Probabilistic Cost Analysis of Logic Programs: A First Case Study

Héctor Soza Pollman
Departamento de Ingeniería de Sistemas y Computación
Universidad Católica del Norte
Av. Angamos 0610, Casilla 1280
Antofagasta, Chile
e-mail: hsoza@ucn.cl

and

Manuel Carro, Pedro López García
Facultad de Informática
Universidad Politécnica de Madrid
Boadilla del Monte
E-28600 Madrid, Spain
e-mail: {mcarro,pedro.lopez}@fi.upm.es

Abstract

Several cost analyses of logic programs have been developed which make it possible to automatically obtain lower and upper bounds of runtime cost of computations. This information is very useful for a variety of purposes, including granularity control for parallel execution, query optimizations in databases, and program transformation and synthesis. However, current techniques suffer a loss accuracy in some quite representative cases (i.e., some divide-and-conquer programs *à la* QuickSort). This paper describes an alternative probabilistic approach which makes it possible to figure out a function estimating program execution cost. Estimations deduced in this fashion did not contradict correct upper and lower bounds in all example cases tried out. At the same time, the accuracy of the cost prediction in programs where bound-based techniques were clearly overestimating the real cost was greatly improved and exact complexity expressions could obtained both for average and for extreme cases. Additionally, one of the advantages of the proposed method is that it needs only a few changes over previous schemes.

Keywords: Logic Programming, Cost Analysis, Complexity Analysis, Program analysis, Resource Consumption Estimation.

1 Introduction

Computation cost is, in general, some measure of resources which a computation needs to consume in order to proceed satistactorily. Usual cost measures are time, computation steps, memory spent, etc. In this work we will deal (in a somewhat abstract fashion) with costs which increase monotonically with the execution, exemplified by time / number of computation steps. Information about the runtime cost of computations can be useful for a variety of applications. For example, it is useful for task scheduling and granularity control (i.e., dynamic control of thread / process creation) in parallel execution [15, 5, 19, 12, 17, 14], program transformation, query optimization in databases [3], resource-aware security in mobile code [10, 9], and to prove that a program meets strict time constraints in real-time systems [1].

In the context of logic programming, the work on cost estimation has generally focused on upper [4] or lower [6] bounds cost analysis. A significant shortcoming of the techniques used to derive the cost estimation is their precision loss in the presence of the frequent case of divide-and-conquer programs in which the sizes of the output arguments of the "divide" part are not independent. In the familiar QuickSort program (see Figure 1), for example, since either of the outputs of the `partition/4` predicate are fed again as the input list to `qsort/2`, the upper bound cost

analysis approach [4] works under the assumption that both outputs can *simultaneously* be the whole input list, thereby significantly overestimating the cost of the QuickSort program (a more detailed account of why this is so can be found in Section 2.3). On the other hand, for most of the applications mentioned before, the average cost of a program is more interesting, and appropriate, than the worst or best case: while those give raise to "safe" decisions, the cost of not performing, e.g., parallel execution when possible in average, might not be balanced by the occasions in which this is done safely. Obviously, giving an acceptable definition of "average" requires defining a probability distribution on the possible inputs which matches the expected runtime conditions, and this seems nontrivial — it is, at least, dependent on the final environment. However, profiling techniques might be usable for estimating input distributions. For these reasons, in this paper we propose a technique for probabilistic case analysis that assume that the input distributions have been previously set. We show how useful cost functions can be obtained and we apply it to a representant of the frequent case of divide-and-conquer programs. These cost functions yield the number of procedure calls (resolution steps) as a function on the size of input data, although they could be adapted to express the cost using other metrics: number of unifications, number of instructions executed, or even execution time [16].

An analysis of combinatorial structures, addressing the automatic average-case complexity analysis of algorithms, has been developed in [8], and it is heavily based on studying type properties. We know of no other work which estimates average cost functions for logic programs.

This paper is organized as follows: Section 2 introduces cost analysis related with term size calculation. Section 3 presents the definition of probabilistic cost and an example of its application. Section 4 discusses briefly the relationship between bounds and probabilistic cost and, finally, Section 5 gives some conclusions and discusses some further work.

2 Cost Analysis of Logic Programs

In our approach, program costs are expressed as functions on the sizes of the input arguments, which yield estimations of the number of execution steps required by the computation. This approach is similar to that of [4, 6], except that our cost functions produce costs associated to a probability distribution instead of upper / lower bounds. Various metrics can be used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc., depending on the particular case. We assume that types, modes, and size measures are available, as a result of a previous automatic analysis (e.g. using the `CiaoPP` system [11]) or by means of programmer-provided annotations, and are put to work when analyzing term sizes and predicate costs. We assume also that the cost unit is a resolution, i.e., a procedure call (or, accordingly, the successful unification of a clause head).

