Task Granularity Analysis in Logic Programs

Saumya K. Debray Nai-Wei Lin Manuel Hermenegildo

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

MCC
3500 West Balcones Center Drive
Austin, TX 78759

Abstract

While logic programming languages offer a great deal of scope for parallelism, there is usually some overhead associated with the execution of goals in parallel because of the work involved in task creation and scheduling. In practice, therefore, the "granularity" of a goal, i.e. an estimate of the work available under it, should be taken into account when deciding whether or not to execute a goal concurrently as a separate task. This paper describes a method for estimating the granularity of a goal at compile time. The runtime overhead associated with our approach is usually quite small, and the performance improvements resulting from the incorporation of grainsize control can be quite good. This is shown by means of experimental results.

1 Introduction

Logic programming languages offer a great deal of scope for parallelism. There are two principal flavors of parallelism in logic programs: AND-parallelism, where subgoals necessary to solve a goal are executed in parallel; and OR-parallelism, where different alternative execution branches are explored concurrently. By employing both kinds of parallelism, it is possible to extract "maximal" parallelism for a program [12].

This is interesting in the abstract. However, just because something *can* be done in parallel does not necessarily mean, in practice, that it *should* be done in parallel. This is because the parallel execution of a task incurs various overheads, e.g. overheads associated with process creation and scheduling, the possible migration of tasks to remote processors and the associated communication overheads, etc. Thus, given the clause

```
part([E|L], M, U1, [E|U2]) :-
      E > M, part(L, M, U1, U2).
```

the test `E > M` can be executed in parallel with the recursive call. However, this test can typically be carried out in one or two machine instructions, and if the overhead associated with spawning the test as a separate task is more than a few instructions, the parallel execution of this goal may not be cost-effective. In general, a goal should not be a candidate for parallel execution if its granularity, i.e. the "work available" underneath it, is less than the work necessary to create a separate task for that goal. This makes it desirable to devise a method whereby the granularity of a goal may be estimated at runtime; in order to be useful, the runtime overhead involved in such a method should be small, i.e. as much work should be done at compile time as possible.

This paper describes a method for statically estimating the granularity of predicates in a logic program. Most of the work in our approach is done at compile time. However, the work done by a call to a recursive predicate typically depends on the size of its input, and hence cannot be estimated in any reasonable way at compile time—for such goals, some runtime work is necessary to determine the cost of any particular call to a recursive predicate. However, the cost incurred in

such runtime computations is generally quite small.

Since compilers are allowed only to perform optimizations that can be guaranteed to not affect a program's runtime behavior adversely, and because interesting program properties are generally undecidable, compile-time analyses are usually expected to satisfy correctness criteria that state that information inferred during the analysis of a program is a sound, possibly conservative, estimate of the program's runtime behavior. Curiously, correctness criteria are not immediately obvious in the context of granularity analysis, since a mistake in granularity analysis can result in loss of performance but appears unlikely to change the semantics of the program. Despite this, it is desirable to be able to state what kind of invariant (with respect to runtime behavior) is satisfied by a granularity analysis algorithm, in order to allow us to reason formally about the behavior of programs that utilize the information inferred by it. Not surprisingly, the problem of determining precisely how much work will be done by a call is statically undecidable. This means that compile-time granularity analysis will be a conservative estimate of the amount of work performed at runtime. As such, it can give either a lower bound or an upper bound on the amount of runtime computation.

The analysis considered in this paper gives granularity estimates that are an upper bound on the amount of work that may be done at runtime. There are a number of reasons for this. An important philosophical reason for this choice is the following: if a lower-bound analysis is conservative, it determines there is less work available than there is in practice, resulting in a loss of parallelism; this is conceptually akin to parallelizing sequential language programs, where actions are performed sequentially unless specified otherwise. If an upper-bound analysis is conservative, however, tasks are executed concurrently even though there may not be enough work available to justify this; this corresponds, conceptually, to "sequentializing" a parallel language, where actions are performed in parallel unless specified otherwise. Because the language models we have in mind resemble the latter rather than the former, upper-bound analyses appear to be more appropriate for our purposes. There are also important practical advantages to choosing upper-bound analyses, since it is difficult to give nontrivial lower bounds in most cases (very often, for example, the case where head unification fails leads to a lower bound estimate of 0, which is not very useful), and also because many important simplifications can be performed if we are required to guarantee only an upper bound. This results in significant simplifications to our algorithms, with concomitant improvements in both compile-time and runtime overhead; in particular, unlike a lower-bound analysis, termination issues do not have to be considered separately.

It is assumed that the reader is acquainted with the fundamentals of logic programming. The remainder of this paper is organized as follows: Section 2 gives an overview of the approach for granularity analysis. Section 3 illustrates the method for the inference of argument size relations. Section 4 presents the scheme for cost estimation of goals. Section 5 describes a mechanism for obtaining (approximate) solutions for difference equations. Section 6 argues the soundness of our granularity analysis algorithm. Section 7 shows some experiment results of using the information from granularity analysis in granularity control. Section 8 describes some related works, and Section 9 gives some conclusions.

2 An Overview of the Approach

Granularity analysis for a set of nonrecursive procedures is relatively straightforward. Recursion is somewhat more problematic: the amount of work done by a recursive call depends on the depth of recursion, which in turn depends on the input. Reasonable estimates for the granularity of recursive predicates can thus be made only with some knowledge of the input. Our technique for dealing with this problem is to do as much of the analysis at compile time as possible, but postpone the actual computation of granularity until runtime. A fundamental criterion in our approach is that the runtime overhead incurred in this computation should be small. Given a recursive predicate p, therefore, we compute an expression $\Phi_p(n)$ that satisfies the following criteria:

1. $\Phi_p(n)$ is relatively easy to evaluate; and

2. $\mathtt{Cost}_p(n) \leq \Phi_p(n)$ for all n, where $\mathtt{Cost}_p(n)$ denotes the cost of computing p for an input of size n.

