Cost Estimation of Logic Programs

Saumya K. Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Cost Analysis : Motivation

1. **Manual Analyses** : Influences choices of data structures and algorithms.

2. **(Semi-)Automatic Analyses** :
   - "Granularity control" in parallelizing compilers.
   - Task and data mapping in distributed systems.
   - Guide program transformation systems.
   - Query optimization in deductive databases.

⋆ **Cost Analysis of Logic Programs** :
   - uses a variety of other program analyses; and
   - offers many applications for combinatorial approximation algorithms.

Related Work

"Classical" : Cohen & Zuckerman 74; Wegbreit 75, 76; Ramshaw 79; Flajolet & Steyaert 81, 87; Hickey & Cohen 88.

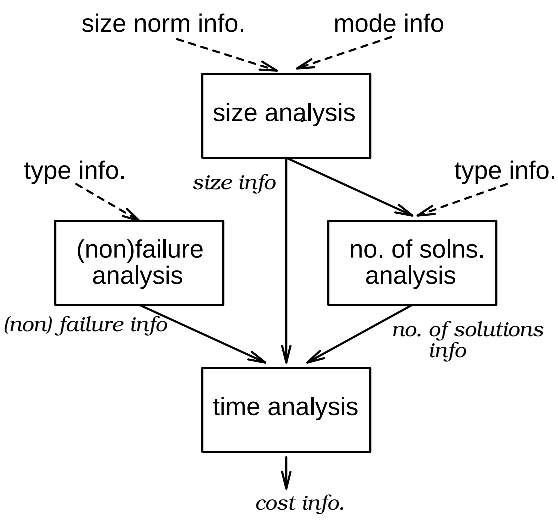
Functional Languages : Le Métayer 88, Rosendahl 89.

Higher order functions : Shultis 85; Le Métayer 87; Sands 90.

Lazy evaluation : Wadler 88, Bjerner & Holmström 89, Sands 90.

Logic Programming : Kaplan 88.

Cost Analysis of Logic Programs: Organization



Upper Bound Analysis of Logic Programs

Fundamental Issue : dealing with nondeterminism.

- Procedures can generate multiple solutions.
- No. of times a program fragment is executed depends on the no. of solutions for earlier goals.

  $\Rightarrow$ in general, assume that all solutions computed for every procedure.
- For nontrivial upper bound analysis, we need good upper bounds on no. of solutions.

  This is complicated due to recursion: no. of solutions becomes dependent on input size.

Upper Bound Analysis: Argument Sizes

Process SCCs in the call graph in topological order:

- For each procedure, express the size of each output argument as a function of its input sizes.
- Set up equations describing size relationships between variables in each clause (assuming size norm info. given).
- Simplify the equations. This yields:
  - For non-recursive procedures: output size as a function of the input size.
  - For recursive procedures: output size obtained as a recurrence in terms of the input sizes.
- Solve difference equations (or use upper bound approximations) to obtain size relationships for a procedure.

Argument Size Analysis: Example

Program :

```
:- mode app(in, in, out).
app([], L, L).
app([H|L1], L2, [H|L3]) :- app(L1, L2, L3).
```

Size Equations :

app(x_1, x_2, x_3) :−	%%	$x_3 = sz\_app(x_1, x_2)$
x_1 = [y_1|y_2],	%%	$x_1 = y_2 + 1$
app(y_2, x_2, z_1),	%%	$z_1 = sz\_app(y_2, x_2)$
x_3 = [y_1|z_1],	%%	$x_3 = z_1 + 1$
app(x_1, x_2, x_3) :−	%%	$x_3 = sz\_app(x_1, x_2)$
x_1 = [],	%%	$x_1 = 0$
x_3 = x_2.	%%	$x_3 = x_2$

Simplification :

$$sz\_app(x, y) = sz\_app(x-1, y) + 1 \qquad [x > 0]$$
$$sz\_app(0, y) = y$$

Solution : $sz\_app(x, y) = x + y$.

Argument Size Analysis: Example

Program :

```
:- mode nrev(in, out), app(in, in, out).
nrev([H|L], R) :- nrev(L, R1), app(R1, [H], R).
nrev([], []).
```

Size Equations :

nrev(x_1, x_2) :−	%%	$x_2 = sz\_nrev(x_1)$
x_1 = [y_1|y_2],	%%	$x_1 = y_2 + 1$
nrev(y_2, z_1),	%%	$z_1 = sz\_nrev(y_2)$
z_2 = [y_1],	%%	$z_2 = 1$
app(z_1, z_2, x_2).	%%	$x_2 = sz\_app(z_1, z_2)$
nrev(x_1, x_2) :−	%%	$x_2 = sz\_nrev(x_1)$
x_1 = [],	%%	$x_1 = 0$
x_2 = [].	%%	$x_2 = 0$

Simplification : assume $sz\_app(x, y) = x + y$, then:

$$sz\_nrev(x) = sz\_nrev(x-1) + 1 \qquad [x > 0]$$
$$sz\_nrev(0) = 0$$

Solution : $sz\_nrev(x) = x$.