In order to obtain lower- and upper-bound approximations of cost functions, `CiaoPP` first performs the following analyses (all using abstract interpretation [13, 2] techniques):

- A mode (and sharing) analysis. This determines which arguments (or parts of them) are inputs and which are outputs for each procedure and procedure call, as well as the dependencies between any variables in the data structures passed via these arguments. The use of mode information implies that each argument of a predicate (denoted by its position in the head) is annotated as being purely "input" or "output", depending on whether or not it is bound to a ground term when the predicate is invoked.[1]

- A type analysis, which infers types for all program variables. Note that type declarations are not compulsory in the language, so the relevant type definitions may also have to be inferred.

- A determinacy analysis. It needs the results of type and mode analysis, and it detects which procedures and procedure calls are deterministic (i.e., they will yield a single solution).

- A non-failure analysis. This also requires results from type and mode analysis, and can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not to terminate. Non-failure analysis is necessary to estimate accurately lower bounds, as it is necessary to account for the possibility of failure of a call to a procedure, leading to a trivial lower bound of 0.

- Inference of size metrics for relevant arguments (which we will detail in short). This is based on type information.

[1] Note that this is a restriction with respect to the more general case of Logic Programming, where data structures can be "incomplete", i.e., partially instantiated.

```
qsort([],[]).
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).

append([],X,X).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :-
        E >= C,
        partition(R,C,Left,Right1).
```

Figure 1: QuickSort in Prolog

We will sketch in what follows how the work done by (recursive) clauses is determined once all the above information is obtained. To this end, it is first necessary to estimate the size of output arguments in the procedure calls in the body of the procedure. The size of output arguments in a procedure call depends, in general, on the size of the input arguments in that call. For example, in the program in Figure 1, the size of the output argument of `append/3` depends on the size of its input arguments. Those input arguments are, in turn, the output arguments of the calls to `qsort/2`, and so on. Therefore, for each output argument we construct an expression which yields its size as a function of the size of the input arguments. This is done by means of abstractions of procedure definitions (a data dependency graph) built using all the information inferred previously. The following steps are then performed:

- Data dependency graphs are used to determine the relative sizes of variable bindings at different program points.
- Size information is used to set up difference equations representing the computational cost of procedures.
- Abstractions (lower and upper bounds) of the solutions of these difference equations are then obtained, which provide lower and upper-bounds on procedure costs and on the functions relating the size of input and output data.

We will use as driving example an implementation of QuickSort, shown in Figure 1, which sorts a list of terms.[2] The first argument of `qsort/2` is an input and the second is an output. The input arguments of `partition/4` are the first and the second ones —a list and a number, respectively. The output arguments of `partition/4` are the third and fourth ones. For the predicate `append/3`, the first and second arguments are inputs and the third argument is an output.

2.1 Lower and Upper Bounds

We will very briefly describe the basics of automatic deduction of lower and upper bounds for the computational cost of predicates. A more formal definition of a lower bound of the cost of a clause is given in [6]; similarly [4, 7] describe the method used to derive cost upper bounds. We will use in this section a clause C which we assume is of the form

$$C \quad \equiv \quad H : -B_1, ..., B_m \tag{1}$$

We assume that C has r input arguments, and we will represent by $\overline{x}$ the r-tuple denoting the sizes of these input arguments. Let $\phi_1(\overline{x}), ..., \phi_m(\overline{x})$ be the size(s) of the input argument(s) for each of the the body literals $B_1, ..., B_m$ respectively. We assume that the program is deterministic, and therefore each body goal has at most one solution. We will denote by $Cost_C(\overline{x})$ the cost of clause C for an input of size $\overline{x}$.

Lower Bounds Let $Cost_C^{inf}(\overline{x})$ be a lower bound of clause C, i.e., $Cost_C^{inf}(\overline{x}) \leq Cost_C(\overline{x})$. A non-trivial lower bound for the clause cost is:

$$Cost_C^{inf}(\overline{x}) = \gamma(\overline{x}) + \sum_{i=1}^{k} Cost_{B_i}^{inf}(\phi_i(\overline{x}))$$

where $\gamma(\overline{x})$ represents the cost of unification of the head and builtin tests (e.g., `E < C`) and k is the ordinal of the last non-failing body goal (literals from $k+1$ up to m are not reached in case goal $k+1$ fails). If the clause C corresponds

[2] Note that we are using the naive version which uses complete list instead of the more "Prologish", difference-list based which appends lists with complexity $O(1)$. See [18] or any other good textbook on Prolog for more information on the matter.