The idea is that $\Phi_p(n)$ is determined at compile time; it is evaluated at runtime, when the size of the input is known, and yields an estimate of the granularity of the

predicate.[1] For example, given a predicate defined by

```
p([]).
p([H|L])  :-  q(H), p(L).
```

assume that the literals `q(H)` and `p(L)` in the body of the second clause can be shown to be independent, so that these literals are candidates for concurrent execution.[2] Suppose the expression $\Phi_q(n)$ giving the cost of q on an input of size n is $3n^2$, and suppose the cost of creating a concurrent task is 48 units of computation. Then, the code generated for the second clause might be of the form

> $n :=$ *size*(`H`);
> **if** $3n^2 < 48$ **then** *execute* `q` *and* `p` *sequentially*
> *as a single task*
> **else** *execute* `q` *and* `p` *concurrently as separate tasks*

Of course, this could be simplified further at compile time, so that the code actually executed at runtime might be of the form

> **if** *size*(`H`) < 4 **then** *execute* `q` *and* `p` *sequentially*
> *as a single task*
> **else** *execute* `q` *and* `p` *concurrently as separate tasks*

The expressions $\Phi_p(n)$ are obtained by setting up difference equations for predicates and obtaining (upper bound) solutions to them. In order to set up difference equations, however, it is necessary to track argument sizes. Consider, for example, the predicate `nrev/2`, defined as:

```
nrev([], []).
nrev([H|L], R)  :-
    nrev(L, R1),append(R1, [H], R).
```

In order to determine the work done in the second clause, it is necessary to estimate the work done by the call to `append`. To do this, it is necessary to be able to estimate the size of the binding of the variable `R1`, relative to that of the input list, at the return from the recursive call to `nrev`. For this, we use an abstraction of clauses called a data dependency graph. Our approach to granularity analysis thus consists of the following steps:

1. Use data dependency graphs to determine the relative sizes of variable bindings at different program points;
2. use the size information to set up difference equations representing the computational cost of predicates;
3. compute upper bounds to the solutions of these difference equations to obtain estimates of task granularities.

These steps are discussed in greater depth in the following sections.

3 Argument Size Relations of Predicates

The cost of a recursive goal depends on the size of input, which determines the depth of recursion. Therefore, to determine the cost of a recursive subgoal in the body of clause, it is necessary to infer the relative sizes between the arguments in the subgoal and those in the head. This section describes a data dependency-based method for statically estimating the argument size relations of predicates.

To facilitate the discussion, we introduce the following terminology: a body literal in a clause is called a *recursive* literal if it is part of a cycle in the call graph for the program that contains the head of that clause. A clause is called *nonrecursive* if no body literal is recursive, and is called *simple recursive* if it contains recursive literals and all the recursive literals have the same predicate symbol as the head; otherwise, it is called *mutually recursive.* A clause is a recursive clause if it is either simple recursive or mutually recursive.

Various measures can be used to determine the "size" of an input, e.g., term-size, term-depth, list-length, integer-value, etc. The measure(s) appropriate in a given situation can generally be determined by examining the operations used in the program. Let $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ be a function that maps ground terms to their

[1] In practice, we might prefer to not have to traverse the entire input at runtime to determine its size. This problem, which can be handled by maintaining some additional information, is somewhat orthogonal to the topic of this paper, and is not pursued further here.

[2] This can be done in most practical cases automatically at compile-time using global analysis techniques. See [5, 7] for details and other references.

sizes under a specific measure m, where $\mathcal{H}$ is the Herbrand universe, i.e. the set of ground terms of the language, and $\mathcal{N}_\perp$ the set of natural numbers augmented with a special symbol $\perp$, denoting "undefined". Examples of such functions are "list_length", which maps ground lists to their lengths and all other ground terms to $\perp$; "term_size", which maps every ground term to the number of constants and function symbols appearing in it; "term_depth", which maps every ground term to the depth of its tree representation; and so on. Thus, $|[\mathtt{a},\mathtt{b}]|_{\text{list_length}} = 2$, but $|f(a)|_{\text{list_length}} = \perp$. Then the size properties of general terms can be described using two functions based on $|\cdot|$. Given a set of terms S, a substitution θ is said to be *S-grounding* if $\theta(t)$ is a ground term for every term t in S. The function $size_m(t)$ defines the size of a term t under the measure m:

$$size_m(t) = \begin{cases} n & \text{if } |\theta(t)|_m = n \text{ for every} \\ & \{t\}\text{-grounding substitution } \theta \\ \\ \perp & \text{otherwise.} \end{cases}$$

The function $\mathit{diff}_m(t_1, t_2)$ gives the size difference between two terms t_1 and t_2 under the measure m:

$$\mathit{diff}_m(t_1,t_2) = \begin{cases} d & \text{if } |\theta(t_2)|_m - |\theta(t_1)|_m = d \\ & \text{for every } \{t_1,t_2\}\text{-grounding} \\ & \text{substitution } \theta \\ \\ \perp & \text{otherwise.} \end{cases}$$

Thus,

$$\begin{aligned} &\mathit{diff}_{\text{list_length}}([\mathtt{c}|\mathtt{L}],[\mathtt{a},\mathtt{b}|\mathtt{L}]) = 1, \\ &\mathit{diff}_{\text{term_depth}}(\mathtt{f}(\mathtt{a},\mathtt{g}(\mathtt{X})),\mathtt{X}) = 2, \\ &\mathit{diff}_{\text{term_depth}}(\mathtt{f}(\mathtt{X},\mathtt{Y}),\mathtt{X}) = \perp. \end{aligned}$$

Where the particular measure under consideration is clear from the context in the discussion that follows, we will omit the subscript in the *size* and *diff* functions.

A directed graph $G = (N, E)$, called a *data dependency graph*, is used to represent data dependencies between literals. Here N is a set of nodes and E a set of edges. A node in the graph denotes a literal and is labelled by the set of argument positions in the literal. There is an edge from a node n_1 to a node n_2 if the literal L_2 denoted by n_2 is dependent on the literal L_1 denoted by n_1, i.e. if a variable binding generated by L_1 is used as an input by L_2. The node n_1 is said to be a *predecessor* of the node n_2, and n_2 a *successor* of n_1. The literal in the head is treated specially in the graph: It is divided into two nodes, the *start* node, consisting of the set of bound, or "input" argument positions, which has no predecessor; and the *end* node, consisting of the set of free, or "output" argument positions, which has no successor. The data dependency graphs are induced by the control strategy of the system, and may be inferred via dataflow analysis [2, 5]. The input/output character of argument positions can be likewise inferred [5, 16] or provided by the users. Hereafter we assume that the data dependency graph is given. In the examples that follow, the "mode" of an n-ary literal, i.e. an indication of which of its argument positions are used as input arguments and which are used as output arguments, is indicated by adding a superscript that is an n-tuple over $\{\mathtt{i},\mathtt{o}\}$: an i in the k-th. position of such a tuple indicates that the k-th. argument of the literal is used as an input argument, while an o indicates that it is used as an output argument.