Upper Bound Analysis: No. of Solutions

- Basic approach as before:
  - for each procedure, no. of solutions expressed as a function of input size;
  - set up equations describing each variable, use symbolic expressions for recursion;
  - simplify, solve resulting difference equations.
- Rules for handling unification:

  No. of bindings for a variable : *min* of the no. of bindings possible for its different occurrences.

  No. of instances of a (compound) term : *product* of the no. of bindings possible for its variables.

  (The second rule can be refined to account for sets of variables whose bindings are not independent of each other.)

No. of Solutions Analysis: Example

Program :

```
:- mode bitlist(in,out), bit(out).
bitlist(0, []).
bitlist(N, [B|L]) :- bit(B), bitlist(N-1, L).
```

Size Equations :

bitlist(x, y) :–	%%	$y = nsols\_bitlist(x)$
x = 0,	%%	$x = 0$
y = [].	%%	$y = 0$
bitlist(x, y) :–	%%	$y = nsols\_bitlist(x)$
bit(z),	%%	$z = nsols\_bit()$
x_1 = $x - 1$,	%%	$x_1 = x - 1$,
bitlist(x_1, y_1),	%%	$y_1 = nsols\_bitlist(x_1)$
y = [z | y_1].	%%	$y = z \times y_1$

Simplification : assume $nsols\_bit() = 2$, then:

$$nsols\_bitlist(x) = 2 \cdot nsols\_bitlist(x-1) \qquad [x > 0]$$
$$nsols\_bitlist(0) = 0$$

No. of Solutions: Improvements

Linear (binary) constraints:

- Suppose we have n variables and m values in a constraint satisfaction problem.

  Construct a graph where a node $v_{p,i}$ corresponds to an assignment of value c_i to variable x_p.
- If the assignments $\{x_p \mapsto c_i, x_q \mapsto c_j\}$ are mutually consistent in the constraint set given, draw an edge between $v_{p,i}$ and $v_{q,j}$.
- A solution to the constraints satisfies all the constraints.

  This corresponds to a clique of size n in the graph.
- Thus:
  - Existence of a solution $\sim$ the existence of an n-clique in the constraint graph.
  - No. of solutions to a n-variable constraint set $\sim$ no. of n-cliques in a graph.
- Deciding the n-cliques problem is NP-hard.

  We give a poly-time algorithm for an upper bound on the no. of n-cliques.

No. of Solutions: Improvements : cont'd

$\{=, \neq\}$-Constraints :

- Construct a graph where the nodes correspond to variables.:
  - If there is a constraint $X = Y$ then variables X and Y correspond to the same node.
  - If there is a constraint $X \neq Y$ then there is an edge between the nodes for X and Y.
  - Each value that the variables can take on corresponds to a different color.
- A solution $\sim$ a coloring of a graph where no two adjacent nodes get the same color.

  $\Rightarrow$ no. of solutions $\sim$ chromatic polynomial of the graph.
- Determining chromatic polynomials is NP-hard.

  We give a poly-time upper bound approximation algorithm for estimating the chromatic polynomial. (No. of solutions for the constraint set easily recovered from this.)

Handling Constraints: Example

Program : 8-queens:

```
queen(X1,X2,X3,X4,X5,X6,X7,X8) :-
   place(X1), place(X2), place(X3), place(X4),
   place(X5), place(X6), place(X7), place(X8),
   safe([X1,X2,X3,X4,X5,X6,X7,X8]).

safe([]).
safe([X|L]) :- noattacks(L,X,1), safe(L).

noattacks([],_,_).
noattacks([Y|L],X,D) :-
   ok(X,Y,D),
   D1 is D+1,
   noattacks(L,X,D1).

ok(X,Y,D) :- X =\= Y, Y-X =\= D, Y-X =\= -D.

place(1).   place(2).   ...   place(8).
```

Given the (type) information that each argument of `queen/8` ranges over $\{1,\ldots,8\}$, we infer that the no. of solutions for `queens/8` is at most 46312.

No. of Solutions Analysis: Summary

1. Do argument size analysis.

   [Use mode and data dependency information.]

2. For each clause, determine the no. of bindings possible for each variable.

   [Use type and argument size information.]

3. The no. of solutions for a clause is given by the total no. bindings possible for the variables appearing in its head.

4. The no. of solutions for a predicate is obtained from the no. of solutions for its clauses:

   (a) Partition the clauses for the predicate into mutually exclusive clusters.