to a non-failing predicate, then $k = m$. A lower bound on the cost $Cost_P(\overline{x})$ of a **predicate** on an input of size n is then given by:

$$Cost_P(\overline{x}) \geq \min\{Cost_C^{inf}(\overline{x}) | C \text{ is a clause defining } P\}$$

Upper Bounds The upper bound of a clause is defined as

$$Cost_C^{sup}(\overline{x}) = \gamma(\overline{x}) + \sum_{i=1}^{m} Cost_{B_i}^{sup}(\phi_i(\overline{x}))$$

and an upper bound of the cost of a deterministic predicate can be taken as that of the more expensive clause:[3]

$$Cost_P(\overline{x}) \leq max\{Cost_C^{sup}(\overline{x}) | C \text{ is a clause defining } P\}$$

The worst-case running time of an algorithm is an upper bound on the running time for any input. Even if some analysis gives an upper bound which is clearly too high, the existence of such an upper bound makes it possible to deduce that the execution terminates, which is a quite relevant property.

2.2 Size and Cost Analysis

The definition of cost for clauses and predicates shown in the previous section assumed a cost function for every literal in the body. When these literals call (directly or indirectly) recursively the predicate being defined, this cost has to be worked out by solving a series of recurrence equations based on the sizes of the input data.

In order to do that, it is necessary to determine which arguments have to be selected to pose those equations (which needs mode analysis), which term measure is to be taken as data size (which additionally needs type analysis), and actually solving the equations. These are issues which fall outside the scope of this paper. Here we will focus on how these size relationships are set up; we will notwithstanding devote some efforts to solve the equations which come up from our running example.

Let us consider a clause defined as in Equation (1) and let us select one output argument x_k whose size we want to determine based on the sizes of the input arguments $\overline{x}$ which we will call $s(\overline{x})$. The size of this argument, as generated by the clause C, is given by an expression $Sz_C^{(k)}(s(\overline{x}))$ which is obtained by setting a series of equations of the form

$$Sz_C^{(k)}(s(\overline{x})) = sz^{(k)}(\overline{x}) + Sz_C^{(k)}(s(\overline{x'}))$$

where $\overline{x'}$ is the form of the tuple in the recursive call and $sz^{(k)}(\overline{x})$ is the difference in size between the output argument in the initial and the recursive call — i.e., how much the recursive call has contributed to the size of the output argument in position k, at the expense of reducing the size of the input argument tuple from $s(\overline{x})$ to $s(\overline{x'})$.

For recursive clauses, the formulas above (simplified for this paper; the reader is kindly referred to, e.g., [4] for a more complete description) generate recurrence equations which must be solved in order to obtain the size of each output position for each clause in terms of the size of the input arguments in the head of the clause.

Example: Consider the clauses of `append/3` (Figure 1). Let x and y be the length of the input lists (i.e., the first and second arguments). The size relations for this predicate are:

$$\begin{aligned} Sz_{append}(0, y) &= y \\ Sz_{append}(x, y) &= 1 + Sz_{append}(x - 1, y) \end{aligned}$$

The first equation is the boundary condition and represents the size of the output list when recursion finishes. The second equation represents the contribution to the size of the output list for every step taken in the input list. The solution to this equation is $Sz_{append}(x, y) = x + y$.

Example: Consider the third clause of `partition/4` (Figure 1). Let x and y be the first and second arguments for that predicate, which can be automatically determined to be the input ones. Let $Sz_{partition}^{(i)}(x, y)$ denote the size of the i-th output argument ($i \in \{3, 4\}$) as a function of the size of the two input arguments x and y. Using list length as

[3] For a non-deterministic predicate a trivial upper bound could be computed by adding the cost of every clause instead of taking the maximum.

a measure to determine the size relations for this clause we can write the following difference equations for the output arguments (third and fourth):

$$\begin{aligned} Sz^{(3)}_{partition}(x,y) &= Sz^{(3)}_{partition}(x-1,y) \\ Sz^{(4)}_{partition}(x,y) &= Sz^{(4)}_{partition}(x-1,y)+1 \end{aligned}$$

By doing the same for the rest of the clauses we obtain a set of recurrence equations (one for each output argument) plus a non-recurrent equation for the base clause(s). These equations can be solved to approximate (either as an upper or as a lower bound) the size of the output arguments. We will see a worked example in the next section.

2.3 Lack of Tightness in Some Upper Bounds

The above formulation can throw very inexact results in seemingly simple cases. We will see that QuickSort is such a case; in this example the size measure used for all predicates is list length [4]. Consider the `qsort/2` program in Figure 1. We will develop and solve the size equations for this program, and then we will use the solved forms to derive a cost estimation for its upper bound.