Example 3.1 Consider the following predicate, called with its first argument the input argument:

```
nrev⟨i,o⟩([], []).
nrev⟨i,o⟩([H|L], R) :-
    nrev⟨i,o⟩(L, R1), append⟨i,i,o⟩(R1, [H], R).
```

Let $head_i$ and $body_i^j$ denote the ith argument position in the head and in the jth literal of the body respectively. The data dependency graphs for the clauses are shown in Figure 1. □

Two functions are associated with a data dependency graph G. The function $\mathsf{input}(G, n)$ gives the set of input argument positions in node n; and the function $\mathsf{output}(G, n)$ gives the set of output argument positions in node n. Let L be a literal corresponding to a node n in G, with $\mathsf{input}(G, n) = \{t_1, \ldots, t_m\}$. Let $size_t$ denote the size of (the term occurring in) an input argument position t. The size of an output argument position in a literal depends, in general, on the size of the input argument positions in that literal: let the ith argument position of L be an output argument, then its size is denoted by $\Psi_L^{(i)}(size_{t_1}, \ldots, size_{t_m})$.

Given a data dependency graph G with node set N, let s and e denote the start node and the end node of G, and $B = N - \{s, e\}$ the set of nodes for the body literals. We distinguish between "intra-literal" argu-

ment size relations, which refer to size relations between the argument positions of a single literal, and "inter-literal" argument size relations, which refer to relations between argument positions of different literals. Then $D = \mathsf{output}(G, e) \cup \bigcup_{n \in B} \mathsf{input}(G, n)$ denotes the set of argument positions for which the inter-literal argument size relations need to be computed; and $I = \bigcup_{n \in B} \mathsf{output}(G, n)$ denotes the set of argument positions for which the intra-literal argument size relations need to be computed. We first consider argument positions in D. Let i be an argument position in D and in a node n, and T_i the term in i. If $size(T_i)$ is defined, then $size_i = size(T_i)$, the inter-literal argument size relation for i. Otherwise, $size_i$ depends on the size of a term T_j in an argument position j in one of the predecessors of n. Then $size_i = size_j + diff(T_j, T_i)$ is the inter-literal argument size relation for i.

Example 3.2 Consider the clauses from Example 3.1. Let $head[i]$ denote the size of the ith argument position in the head and $body_j[i]$ in the jth literal of the body. Using *size* and *diff* functions we get the following inter-literal argument size relations:

$$
\begin{aligned}
&body_1[1] = head[1] + diff(\texttt{[H|L]}, \texttt{L}),\\
&body_2[1] = \Psi^{(2)}_{nrev}(body_1[1]),\\
&body_2[2] = size(\texttt{[H]}),\\
&\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(3)}_{append}(body_2[1], body_2[2]).
\end{aligned}
$$

Since the depth of recursion for both `nrev/2` and `append/3` depend on the list length of input, using list-length as measure we have

$$
\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = \Psi^{(2)}_{nrev}(body_1[1]),\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(3)}_{append}(body_2[1], body_2[2]).
\end{aligned}
$$

□

We next consider argument positions in I, the set of output argument positions in the body literals. Let i be an argument position in I and in a node n. If the literal L denoted by n is nonrecursive, then the intra-literal argument size relations for i can be obtained by (recursively) analyzing the predicate for L separately. This does not work for recursive literals: for these, we take the set of argument size relations for the various argument positions in the clause and *normalize* them. This is done as follows: consider a set of argument size relations $\mathcal{E}$ for a clause: let $\mathcal{E}^{\circ}$ denote the intra-literal argument size relations in $\mathcal{E}$, $\mathcal{E}^{\bowtie}$ the inter-literal argument relations in $\mathcal{E}$, and $\mathcal{E}^{\diamond} \subseteq \mathcal{E}^{\bowtie}$ the inter-literal argument relations corresponding to the input argument positions in the body literals. The idea behind normalization is to propagate information about size relations in the body of a clause until we have size relations for the head. The process can be defined as follows: repeatedly apply the following transformations to $\mathcal{E}^{\bowtie}$ until there is no change:

- If $e \in \mathcal{E}^{\diamond}$ is an inter-literal argument size relation $\phi = \psi$ such that there is at least one occurrence of ϕ in the the right hand side of some equation in $\mathcal{E}^{\bowtie}$, then replace each occurrence of ϕ in the right hand sides of equations in $\mathcal{E}^{\bowtie}$ by ψ.

- If $e \in \mathcal{E}^\circ$ is an intra-literal argument size relation $\phi = \psi$ such that there is at least one occurrence of an instance of ϕ in the the right hand side of some equation in $\mathcal{E}^\bowtie$, then replace each occurrence of an instance of ϕ in the right hand sides of equations in $\mathcal{E}^\bowtie$ by the appropriate instance of ψ.

Example 3.3 Consider the following clause from Example 3.1:

$$\texttt{nrev}^{\langle \texttt{i,o} \rangle}\texttt{([H|L], R) :-}$$
$$\quad \texttt{nrev}^{\langle \texttt{i,o} \rangle}\texttt{(L, R1), append}^{\langle \texttt{i,i,o} \rangle}\texttt{(R1, [H], R).}$$

Assume that while processing the call graph of the program in topological order, the output size function for `append/3` was computed as $\Psi^{(3)}_{append}(x, y) = x + y$ (see the Appendix for details). Thus, before normalization, this equation is the only intra-literal argument size relation available, and $\mathcal{E}^\circ = \{\Psi^{(3)}_{append}(x, y) = x + y\}$. From Example 3.2, $\mathcal{E}^\bowtie$ contains the following equations:

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = \Psi^{(2)}_{nrev}(body_1[1]),\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(3)}_{append}(body_2[1], body_2[2]).
\end{aligned}$$

Only the first three equations are relations for input argument positions in body literals and need to be propagated. Thus $\mathcal{E}^\diamond$ contains the following equations:

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = \Psi^{(2)}_{nrev}(body_1[1]),\\
&body_2[2] = 1.
\end{aligned}$$

During normalization, the expression $body_1[1]$ in the equation

$$body_2[1] = \Psi^{(2)}_{nrev}(body_1[1])$$

is replaced by $head[1] - 1$. The expression $\Psi^{(3)}_{append}(body_2[1], body_2[2])$ in equation

$$\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(3)}_{append}(body_2[1], body_2[2])$$

is first replaced by the expression $body_2[1]+ body_2[2]$, after which the expressions $body_2[1]$ and $body_2[2]$ are replaced by $\Psi^{(2)}_{nrev}(head[1]-1)$ and 1, respectively. Further normalization yields no new changes, and the resulting set of equations is:

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = \Psi^{(2)}_{nrev}(head[1] - 1),\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(2)}_{nrev}(head[1] - 1) + 1.
\end{aligned}$$

The last equation expresses the desired size relationship for the head arguments of the clause. □

A clause is said to be *range-restricted* if each variable occurring in an output argument position in the head also occurs either in an input argument position in the head, or in an output argument position in the body (the intuition is that the binding for such a variable is either a term given as an input argument, or a term produced as an ouput argument by a body literal). In addition, a clause is said to be *well-connected* if the expressions in terms of *size* or *diff* functions in each inter-literal argument size relation for it are defined (i.e. not $\perp$). The idea is that if all the inter-literal argument size relations are defined before normalization, then all the inter-literal argument size relations are defined after normalization. The approach given above is applicable to range-restricted and well-connected clauses:

Theorem 3.1 *Normalization of argument size relations terminates for all clauses.* □

Proof Given the inter-literal argument size relations and intra-literal argument size relations for nonrecursive literals, the number of iterations of the transformations applied in normalization is bounded by the height of the data dependency graph. □

Theorem 3.2 *Let G be a data dependency graph for a program P in which all the clauses are range-restricted and well-connected. If the call graph of P is processed in topological order, then for each clause C in P, the following hold after the argument size relations that hold in C are normalized:*

1. *If C is nonrecursive, then the argument sizes for the output argument positions in the head of C are obtained as a closed form function of the sizes of the input argument positions in the head of C;*
2. *if C is simple recursive, then the argument sizes for the output argument positions in the head of C are obtained as a difference equation in terms of the sizes of the input argument positions in the head of C; and*

3. *if C is mutually recursive, then the argument sizes for the output argument positions in the head of C are obtained as a difference equation, which is part of a system of difference equations for mutually recursive clauses, in terms of the sizes of the bound argument positions in the head of C.*

Proof By induction on the number of literals in the body of C. □

After normalization, the size of each argument position in the literals of a nonrecursive clause is in terms of the sizes of the input argument positions in the head. For a simple recursive clause, the result of normalization is a difference equation giving the sizes for output argument positions in the head in terms of the sizes of the input argument positions in the head. Using the nonrecursive clauses as the base cases, we can obtain the boundary conditions for these difference equations. The mechanism discussed in Section 5 can then be used to get (approximate) solutions for the difference equations. These solutions are closed form and in terms of the sizes of the input argument positions in the head. By adding the new closed form argument size relations into the set of intra-literal argument size relations, normalization can be applied once again. After this, the size of each argument position in the head of clause is in terms of the sizes of the input argument positions.

Example 3.4 Consider the clauses from Example 3.1. From Example 3.3, after normalization, we have the following equations for the second clause:

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = \Psi^{(2)}_{nrev}(head[1] - 1),\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(2)}_{nrev}(head[1] - 1) + 1.
\end{aligned}$$

Solving the difference equation $\Psi^{(2)}_{nrev}(head[1]) = \Psi^{(2)}_{nrev}(head[1] - 1) + 1$ with the boundary condition $\Psi^{(2)}_{nrev}(0) = 0$, we get

$$\Psi^{(2)}_{nrev}(n) = n.$$

Adding this equation into $\mathcal{E}^\circ$, the set of intra-literal argument size relations, and applying the normalization once again, we get

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = head[1] - 1,\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = head[1].
\end{aligned}$$

□

The situation in mutually recursive clauses is similar to the situation in simple recursive clauses. The only difference is that we need to solve a system of difference equations for mutually recursive clauses instead of a single difference equation for a simple recursive clause. The mechanism to solve a system of difference equations is also discussed in Section 5.

4 Cost Estimation

Since it is generally not known, in advance, how many of the solutions generated by a predicate will be demanded, a conservative upper bound can be obtained by assuming that all solutions are needed, and that all clauses are executed. Thus, given a predicate `p` defined as

$$\begin{aligned}
&\mathtt{cl}_1 : \ \mathtt{Head}_1 \text{ :- } \mathtt{Body}_1.\\
&\qquad\vdots\\
&\mathtt{cl}_n : \ \mathtt{Head}_n \text{ :- } \mathtt{Body}_n.
\end{aligned}$$

the cost of predicate `p`, written $\mathtt{Cost}_\mathtt{p}$, can be expressed as

$$\mathtt{Cost}_\mathtt{p} \leq \sum_{i=1}^{n} \mathtt{Cost}_{\mathtt{cl}_i}, \tag{1}$$

where $\mathtt{Cost}_{\mathtt{cl}_i}$ denotes the cost of the ith clause $\mathtt{cl}_i$. It is straightforward to take indexing into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses. In considering the computational cost of a clause, the fact that we provide an upper bound on the actual runtime cost allows us to assume that each literal in the body of the clause succeeds. The cost of a clause can thus be bounded by the cost of head unification together with the costs of each of its body literals: given a clause `cl` defined as

$$\mathtt{H} \text{ :- } \mathtt{L}_1, \ldots, \mathtt{L}_m.$$

its cost $\texttt{Cost}_{\texttt{cl}}$ can be expressed as

$$\texttt{Cost}_{\texttt{cl}} \leq \texttt{Cost}_{\texttt{H}} + \sum_{i=1}^{m} (\prod_{j \prec i} \texttt{Sols}_{\texttt{L}_j}) \texttt{Cost}_{\texttt{L}_i}, \qquad (2)$$

where $\texttt{Cost}_{\texttt{H}}$ is the cost of resolving the head of the clause with the literal being solved, and $\texttt{Sols}_{\texttt{L}_j}$ is the number of solutions literal $\texttt{L}_j$ can generate. Here we use $j \prec i$ to denote the predecessors $\texttt{L}_j$ of the literal $\texttt{L}_i$ in the data dependency graph for the clause. Compile-time estimation of the number of solutions a predicate can generate is a nontrivial problem that is beyond the scope of this paper; to simplify the discussion that follows, we restrict ourselves to the simple case where each literal is determinate, i.e. produces at most one solution. Techniques for the static inference of determinacy are discussed by Mellish [16]. In this case, equation (2) simplifies to

$$\texttt{Cost}_{\texttt{cl}} \leq \texttt{Cost}_{\texttt{H}} + \sum_{i=1}^{m} \texttt{Cost}_{\texttt{L}_i}. \qquad (3)$$

Note that our techniques are directly applicable to committed-choice logic programming languages [3, 21, 24], which are intrinsically determinate: the committed choice nature of such languages also permits simplifications to (1) above. It may be possible to do better than this in some cases, e.g. given a clause of the form

$\texttt{H :- Test} \rightarrow \texttt{Alt}_1\texttt{; Alt}_2.$

its cost can be given as $\texttt{Cost}_{\texttt{H}} + \texttt{Cost}_{\texttt{Test}} + \max(\texttt{Cost}_{\texttt{Alt}_1}, \texttt{Cost}_{\texttt{Alt}_2})$. The ideas presented in this paper extend easily to such cases.

There are a number of different metrics that can be used as the unit of cost in these expressions, e.g. the number of resolutions, the number of unifications, or the number of instructions executed. If the cost metric is the number of resolutions, then $\texttt{Cost}_{\texttt{H}}$ is 1; if it is the number of unifications, then $\texttt{Cost}_{\texttt{H}}$ is the arity of H.