   (b) For each cluster, sum the no. of solutions for the clauses.

   (c) For the predicate, take the *max* of the no. of solutions for its clusters.

Upper Bound Time Analysis

- Time to execute a call to a procedure p =

  $\sum$ time to execute a clause for p

- Time to execute a clause =

  time for head unification +
  time to execute the body

- Time to execute a clause body $B_1, \ldots, B_n$ =

  $\sum_i$ (time to execute B_i $\times$
  no. of times B_i is executed)

- No. of times B_i is executed =

  $\prod_{j=1}^{i-1}$ No. of solutions of B_j

Execution Time Analysis: Example

Program :

```
:- mode nrev(in, out), app(in, in, out).
nrev([H|L], R) :- nrev(L, R1), app(R1, [H], R).
nrev([], []).
```

Cost Equations (in terms of no. of procedure calls) :

nrev(x_1, x_2) :−	%% $T\_nrev(x_1)$,
x_1 = [y_1|y_2],	
nrev(y_2, z_1),	%% $T\_nrev(y_2)$,
z_2 = [y_1],	
app(z_1, z_2, x_2).	%% $+\ T\_app(z_1, z_2)$
nrev(x_1, x_2) :−	%% $T\_nrev(x_1) = 1$
x_1 = [],	
x_2 = [].	

Simplification : assume $T\_app(x,y) = x$, then:

$$T\_nrev(x) = T\_nrev(x-1) + x \qquad [x > 0]$$
$$T\_nrev(0) = 1$$

Solution : $T\_nrev(x) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$.

Lower Bound Analysis

Fundamental Issue : failure

- Possible failure of head unification produces a trivial lower bound of zero.

  ⇒ For nontrivial lower bounds we need to guarantee non-failure.

- The *type* of a procedure is usually given as an approximation to its success set.

  But:

  – Most type analyses formulated as *upper approximations* to the success set.

  – Nontrivial lower approximations to the success set are hard to get.

  We use a notion of *covering* that allows us to use upper approximations in guaranteeing non-failure.

Lower Bound Analysis: Covering

Intuition : A test $S(x) = S_1(x) \vee \cdots \vee S_n(x)$ *covers* a type assignment T for x if, for any $t \in T$, $S_i(t)$ holds for some i, $1 \leq i \leq n$.

Example : Consider the test

$$(\texttt{X = []}) \vee$$
$$(\texttt{X = [H|L]} \wedge \texttt{H} \geq \texttt{Y}) \vee$$
$$(\texttt{X = [H|L]} \wedge \texttt{H} < \texttt{Y})$$

and the type assignment

$$\{\texttt{X : list(int), Y : int}\}$$

For any list of integers X and integer Y :

1. either X is empty or not;
2. if X not empty then $X = [H|L]$: either $H \geq Y$ or not.

⇒ This test covers this type assignment.

Covering Analysis: Complexity

- Undecidable for arbitrary tests and type assignments.

  Proof : Given an arbitrary polynomial $P(x_1, \ldots, x_n)$, consider the test

  $$\mathcal{T}(\bar{x}) \equiv P(x_1, \ldots, x_n) \neq 0.$$

  $\mathcal{T}(\bar{x})$ covers the type assignment

  $$\{x_1 \mapsto \texttt{integer}, \ldots, x_n \mapsto \texttt{integer}\}$$

  if and only if $P(x_1, \ldots, x_n)$ has no integer roots.

- Co-NP-hard for finite type assignments.

  Proof : Let $\mathcal{T}(\bar{x}) \equiv B(x_1, \ldots, x_n)$ be a propositional formula.

  $\mathcal{T}(\bar{x})$ covers the type assignment

  $$\{x_1 \mapsto \texttt{Bool}, \ldots, x_n \mapsto \texttt{Bool}\}$$

  if and only if $B(x_1, \ldots, x_n)$ is a tautology.

Covering Analysis: The Herbrand Case

- The undecidability of general covering is rooted in tests involving arbitrary integer arithmetic.

- Covering analysis for pure Horn clause programs (i.e., without arithmetic) is decidable.

  Intuition : To determine whether a set of unifications cover a type assignment, "trace" down the decision tree.

- The decision problem is co-NP-complete [Kunen 87].

- In practice, the number of tests involved is not large.

Covering Analysis: Arithmetic

- Covering analysis for linear arithmetic constraints is decidable.

  Intuition : transform to integer linear programming.

  Example : The covering problem for the tests

  $(\forall x)(\forall y)[x = y \lor x \geq y \lor x < y]$

  becomes

  $\neg(\exists x)(\exists y)[x \neq y \land x < y \land x \geq y]$

  from which we can derive a set of integer programming problems.