The difference equations for the size relations of the clauses of `partition/4` are as follows:

$$\begin{aligned} Sz^{(3)}_{partition}(0,y) &= 0 \\ Sz^{(3)}_{partition}(x,y) &= Sz^{(3)}_{partition}(x-1,y)+1 \\ Sz^{(3)}_{partition}(x,y) &= Sz^{(3)}_{partition}(x-1,y) \\ Sz^{(4)}_{partition}(0,y) &= 0 \\ Sz^{(4)}_{partition}(x,y) &= Sz^{(4)}_{partition}(x-1,y) \\ Sz^{(4)}_{partition}(x,y) &= Sz^{(4)}_{partition}(x-1,y)+1 \end{aligned}$$

Since there are several recursive clauses which contribute differently to the size of the output arguments and we are trying to deduce an upper bound on the cost, we take into consideration the equations which produces the bigger cost. In this case we will select the second clause for the third argument and the third clause for the fourth argument. The first clause, associated with the empty list in the input, produces the boundary condition of these difference equations. Note that the second argument does not contribute (at least directly) to the size of the output argument. In what follows we will omit it.

Although this particular set of equations is easy to solve, we will use a lemma[4] for a wider class of equations which will be useful later.

Lemma 1 *Let us suppose the next recurrence equation:*

$$\begin{aligned} y(0) &= C \\ y(x) &= p\ y(x-k)+f(x) \end{aligned}$$

where k is an arbitrary integer value such that $0<k<x$, C is a constant, p is an integer such that $p>0$, and $f(x)$ is any function. The solution of this equation is:

$$y(x) = p^{\frac{x}{k}}\ C + \sum_{i=0}^{\frac{x}{k}-1} p^i\ f(x-ik)$$

Proof: *By induction on x. For $x=0$ the result holds trivially because the sum is null for terms of $i<0$. Let us assume that the Lemma holds for all $y(x-k)$ such that $0<k<x$. Then we have*

$$y(x-k) = p^{\frac{x-k}{k}}\ C + \sum_{i=0}^{\frac{x-k}{k}-1} p^i\ f(x-(i+1)k)$$

[4]Note that we do not claim authorship of this lemma.

and we expand the corresponding equations for $y(x)$ as follows:

$$\begin{aligned} y(x) &= p\,y(x-k) + f(x) = p\,(\,p^{\frac{x-k}{k}}\;C + \sum_{i=0}^{\frac{x-k}{k}-1} p^i\;f(x-(i+1)k)) + f(x) \\ &= p^{\frac{x}{k}}\,C + \sum_{i=1}^{\frac{x-k}{k}-1} p^i\,f(x-ik) + f(x) = p^{\frac{x}{k}}\,C + \sum_{i=0}^{\frac{x}{k}-1} p^i\,f(x-ik) \end{aligned}$$

□

Then, the solutions of the equations which express the relation size for `partition/4`, according to Lemma 1, are $Sz^{(3)}_{partition}(x,y) = x$ and $Sz^{(4)}_{partition}(x,y) = x$. We already saw in the previous section the solution of the size equations for the predicate `append/3`. For `qsort/2` we have the following equations:

$$\begin{aligned} Sz^{(2)}_{qsort}(0) &= 0 \\ Sz^{(2)}_{qsort}(x) &= Sz^{(3)}_{append}(Sz^{(2)}_{qsort}(Sz^{(3)}_{partition}(x-1)), 1 + Sz^{(2)}_{qsort}(Sz^{(4)}_{partition}(x-1))) \\ &= Sz^{(3)}_{append}(Sz^{(2)}_{qsort}(x-1), 1 + Sz^{(2)}_{qsort}(x-1)) \\ &= Sz^{(2)}_{qsort}(x-1) + 1 + Sz^{(2)}_{qsort}(x-1) \\ &= 2\;Sz^{(2)}_{qsort}(x-1) + 1 \end{aligned}$$

The size relation for the clauses of `qsort/2`, obtained by using Lemma 1, is $Sz^{(2)}_{qsort}(x,y) = 2^x - 1$. This is clearly an overestimation which comes from a gross upper bound approximation of the output arguments of `partition/4`.