Once argument size relations have been determined as described earlier, the estimation of the cost of a predicate is carried out by setting up equations that describe the work done by each clause. These difference equations can be solved by the technique described in the next section.

Example 4.1 Consider the nrev/2 predicate from Example 3.4. We have the following argument size relations for the recursive clause:

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_2[1] = head[1] - 1,\\
&body_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(head[1]) = head[1].
\end{aligned}$$

Thus the cost equation for the recursive clause is obtained as

$$\begin{aligned}
&\texttt{Cost}_{\texttt{nrev}}(n) = \\
&\qquad 1 + \texttt{Cost}_{\texttt{nrev}}(n-1) + \texttt{Cost}_{\texttt{append}}(n-1).
\end{aligned}$$

The first clause yields the equation $\texttt{Cost}_{\texttt{nrev}}(0) = 1$, which serves as the boundary condition. Assume that the analysis of append/3 has determined that $\texttt{Cost}_{\texttt{append}}(n) = n + 1$ (see the Appendix for details of how this is done), then we get the cost equations

$$\begin{aligned}
&\texttt{Cost}_{\texttt{nrev}}(0) = 1,\\
&\texttt{Cost}_{\texttt{nrev}}(n) = 1 + n + \texttt{Cost}_{\texttt{nrev}}(n-1),\ n > 0.
\end{aligned}$$

These equations can be solved as discussed below. □

It should be noted that the principal source of imprecision in our analysis is in the solution of difference equations: for nonrecursive predicates our analysis is quite precise. For many programs, in fact, an analysis of only the nonrecursive predicates yields sufficient speed improvements to make the analysis worthwhile.

5 Solving Difference Equations

Both the argument size relations and cost functions for recursive predicates are in the form of difference equations. In order to evaluate these functions efficiently, it is necessary to transform difference equations into closed-form functions [15, 23]. The automatic solution of arbitrary difference equations is a difficult problem, but reasonable results can be achieved if difference equations are restricted to some common forms [4, 10]. In our case, however, since granularity analysis is expected to be part of a compiler, there is an additional requirement that the solution of such equations be efficient. This forces us to sacrifice precision in some cases. Although automatic algebra systems can solve a wide class of difference equations, a table-driven method is chosen for the sake of efficiency and flexibility.

Consider the following notion of "approximation" over functions over real numbers:

Definition 5.1 Given two functions f_1 and f_2 over the reals, f_1 *approximates* f_2, written $f_1 \sqsubseteq f_2$, if and only if for all $n \in \mathbf{R}$, $f_2(n) \leq f_1(n)$. ▮

Given a difference equation r, let $Sol(r)$ be a function that is a solution to r, if one exists; in general, $Sol(r)$ may not be expressible as a simple closed form expression. Since we are interested only in the computation of upper bounds to the solutions of difference equations, however, we may be satisfied with approximations to $Sol(r)$. To this end, we define the notion of a "granularity analysis structure", as follows:

Definition 5.2 A *granularity analysis structure* G is a 5-tuple $\langle \mathcal{R}, \mathcal{S}, \mathcal{F}, \alpha, \mathsf{soln} \rangle$, where

- $\mathcal{R}$ is a set of difference equations, called the *domain* of G;
- $\mathcal{S}$ is a set of difference equations, called the *approximation set* of G;
- $\alpha : \mathcal{R} \longrightarrow \mathcal{S}$ is a function, called the *approximation function*, such that for every $r \in \mathcal{R}$, if $Sol(r)$ exists then $Sol(\alpha(r)) \sqsubseteq Sol(r)$.
- $\mathcal{F}$ is a set of closed form expressions, called the *solution set*;
- $\mathsf{soln} : \mathcal{S} \longrightarrow \mathcal{F}$ is a function, called the *solution function*, such that $\mathsf{soln}(f) = Sol(f)$ for every $f \in \mathcal{S}$.

▮

Given a granularity analysis structure $G = \langle \mathcal{R}, \mathcal{S}, \mathcal{F}, \alpha, \mathsf{soln} \rangle$, its domain $\mathcal{R}$ is the set of difference equations we are interested in solving. Given a class of programs that we want to perform granularity analysis on, the difference equations obtained expressing the argument size or cost relationships for any program in this class should be in $\mathcal{R}$. The approximation set $\mathcal{S}$ provides a "library" of difference equation schemas that have known (and, hopefully, easily computed) solutions. The approximation function α maps each equation r in the domain $\mathcal{R}$ to a known "library schema" s in the approximation set $\mathcal{S}$ such that the solution to s is an upper bound on the solution to r. The solution set $\mathcal{F}$ is a set of closed form expressions that are solutions to equations in $\mathcal{S}$, and the solution function soln maps each "library schema" s in $\mathcal{S}$ to its solution in $\mathcal{F}$. The idea, then, is to approximate the solution to a difference equation $r \in \mathcal{R}$ by the closed form expression $\mathsf{soln}(\alpha(r))$.

The following example may make the idea clear. Suppose that $\mathcal{S}$ contains the schema s:

$$f(0) = C,$$
$$f(n) = Af(n-k) + B \qquad \text{for } n > 0;$$

and $\mathsf{soln}(s)$ returns $f(n) = (C + B/(A-1))A^{n/k} - B/(A-1)$ as the closed form solution. Consider the predicate `fib/2` defined as

```
fib(0, 0).
fib(1, 1).
fib(M,N) :- M > 1, M1 is M - 1, M2 is M - 2,
     fib(M1,N1), fib(M2,N2), N is N1 + N2.
```

With the number of resolutions as the metric and builtin functions having cost 0, the difference equations obtained for this predicate are

$$\mathtt{Cost_{fib}}(0) = 1,$$
$$\mathtt{Cost_{fib}}(1) = 1,$$
$$\mathtt{Cost_{fib}}(n) =$$
$$\mathtt{Cost_{fib}}(n-1) + \mathtt{Cost_{fib}}(n-2) + 1,\ n > 1.$$

Assuming that the function $\mathtt{Cost_{fib}}$ is monotone, these equations can be simplified to

$$\mathtt{Cost_{fib}}(0) = 1,$$
$$\mathtt{Cost_{fib}}(n) \leq 2\mathtt{Cost_{fib}}(n-1) + 1,\ n > 0.$$

When matched against the above schema, this yields the solution $\mathtt{Cost_{fib}}(n) \leq 2^{n+1} - 1$.

