given the program

```
:- mode subset(in, out).
subset([], X) :- X = [].
subset([H|L], X) :- X = [H|X1], subset(L, X1).
subset([H|L], X) :- subset(L, X).
```

we can infer that given an input list of length n, this predicate produces at least 2^n solutions.

4.4 Computational Cost

Once information about non-failure and argument sizes has been computed, the estimation of lower bounds on the computational cost of a predicate is not difficult. If we cannot guarantee that all solutions to a predicate are needed, then the cost of a clause is at least the cost of the input tests together with the sum of the costs of the body literals upto the leftmost "possibly failing" goal. In contexts where all solutions are required, e.g., within a `setof` or in a distributed implementation, this can be improved by taking the number of solutions into account to estimate how many times each of these literals must be executed. The cost of a predicate is at least the *min* of the costs of its clauses. This estimate can be improved with knowledge about the order in which clauses are tried and about the indexing scheme used.

5 Directions for Future Work

There are many directions in which the work described can be extended: here we consider just two. A significant shortcoming of our current approach is that the size of each output argument is treated as a function of the input sizes, independent of the sizes of other output arguments. As a result, relationships between the sizes of different output arguments are lost. This, in turn, can cause a significant loss in precision, especially in divide-and-conquer programs. For example, in a quicksort program where a list of numbers is split into two lists that are recursively sorted, our approach determines that given an input list of length n, each of the lists obtained from the splitting can have length $n-1$ in the worst case: the information that their lengths must also sum to $n-1$ is lost. Because of this, we infer an exponential upper bound on the time complexity of the quicksort program: this is sound, but somewhat less precise than we would like. It may be possible to alleviate this problem by using constraint-based reasoning both for expressing output sizes in terms of input sizes, and also for expressing relationships between the sizes of different variables in a clause. On the other hand, as a pragmatic short-term solution it may be possible to get a lot of mileage simply by identifying and treating divide-and-conquer programs specially.

Another interesting area of investigation is average case analysis. For most of the applications identified at the beginning of this paper, the average cost of a program is far more interesting, and appropriate, than the worst case. Obviously, giving an acceptable definition of "average" requires defining a probability distribution on the possible inputs, and this seems nontrivial. However, one can imagine that profiling techniques might be usable for estimating input distributions, so techniques for average case analysis that assume that the input distributions are given are worth investigating.

Acknowledgements

The work of S. Debray and N.-W. Lin was supported in part by the National Science Foundation under grant CCR-9123520. The work of M. Hermenegildo and P. López was supported in part by ESPRIT project 6707/TIC93-0976-CE "PARFORCE" and CICYT proyect number TIC93-0737-C02-01.

References

[1] J. A. Bondy, "Bounds for the Chromatic Number of a Graph," *Journal of Combinatorial Theory* 7, (1969), pp. 96–98.

[2] S. K. Debray and D. S. Warren, "Functional Computations in Logic Programs," *ACM Transactions on Programming Languages and Systems* 11, 3 (July 1989), pp. 451–481.

[3] S. K. Debray and N.-W. Lin, "Cost Analysis of Logic Programs", *ACM Transactions on Programming Languages and Systems*, vol. 15 no. 5, Nov. 1993, pp. 826–875.

[4] S. K. Debray and N. Lin, "Static Estimation of Query Sizes in Horn Programs," *Proc. Third International Conference on Database Theory*, Paris, France, December 1990, pp. 514–528.

[5] S. K. Debray, N.-W. Lin, and M. Hermenegildo, "Task Granularity Analysis in Logic Programs", *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pp. 174–188. ACM Press, June 1990.

[6] S. K. Debray, P. López García, M. Hermenegildo, and N.-W. Lin, "Lower Bound Cost Estimation for Logic Programs", manuscript, April 1994.

[7] S. Decorte, D. De Schreye, and M. Fabris, "Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis", *Proc. 1993 International Symposium on Logic Programming*, 1993, pp. 420–436. MIT Press.

[8] N.-W. Lin, *Automatic Complexity Analysis of Logic Programs*, Ph.D. Dissertation, The University of Arizona, 1993.

[9] Ju. V. Matijasevič, "Enumerable Sets are Diophantine", *Doklady Akademii Nauk SSSR*, 191 (1970), 279-282 (in Russian; English translation in *Soviet Mathematics—Doklady*, 11 (1970), 354-357).