We will now derive the cost equations. As we will use resolution steps as the basis of our measure, we will need the functions which relate input and output sizes. The cost equations for `partition/4` are therefore:

$$\begin{aligned} C_{partition}(0) &= 1 \\ C_{partition}(x) &= 1 + C_{partition}(x-1) \end{aligned}$$

whose solution is $C_{partition}(x) = x+1$. Analogously, the difference equations for the cost of `append/3`, considering two input lists of large x and y respectively, are:

$$\begin{aligned} C_{append}(0,y) &= 1 \\ C_{append}(x,y) &= 1 + C_{append}(x-1,y) \end{aligned}$$

and its solution is $C_{append}(x,y) = x+1$. For `qsort/2` we write (note that we are using a line per term corresponding to body goals):

$$\begin{aligned} C_{qsort}(0) &= 1 \\ C_{qsort}(x) &= 1 + \\ &\quad C_{partition}(x-1) + \\ &\quad C_{qsort}(Sz^{(3)}_{partition}(x-1)) + \\ &\quad C_{qsort}(Sz^{(4)}_{partition}(x-1)) + \\ &\quad C_{append}(Sz_{qsort}(Sz^{(3)}_{partition}(x-1)), 1 + Sz_{qsort}(Sz^{(4)}_{partition}(x-1))) \\ &= 1 + x + 2\,C_{qsort}(x-1) + 2^x - 1 \\ &= x + 2^x + 2\,C_{qsort}(x-1) \end{aligned}$$

We can solve this recurrence to obtain $C_{qsort}(x) = (x+3)2^x - x - 2$, which is, again, exponential. We have lost accuracy and, while this is a correct upper bound (and makes it possible to deduce that `qsort/2` always terminates), is of little practical use for a number of other purposes, namely to obtain a meaningful estimation of the runtime needed to perform, e.g., granularity control of parallel execution.

3 Probabilistic Cost

In order to work around the overestimation which we have just seen, we will, instead of relying on an upper bound, assign sizes to arguments in a way that depends on some probability assignment which states a possibility of being selected assigned to every clause.

3.1 Probabilistic Size Equations

We will set up the size equations for every clause as we had done before. However, assuming determinism and non-failure, we will work out a size expression per predicate which tries to take into account how every clause contributes to the output arguments. Let us suppose a predicate Q which is defined by n clauses $C_1, ..., C_n$, and let $Sz_{C_i}^{(j)}(\overline{x})$ be a function on the size of the clause input arguments which returns the size of the output argument j.

Let P_i be a distribution of probability which associates to every clause C_i (except for the base ones) the likelihood of being selected in a particular call. As usual we require that

$$\sum_{j=1}^{n} P_j = 1 \tag{2}$$

A predicate-level equation for $Sz_Q^{(k)}(x)$ can then be stated as follows:

$$Sz_Q^{(k)}(\overline{x}) = \sum_{j=1}^{n} P_j \, Sz_{C_j}^{(k)}(\overline{x}) \tag{3}$$

I.e., we assume that every clause contributes differently to the size of the output arguments. Similarly to the upper and lower bounds, we define the probabilistic cost $\overline{Cost}_C(x)$ of a clause C, as the sum of the costs associated to the head of the clause plus the probabilistic cost associated with each of the body literals:

$$\overline{Cost}_C(\overline{x}) = \gamma(x) + \sum_{i=1}^{m} \overline{Cost}_{B_i}(\overline{\phi}_i(\overline{x}))$$

where $\overline{\phi}_i$ is the size of the i-th argument given by the probabilistic method.

As we did before, we first determine the relative sizes of the output arguments by setting up and solving a set of difference equations. The main difference is that in this case instead of selecting a single equation (c.f., clause) to return an upper or lower bound, we will combine the effect of the different clauses in each output argument. We will now apply this method to the QuickSort example in order to evaluate its application in a concrete case.

3.2 Probabilistic Analysis of QuickSort

Let us focus again on the Quicksort program (Figure 1). For the predicate `partition/4`, let us assume that the second clause can be selected with probability p ($0 \leq p \leq 1$) and that the third clause is to be selected with probability $1 - p$. The difference equations which express the sizes of the third and fourth (output) arguments in terms of the first and second arguments take into account now the probability assigned to each clause. Therefore, the third argument is contributed to with weight p by the second clause and with weight $1 - p$ by the third clause. The equations which relate input and output sizes are:

$$\begin{aligned}
S^{(3)}_{partition}(0, y) &= 1 \\
S^{(3)}_{partition}(x, y) &= p\,[S^{(3)}_{partition}(x-1, y) + 1] + (1-p)\, S^{(3)}_{partition}(x-1, y) \\
&= p + S^{(3)}_{partition}(x-1, y)
\end{aligned}$$

$$\begin{aligned}
S^{(4)}_{partition}(0, y) &= 1 \\
S^{(4)}_{partition}(x, y) &= (1-p)\,[S^{(4)}_{partition}(x-1, y) + 1] + p\, S^{(4)}_{partition}(x-1, y) \\
&= 1 - p + S^{(4)}_{partition}(x-1, y)
\end{aligned}$$