Mutually recursive predicates result in a system of simultaneous difference equations in more than one variable. It is possible in principle to reduce a system of difference equations in more than one variable to a single difference equation in one variable [15]. Consider the system of difference equations in two variables x and y:

$$3nx_n = y_n - y_{n+1} + 1,$$
$$x_n^2 = y_n + y_{n+1} - 1.$$

Substituting $n+1$ for n, we get two more equations

$$3(n+1)x_{n+1} = y_{n+1} - y_{n+2} + 1,$$
$$x_{n+1}^2 = y_{n+1} + y_{n+2} - 1.$$

Eliminating the variable y_n in the first two equations and y_{n+2} in the last two equations, we get

$$2y_{n+1} = x_n^2 - 3nx_n + 2,$$
$$2y_{n+1} = x_{n+1}^2 + 3(n+1)x_{n+1}.$$

Finally, eliminating the variable y_{n+1} we get

$$x_{n+1}^2 + 3(n+1)x_{n+1} = x_n^2 - 3nx_n + 2,$$

a difference equation in only one variable x.

It is easy to verify that a system of linear difference equations can be reduced to linear difference equations in one variable; and a system of linear difference equations with constant coefficients can be reduced to linear difference equations with constant coefficients in one variable. Therefore, the mechanism described above is applicable to both simple recursive and mutually recursive clauses.

If the initial difference equation r cannot be simplified to a form that matches any of the equation schemas in $\mathcal{S}$, then the solution to r is returned as $\lambda x.\infty$, the function that does an infinite amount of work for any size of input (in particular, equations for predicates with nonterminating execution branches do not have solutions). The practical effect of this is that if the system is unable to find a solution to the difference equations for a procedure p, then calls to p are always executed in parallel. This is consistent with our philosophy of "sequentializing a parallel language", where tasks are executed in parallel unless it can be proven that it is better to execute them sequentially. Note that since predicates with nonterminating execution branches are always executed in parallel, termination properties of programs are unaffected by such sequentialization.

From a compilation point of view, we may want to obtain a "threshold input size" for predicates instead of the actual solutions to their cost equations. The idea is that if the cost of a predicate for an input of size n is given by $f(n)$, and the task management overhead on the system under consideration is W, then we wish to obtain a value K such that $n > K$ if and only if $f(n) > W$. From this, we can generate code that conditionally executes tasks in parallel depending on the size of the input. The constant K can be obtained by associating, with each solution f to equations in $\mathcal{R}$ a function g such that $g(W) = K$, where W is the task management overhead and K the threshold input size. The value of $g(x)$ is defined as the least y such that $f(y) > x$: since f is known ahead of time, g can also be computed ahead of time. Thus, once the task management overhead W has been determined for a system, the input threshold for each equation known to the difference equation solver can be determined statically.

6 Soundness

A predicate is called *size-monotonic* if the sizes of outputs are monotonic on the sizes of inputs, and called *cost-monotonic* if the cost of execution is monotonic on the sizes of inputs. Here we assume that all the predicates are size-monotonic and cost-monotonic.[3]

By definition, if $size(T)$ is defined on T and $diff(T_1, T_2)$ is defined on (T_1, T_2), then $size(T) = size(\theta(T))$ and $diff(T_1, T_2) = diff(\theta(T_1), \theta(T_2))$ for any substitution θ. Since the transformations applied during normalization replace equals by equals and all the predicates are size-monotonic, the soundness of argument size relations is reduced to the soundness of difference equation solver. Because difference equation solver always returns an upper bound on the solution to original equations, the soundness of difference equation solver and thus argument size relations are achieved. By assumption, all the predicates are cost-monotonic, the soundness of cost functions follows immediately the soundness of argument size relations.

7 Experimental Results

We have run a series of experiments in granularity control using two existing parallel logic programming systems: ROLOG and &-Prolog. ROLOG is a pure logic programming system based on Kale's reduce-or model[11, 18]; programs are annotated for parallelism by the user. &-Prolog is a parallel Prolog system based on strict- and non-strict independence [7] which uses a modified RAP-WAM abstract machine [6], and where annotations for parallelism can be automatic or user-provided. In the spirit of the "sequentializing parallel programs" philosophy pointed out in the introduction, in both cases granularity control was added to the parallelized programs and speedup measurements performed while running on a Sequent Symmetry multiprocessor.

[3] There are interesting classes of programs, e.g. theorem-provers, that are neither size-monotonic nor cost-monotonic. The techniques described in this paper do not apply to such programs.

A plot of total execution time against grain size is given in Figure 2. We can draw two broad inferences from this figure. The first is that significant speedups can be achieved by proper use of grainsize information. The second, based on the fairly wide "trough" in the curve of execution time vs. grain size, is that it is not essential to be absolutely precise in inferring the best grain size for a problem: there is a reasonable amount of leeway in how precise this information has to be; this suggests that granularity inference can usefully be performed automatically by a compiler.

Tables 1 and 2 show execution times for our benchmarks compiled with no information of task granularity, compared to the case when they are compiled using grain size information inferred by our algorithm. It can be seen from these tables that the runtime overhead incurred by our approach is small, and that granularity analysis can thus yield good speedups. It works better if the task management overhead is relatively high, as in ROLOG, or in systems that involve non-shared memory architectures. If the task creation and management overhead is sufficiently small, it may better not to use granularity control at all (this happens in some cases for &-Prolog). However, such situations can be determined ahead of time, simply by considering the overhead associated with task creation and management.

The runtime overhead incurred by our approach arises from two factors: the maintenance of size information and the grain size tests. It is observed that many predicates can be classified as either parallel predicates or sequential predicates at compile time, so no grain size control is needed for them, and thus no runtime overhead is associated with them. In this case, programs gain large speedups, for example, in our benchmarks matrix multiplication and polygon inclusion. Furthermore, the runtime overhead for predicates that need grain size control can be largely reduced by unfolding grain size tests loop; that is, grain size test is performed at every two or more iterations instead.

Sometimes the runtime overhead does overpass the gains from sequentializing small parallel tasks, for example, in our benchmark flatten. At this time our compiler does not take the runtime overhead into account in deriving the cost functions and inferring the threshold input size, so we get negative result in benchmark flatten. However, we do get positive results in most benchmarks and we shall incoporate the consideration for runtime overhead into our compiler for more precise estimates.

8 Related Work

A number of researchers have investigated the automatic analysis of the (time) complexity of programs (see, for example, [1, 8, 14, 19, 26, 27]). Our work differs from these in two main ways: first, it is not sufficient for us to infer asymptotic complexities, because without information about the constants involved it is not possible to estimate the amount of computation that might be involved in solving a problem; second, since our analyses are intended to be performed at compile time, it is essential that they be efficient, and because of this we occasionally sacrifice some accuracy to obtain fast analyses.

In Metric [27], an automatic program analysis system, Wegbreit proposes a very general and flexible framework for program performance analysis that can compute the best-case, worst-case and expected complexities, and incorporate new measures and difference equations solving techniques to increase its precision. However, this system is unable to deal with mutually recursive programs.