- The decision problem is co-NP-complete.

- In practice, the number of tests involved is not large.

- In practice, almost all arithmetic tests are linear.

- Give up on non-linear arithmetic.

Non-Failure Analysis

1. For each body goal:
   (a) determine (input) argument types ;
   (b) check whether [extended] head unification tests of the callee cover the input type assignment.
2. Determine which procedures and body goals may fail:
   - A body goal may fail if:
     - its input argument types are not covered by the [extended] head unification tests of the callee; or
     - if the called procedure may fail.
   - A procedure may fail if any of its body goals may fail.
3. Propagate *"may fail"* information along call graph via depth first traversal.

Lower Bound Analysis: No. of Solutions

- Non-failure analysis gives a lower bound of 1 solution.

- Improving this: due to "interference" between computations, analogies from upper bound analysis may not apply.

  A simple sufficient condition to rule out interference: *the output arguments of a clause are distinct and unaliased, and occur only once in the body.*

- If a set of clauses has the same (extended) head unification tests, we can sum their no. of solutions.

Lower Bound No. of Solutions: Example

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).
```

Equations :

$$nsols\_subset(0) = 0$$
$$nsols\_subset(n) = 2 \cdot nsols\_subset(n-1)$$

Hence we infer a lower bound of 2^n solutions for an input of length n.

Lower Bound Time Analysis

Analogous to upper bound time analysis, except:

- in each clause, consider only body goals until the leftmost goal that may fail;
- assume only one solution needed, except in contexts where all solutions will be computed.

This basic analysis can be improved, e.g., by using low-level information about indexing structures.

Lower Bound Time Analysis: Example

```
qsort([],[]).
qsort([H|L],R) :-
  part(H,L,Sm,Lg),
  qsort(Sm,SmS), qsort(Lg,LgS),
  append(SmS,[H|LgS],R).

part(_,[],[],[]).
part(X,[Y|L],[Y|Sm],Lg) :- X >= Y, part(X,L,Sm,Lg).
part(X,[Y|L],Sm,[Y|Lg]) :- X <  Y, part(X,L,Sm,Lg).

append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

- Covering analysis infers that all of `qsort`/2, `part`/4 and `append`/3 are non-failing.
- Given a list of length n, we infer that `part`/4 has cost n (procedure calls).
- For `qsort`/2, the existing algorithm infers a lower bound of $n + 3$ for an input of length n.

An Application: Dynamic Program Partitioning

Experimental results for dynamic task granularity control on &-Prolog (Sequent Symmetry, 4 processors) :

programs	*seq*	*par*	*par+GC*
qsort(1000)	3.670	1.399	1.409
(*O*=m)		1	-0.0%
qsort(1000)	3.670	1.819	1.429
(*O*=5)		1	+21%
fib(19)	1.839	0.729	0.549
(*O*=m)		1	+24%
fib(19)	1.839	0.970	0.639
(*O*=5)		1	+34%
hanoi(13)	6.309	2.509	2.399
(*O*=m)		1	+4.4%
hanoi(13)	6.309	2.690	2.419
(*O*=5)		1	+10.1%
unb_matrix	2.099	1.009	0.870
(*O*=m)		1	+13.8%
unb_matrix	2.099	1.039	0.870
(*O*=5)		1	+16.3%

Conclusions

Current Status of Implementation :

- CASLOG (upper bound complexity analysis system) implemented by Nai-Wei Lin, available by anonymous FTP from `ftp.cs.arizona.edu`.

  Integrated into &-Prolog, for dynamic granularity control, by Pedro Lopez Garcia.
- Lower bound analyses currently being implemented by Pedro Lopez Garcia.

Future Work :

- Better handling of multi-output procedures (divide-and-conquer programs).
- Average case analysis.

CASLOG : Example Analysis

Program :

```
:- mode perm(in, out), select(out, in, out).
perm([], []).
perm(X, [R|Rs]) :- sel(R, X, Y), perm(Y, Rs).

sel(X, [X|Xs], Xs).
sel(X, [Y|Ys], [Y|Zs]) :- sel(X, Ys, Zs).
```

Cost Expressions Inferred :

$$\texttt{perm/2} : \begin{array}{lcl} sz\_perm(n) & = & n \\ nsols\_perm(n) & = & n! \\ time\_perm(n) & = & \sum_{i=1}^{n}(3n!/i!) + 3n! - 2 \end{array}$$

$$\texttt{sel/3} : \begin{array}{lcl} sz\_sel^{(1)}(n) & = & \perp \\ sz\_sel^{(3)}(n) & = & n-1 \\ nsols\_sel(n) & = & n \\ time\_sel(n) & = & 2n \end{array}$$

Analysis time : 0.33 secs.