Note that the effect of the second argument to `partition/4` is now somehow captured by the probability of clause selection and it is not completely ignored as it was in the previous scheme. The solutions to these equations, using Lemma 1, are $S^{(3)}_{partition}(x,y) = p\,x$ and $S^{(4)}_{partition}(x,y) = (1-p)\,x$, for all $x \geq 0$. This is intuitively clear, as the second clause (with probability p) is the only one which contributes to the third argument, and similarly with the third clause and the fourth argument. It is important to note that the sizes of the output arguments (two lists) add up to the size of the input list now — an invariant which is clearly true and which was not met when we used an upper bound of the size. We had already determined in section 2.3 an exact size expression for the output argument of `append/3`: $S^{(3)}_{append}(x,y) = x + y$. With that expression we can write the difference equations for the size relation of the arguments of `qsort/2` as follows:

$$\begin{aligned} S^{(2)}_{qsort}(0) &= 0 \\ S^{(2)}_{qsort}(x) &= S^{(2)}_{qsort}(p(x-1)) + S^{(2)}_{qsort}((1-p)(x-1)) + 1 \end{aligned}$$

To help us in solving this recurrence equation we will prove another lemma.

Lemma 2 *Let us suppose the following recurrence equation:*

$$\begin{aligned} y(0) &= C \\ y(x) &= hx + d + y(\alpha(x-1)) + y((1-\alpha)(x-1)) \end{aligned}$$

for $x \geq 0$. C, h, and d are constants, $h, d \geq 0$, and α is a value such that $\frac{1}{2} \leq \alpha < 1$. Then $y(x)$ is bounded by:

$$y(x) \leq (2^k - 1)d + h(kx - 2^k + k + 1) + 2^k C$$

where $k = 1 + \log_{\frac{1}{\alpha}}((1-\alpha)(x-1)+1)$ is the depth of recursion. Furthermore, if $h > 0$ then $y(x)$ is $O(x \log x)$ for all $x > 0$. If $h = 0$, $d > 0$ and $C = 0$ then $y(x) = dx$, for all $x > 0$.

Proof: *By expanding the recurrent definition of $y(x)$ we obtain:*

$$\begin{aligned} y(x) &= hx + d + y(\alpha(x-1)) + y((1-\alpha)(x-1)) \\ &= h(2x-1) + 3d + y(\alpha^2(x-1) - \alpha) \\ &+ y(\alpha(1-\alpha)(x-1) - (1-\alpha)) + y(\alpha(1-\alpha)(x-1) - \alpha) \\ &+ y((1-\alpha)^2(x-1) - (1-\alpha)) \end{aligned}$$

When we reach the k-th level of its expansion as a binary tree we arrive at an expression as:

$$\begin{aligned} y(x) &= h(kx - 2^k + k + 1) + (2^k - 1)d \\ &+ y(\alpha^k(x-1) - \frac{1-\alpha^k}{1-\alpha} + 1) + \ldots + y((1-\alpha)^k(x-1) - \frac{1-(1-\alpha)^k}{\alpha} + 1) \end{aligned}$$

There are, naturally, 2^k terms in the leaves of the expansion. We will suppose that the recursion stops when the evaluation of its first term is zero. Therefore we establish:

$$\alpha^k(x-1) - \frac{1-\alpha^k}{1-\alpha} + 1 = 0$$

Note that assuming that any other term in the leaves is zero will also be valid, since it is used only to determine the depth of the recursion, k, based on the input argument x:

$$k = 1 + \log_{\frac{1}{\alpha}}((1-\alpha)(x-1)+1)$$

and, from that, we derive an upper bound of $y(x)$ by substituting the value of k in the expansion of $y(x)$:

$$y(x) \leq h(kx - 2^k + k + 1) + (2^k - 1)d + 2^k C$$

This result indicates, if $h > 0$, that:

$$\begin{aligned} y(x) &\leq hkx + (d + C - h)2^k + h(k+1) - d \\ &\leq hx(1 + \log_{\frac{1}{\alpha}}((1-\alpha)(x-1)+1)) \\ &+ 2(d + C - h)((1-\alpha)(x-1)+1)^{\log_{\frac{1}{\alpha}} 2} + h(k+1) - d \end{aligned}$$

Since $\frac{1}{2} \leq \alpha < 1$*, then it follows that* $\log_{\frac{1}{\alpha}} 2 \leq 1$*, and then* $y(x) \leq Kx \log x$*, with* K *a constant. Then* $y(x)$ *is* $O(x \log x)$*.*