There are two main differences between logic programs [13] and functional programs [1, 8, 14, 19, 26] with regard to complexity analysis. The nondeterminism in logic programs makes the composition of cost functions much harder than that in functional programs, although we can not deal with nondeterministic programs either at this point. The possibilities of partial instantiation of output arguments in logic programs enable an interesting class of problems to be implemented in a way that can not be realized in functional programs. These partially instantiated data structures make the inference of argument size relations much difficult, but it is possible that some of the programs whose costs depend only on the top level structural properties of data structures can be analyzed using our techniques.

Rabhi and Manson have recently investigated the use of complexity functions to control parallelism in the parallel evaluation of functional programs [17]. They also obtained encouraging experimental results for divide-and-conquer algorithms on a parallel graph reduction machine.

The problem of program partitioning has been considered, in the context of functional programs, by Sarkar, who bases his algorithm on information obtained via runtime profiling rather than compile-time analysis [20]. Hudak considers "serial combinators"

with reasonable grain sizes [9], but does not discuss the compile time analysis necessary to estimate the amount of work that may be done by a call.

Kaplan has investigated the automatic inference of the complexity of logic programs [13], but under fairly restrictive assumptions that rule out many interesting programs (e.g. Kaplan's techniques do not permit granularity analysis of quicksort). Van Gelder has investigated a different approach to reasoning about the argument sizes of predicates, but this approach has been restricted to simple linear recursive programs [25].

Tick has recently given a simple algorithm to estimate relative granularities to guide task scheduling decisions for parallel logic programs [22], but his analysis is not expressive enough to enable a compiler to generate code of the form "if the input size exceeds this threshold then execute these goals in parallel, else do them sequentially".

9 Conclusions

Parallel logic programming languages offer a great deal of scope for parallelism. However, because of the overhead associated with task creation and management, the "work available" underneath a goal should be taken into account when deciding whether or not to execute it concurrently as a separate task. This paper describes how programs may be statically analyzed to estimate task granularities. The analysis can handle a large class of recursive predicates. The granularity information inferred can usefully be utilized by compilers to improve the quality of code generated. The runtime overhead associated with our approach is usually quite small.

10 Acknowledgements

This paper benefited from discussions with Evan Tick on the issue of granularity analysis of logic programs. We also thank L. V. Kale and B. Ramkumar for makeing availabe their ROLOG system for our experiments, and for many helpful discussions and suggestions. The first two authors were supported in part by the National Science Foundation under grant number CCR-8702939.

References

[1] B. Bjerner and S. Holmström, "A Compositional Approach to Time Analysis of First Order Lazy Functional Programs," *Proc. 1989 ACM Functional Programming Languages and Computer Architecture*, pp. 157-165.

[2] J. Chang, A. M. Despain and D. DeGroot, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," in *Digest of Papers, Compcon 85*, IEEE Computer Society, Feb. 1985.

[3] K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Languages and Systems* 8, 1 (Jan. 1986), pp. 1-49.

[4] J. Cohen and J. Katcoff, "Symbolic Solution of Finite-Difference Equations," *ACM Transactions on Mathematical Software* 3, 3 (Sep. 1977), pp. 261-271.

[5] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs," *ACM Transactions on Programming Languages and Systems* 11, 3 (July 1989), pp. 419-450.

[6] M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD thesis, The University of Texas at Austin, 1986.

[7] M. Hermenegildo and F. Rossi., "On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs," *1989 North American Conference on Logic Programming*, MIT Press, October 1989, pp. 369-389.

[8] T. Hickey and J. Cohen, "Automating Program Analysis," *J. ACM* 35, 1 (Jan. 1988), pp. 185-220.

[9] B. Goldberg and P. Hudak, "Serial Combinators: Optimal Grains of Parallelism," *Proc. Functional Programming Languages and Computer Architecture*, Nancy, France, Aug. 1985. Springer-Verlag LNCS vol. 201, pp. 382-399.

[10] J. Ivie, "Some MACSYMA Programs for Solving Recurrence Relations," *ACM Transactions on Mathematical Software* 4, 1 (March 1978), pp. 24-33.

[11] L. V. Kalé, *Parallel Architectures for Problem Solving*, PhD Thesis, SUNY, Stony Brook, 1985.

[12] L. V. Kalé, "Completeness and Full Parallelism of Parallel Logic Programming Schemes," *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, California, IEEE, 1987, pp. 125-133.

[13] S. Kaplan, "Algorithmic Complexity of Logic Programs," *Logic Programming, Proc. Fifth International Conference and Symposium*, Seattle, Washington, 1988, pp. 780-793.

[14] D. Le Métayer, "ACE: An Automatic Complexity Evaluator," *ACM Transactions on Programming Languages and Systems* 10, 2 (April 1988), pp. 248-266.

[15] H. Levy and F. Lessman, *Finite Difference Equations*, Sir Isaac Pitman & Sons, London, 1959.

[16] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *J. Logic Programming* 2, 1 (April 1985), pp. 43-66.

[17] F. A. Rabhi and G. A. Manson, "Using Complexity Functions to Control Parallelism in Functional Programs," Research Report CS-90-1, Department of Computer Science, University of Sheffield, England, Jan. 1990.

[18] B. Ramkumar and L. V. Kalé, "Compiled Execution of the Reduce-OR Process Model on Multiprocessors," *1989 North American Conference on Logic Programming*, MIT Press, October 1989, pp. 313-331.

[19] M. Rosendahl, "Automatic Complexity Analysis," *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, 1989, pp. 144-156.

[20] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, Massachusetts, 1989.

[21] E. Y. Shapiro, "A Subset of Concurrent Prolog and its Interpreter," Technical Report CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Israel, Feb. 1983.

[22] E. Tick, "Compile-Time Granularity Analysis for Parallel Logic Programming Languages," *International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1988, pp. 994-1000.

[23] A. Tucker, *Applied Combinatorics*, John Wiley & Sons, New York, 1985.

[24] K. Ueda, *Guarded Horn Clauses*, D. Eng. Thesis, University of Tokyo, 1986.

[25] A. Van Gelder, "Deriving Relations Among Argument Sizes in Logic Programs," Unpublished memorandum. Department of Computer Science, Stanford University, Stanford, Califonia, 1986.

[26] P. Wadler, "Strictness Analysis Aids Time Analysis," *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, 1988, pp. 119-132.

[27] B. Wegbreit, "Mechanical Program Analysis," *Communications of the ACM* 18, 9 (Sep. 1975), pp. 528-539.