If $h = 0$*,* $d > 0$*, and* $C = 0$ *then the recurrence equation becomes:*

$$\begin{aligned} y(0) &= 0 \\ y(x) &= d + y(\alpha(x-1)) + y((1-\alpha)(x-1)) \end{aligned}$$

whose solution is $y(x) = dx$ *for all* $x > 0$*, which can be proved easily by induction on* x*: for* $x = 0$ *the result* $y(0) = 0$ *holds trivially. Let us assume than the result holds for some* $x - 1$*. Then:*

$$\begin{aligned} y(x) &= d + y(\alpha(x-1)) + y((1-\alpha)(x-1)) \\ &= d + d\alpha(x-1) + d(1-\alpha)(x-1) \\ &= d + d(x-1) \\ &= dx \end{aligned}$$

□

The solution to the size relation equations for `qsort/2`, is, using the above lemma, $S^{(2)}_{qsort}(x) = x$, which is a much better approximation (in our case, exact) than the one obtained in Section 2.3. The equation for the probabilistic cost of `qsort/2` becomes now

$$\begin{aligned} \overline{Cost}_{qsort}(0) &= 1 \\ \overline{Cost}_{qsort}(x) &= 1 + \overline{Cost}_{partition}(x-1) + \overline{Cost}_{qsort}(p(x-1)) \\ &\quad + \overline{Cost}_{qsort}((1-p)(x-1)) + \overline{Cost}_{append}(p(x-1)) \end{aligned}$$

We had already found in Section 2.3 an exact solution in solved form (and which are, therefore, correct also as probabilistic cost) for the costs of `partition/4` and `append/3`:

$$\begin{aligned} \overline{Cost}_{partition}(x) &= x + 1 \\ \overline{Cost}_{append}(x) &= x + 1 \end{aligned}$$

After substituting them in the equation for $\overline{Cost}_{qsort}(x)$ we obtain:

$$\begin{aligned} \overline{Cost}_{qsort}(0) &= 1 \\ \overline{Cost}_{qsort}(x) &= (p+1)x + 2 - p + \overline{Cost}_{qsort}(p(x-1)) + \overline{Cost}_{qsort}((1-p)(x-1)) \end{aligned}$$

As per Lemma 2, for $p = 0$ and $p = 1$ we were able to obtain an exact solution. For other values of p we obtained an approximation. We obtain, for some values of p (including $p = 1$ and $p = 0$):

$$\begin{array}{ll} \overline{Cost}_{qsort}(x) = \frac{x^2+7x+2}{2} & \text{for } p = 0 \\ \overline{Cost}_{qsort}(x) = x^2 + 3x + 1 & \text{for } p = 1 \\ \overline{Cost}_{qsort}(x) \leq (x+1)(1 + \frac{3}{2}\log_2(x+1)) & \text{for } p = \frac{1}{2} \\ \overline{Cost}_{qsort}(x) \leq \frac{4}{3}(2x+1)^{\log_3 2} - \frac{1}{3} + \frac{4}{3}(x+1)\log_3(2x+1) & \text{for } p = \frac{1}{3} \\ \overline{Cost}_{qsort}(x) \leq \frac{3}{2}(3x+1)^{\frac{1}{2}} - \frac{1}{2} + \frac{5}{4}(x+1)\log_4(3x+1) & \text{for } p = \frac{1}{4} \\ \overline{Cost}_{qsort}(x) \leq \frac{7}{4}(7x+1)^{\frac{1}{3}} - \frac{3}{4} + \frac{9}{8}(x+1)\log_8(7x+1) & \text{for } p = \frac{1}{8} \end{array}$$

In all cases, but in the extreme ones, we obtain complexity orders of $O(x \log x)$. However the associated cost (i.e., the numerical value of the cost for a given x) grows larger as the input list is "more sorted", approaching a cost which is $O(x^2)$. The "average" case (for $p = \frac{1}{2}$) gives an expected average cost.

Probability of selection	Execution time (ms)	Predicted steps	steps/ms.
0	12506	50035001.000	4000.880
1/8	64	422573.466	6602.703
1/4	50	279132.163	5582.640
1/3	45	241090.314	5357.556
1/2	48	209338.781	4361.208
1	24879	100030001.000	4020.660

Table 1: Comparison between predicted and experimental performance

3.3 An Experimental Assessment

Table 1 shows some results of several experiments with `qsort/2`. They were performed on a Pentium Mobile 1.7 GHz, with the clock speed set to its maximum (to avoid skews due to automatic CPU frequency adjustments), and running Linux with kernel 2.6.15. The measurements were done by sorting a 10000-element list. Standard protections against spurious interference were taken (i.e., cache and memory warming-up, computer isolation and low load, averaging benchmark results, etc.).

The first column shows the probability p of selecting the clauses of `partition/4`. We made sure that this is actually the probability that each clause in `partition/4` is chosen in each test by carefully working out a set of inputs which precisely select each of the clauses an appropriate number of times. $p = 1$ and $p = 0$ correspond, obviously, to an input list sorted in either increasing or decreasing order. The second column in the Table shows the running time for each case of input data. The third column shows the predicted steps using cost equations obtained using a probabilistic approach, and the fourth column shows the ratio between predicted steps and actual time. This last column makes it possible to determine quickly whether there is a (strong) correlation between the predicted time and the time obtained experimentally.

From that table we can see that the prediction makes a reasonable guess. The runs where the input list is not sorted are notably faster than the other executions, as expected; despite this, the prediction stays within a reasonable error for those cases. It has to be taken into account that, for the sake of simplicity, we have left out several execution details (such as cost of builtins, cost of shallow backtracking, etc.), which makes the model less accurate than possible. Additionally, the solution of lemma 2 is, at this moment, only approximate. This adds up to the lack of a more refined model and can account for the discrepancy between the predicted and the expected execution time.

We expect that more precise predictions can be made with a more refined cost model (taking into account specific costs for builtins, head unifications which are made only partially and do not succeed finally because they fail at some point in the middle of unifying the arguments, etc.).

4 Relation with Upper and Lower Bounds

It is trivial to determine that the probabilistic cost lies between the lower and upper bounds, as it should be indeed. The key observation is that the probabilistic cost is in our model a linear combination (Equation (3)) of the contributions of individual clauses with a restriction (Equation (2)) on the values of the coefficients. When lower and upper bounds are used, they boil down to giving one clause all the available weight. If we try to select a set of weights (abiding by the restrictions on their values) which give the greatest (resp., lowest) possible value to the cost expressions, we will find that we will precisely end up with the same selection of rules as the upper (resp., lower) bounds.

5 Conclusions and Further Work

This paper presents an idea and sketches a technique which applies probabilities to the estimation of the computational cost of logic programs. It is based on a modification of previously existing techniques to deduce lower and upper bounds for their cost. Probabilities express the likelihood that a given clause is selected over the others, and should ultimately represent the conditions in some running environment.

By applying this technique we obtain recurrence relations which contain the expected distribution of the recursive clauses as parameters. The results of this approach allow us to approximate the cost (in terms of resolution steps) of deterministic logic programs. When a general solution which keeps the probability assignment as parameter can be found, we obtain an expression for the computational cost of that predicate or program. For the cases in which this has not (yet) been possible, the recurrence relation can be specialized for some selected values of the probabilities,

which simplifies the equations and makes solving them easier. In this case special attention has to be paid to values of probabilities which would correspond to singular points in the solved form.

In the case of our running example, QuickSort, we obtain an average (i.e., assuming that the input list is not sorted) case complexity of $O(n \log n)$, being n the length of the input list. When we assume that the input list is sorted we obtain a complexity of $O(n^2)$. Both these results improve upon the upper bound previously obtained for this predicate, which was exponential.

As for future work, we intend to refine the model to take into account more execution characteristics, find a tighter relationship between the solutions given by the probabilistic model with the classical $O(\cdot)$ approximation and with other measures of complexity, investigate further in the analytical solution of the kind of recurrence equations that the probabilistic method generates, and progress towards the automatic implementation of this analysis and the solution of the difference equations generated.

References

[1] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. Optimizing Prolog for Small Devices: A Case Study. Technical Report CLIP4/2006.0, Technical University of Madrid, School of Computer Science, UPM, April 2006. Under consideration for publication.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992.

[3] S. K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.

[4] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[5] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 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. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[7] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[8] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-Case Analysis of Algorithms. *Theor. Comp. Sci.*, 79(1):37–109, 1991.

[9] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System. In *Proc. of EURO–PAR 2004*, number 3149 in LNCS, pages 21–37. Springer-Verlag, August 2004.

[10] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

[11] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[12] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[13] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[14] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[15] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[16] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation for Logic Programs via Static Analysis and Profiling. In S. Mu noz and W. Vanhoof, editors, *16th Workshop on Logic Programming Environments*, pages 45–60. University of Namur, Institut d'Informatique, August 2006.

[17] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Res. Rep. CS-90-1, Dept. of Computer Science, Univ. of Sheffield, England, January 1990.

[18] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[19] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.