A Appendix: An Example

This appendix considers in detail the analysis of the "naive reverse" program. This example has been chosen because it is simple, yet shows the interaction of different recursive predicates. The program is as follows:

```
nrev([], []).
nrev([H|L], R) :-
     nrev(L, R1), append(R1, [H], R).

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

First, consider the predicate `append/3`, called with the first two arguments as input arguments. Let $head[i]$ and $body_j[i]$ denote the ith argument position in the head and in the jth literal of the body respectively. The inter-literal argument size relations for the recursive clause are

$$\begin{aligned}
&body_1[1] = head[1] - 1,\\
&body_1[2] = head[2],\\
&\Psi^{(3)}_{append}(head[1], head[2]) =\\
&\qquad \Psi^{(3)}_{append}(\underline{body_1[1]}, \underline{body_1[2]}) + 1,
\end{aligned}$$

where the expressions being substitued for during normalization are underlined for clarity. Normalization then yields the equations

$$\begin{aligned}
&\mathit{body}_1[1] = \mathit{head}[1] - 1,\\
&\mathit{body}_1[2] = \mathit{head}[2],\\
&\Psi^{(3)}_{append}(\mathit{head}[1], \mathit{head}[2]) =\\
&\qquad \Psi^{(3)}_{append}(\mathit{head}[1] - 1, \mathit{head}[2]) + 1.
\end{aligned}$$

In addition, from the first clause of `append`, we obtain the equation $\Psi^{(3)}_{append}(0, \mathit{head}[2]) = \mathit{head}[2]$. Thus, the equations expressing the size of the output argument as a function of the input argument sizes are seen to be of the form

$$\begin{aligned}
&f(0, y) = y,\\
&f(n, y) = 1 + f(n - 1, y).
\end{aligned}$$

These equations can be solved, e.g. using the techniques of Section 5, to obtain

$$\Psi^{(3)}_{append}(x, y) = x + y.$$

This equation is now used as an intra-literal argument size relation for the predicate `nrev`.

Next, consider the predicate `nrev`/2, called with the first argument as the input argument and the second as an output argument. The inter-literal argument size relations obtained initially for the recursive clause are

$$\begin{aligned}
&\mathit{body}_1[1] = \mathit{head}[1] - 1,\\
&\mathit{body}_2[1] = \Psi^{(2)}_{nrev}(\underline{\mathit{body}_1[1]}),\\
&\mathit{body}_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(\mathit{head}[1]) = \underline{\Psi^{(3)}_{append}(\underline{\mathit{body}_2[1]}, \underline{\mathit{body}_2[2]})},
\end{aligned}$$

together with the (intra-literal) argument size relation $\Psi^{(3)}_{append}(x, y) = x + y$ obtained above. When normalized (again the expressions being substituted for are underlined), this yields the equations

$$\begin{aligned}
&\mathit{body}_1[1] = \mathit{head}[1] - 1,\\
&\mathit{body}_2[1] = \Psi^{(2)}_{nrev}(\mathit{head}[1] - 1),\\
&\mathit{body}_2[2] = 1,\\
&\Psi^{(2)}_{nrev}(\mathit{head}[1]) = \Psi^{(2)}_{nrev}(\mathit{head}[1] - 1) + 1.
\end{aligned}$$

From the first clause for `nrev`/2, we obtain $\Psi^{(2)}_{nrev}(0) = 0$. Thus, the output argument size function for `nrev`/2 is given by the equations

$$\begin{aligned}
&\Psi^{(2)}_{nrev}(0) = 0,\\
&\Psi^{(2)}_{nrev}(n) = 1 + \Psi^{(2)}_{nrev}(n - 1).
\end{aligned}$$

The solution to this is $\Psi^{(2)}_{nrev}(n) = n$, i.e. the size of the output of `nrev`/2 is equal to the size of its input.

This shows how normalization of argument size relations can be used to track argument sizes. The cost analysis for these predicates now proceeds as follows: first, we consider the clauses defining `append`/3. For this predicate, it can be seen that when the first argument is bound to a list, indexing can be used to select between the two clauses directly. This yields the equations

$$\begin{aligned}
&\mathtt{Cost}_{\mathtt{append}}(0, y) = 1 \text{ (the cost of head unification)},\\
&\mathtt{Cost}_{\mathtt{append}}(n, y) = 1 + \mathtt{Cost}_{\mathtt{append}}(n - 1, y).
\end{aligned}$$

These equations can be solved to yield

$$\mathtt{Cost}_{\mathtt{append}}(n, y) = n + 1.$$

This is then applied to the clauses defining `nrev`/2, together with argument size information obtained earlier, yielding the inter-literal cost equations

$$\begin{aligned}
&\mathtt{Cost}_{\mathtt{nrev}}(0) = 1,\\
&\mathtt{Cost}_{\mathtt{nrev}}(n) =\\
&\qquad 1 + \mathtt{Cost}_{\mathtt{nrev}}(n - 1) + \mathtt{Cost}_{\mathtt{append}}(n - 1, 1),
\end{aligned}$$

together with the intra-literal cost equation

$$\mathtt{Cost}_{\mathtt{append}}(n, y) = n + 1.$$

Normalization of these equations yields

$$\begin{aligned}
&\mathtt{Cost}_{\mathtt{nrev}}(0) = 1,\\
&\mathtt{Cost}_{\mathtt{nrev}}(n) = 1 + \mathtt{Cost}_{\mathtt{nrev}}(n - 1) + n.
\end{aligned}$$

These equations can then be solved to obtain the cost function

$$\mathtt{Cost}_{\mathtt{nrev}}(n) = 0.5n^2 + 1.5n + 1.$$

Figure 2: Execution time vs. task granularity

ROLOG on 4 processors (Symmetry)			
programs	$\mathbf{T_0}$ **(ms)**	$\mathbf{T_1}$ **(ms)**	**speedup**
consistency(500)	820	560	31.7%
fib(15)	1170	850	27.3%
hanoi(6)	270	240	11.1%
quick-sort(75)	600	580	3.3%
LR(1)-set(3)	1264	1241	2.0%
double-sum(2048)	2660	2259	15.1%
fft(256)	2760	2636	4.5%
flatten(536)	1161	1387	-19.5%
matrix-multi(8)	575	250	56.5%
merge-sort(128)	2226	1912	14.1%
poly-inclusion(30)	8979	5537	38.3%
tree-traversal(8)	1890	1832	3.0%
$\mathbf{T_0}$: execution time with no granularity control.			
$\mathbf{T_1}$: execution time with granularity control.			

Table 1: Execution times for benchmarks on ROLOG

&-Prolog on 4 processors (Symmetry)			
programs	$\mathbf{T_0}$ **(ms)**	$\mathbf{T_1}$ **(ms)**	**speedup**
consistency(500)	139	139	0%
fib(15)	277	196	29.2%
hanoi(6)	69	80	-15.9%
quick-sort(75)	111	93	16.2%
$\mathbf{T_0}$: execution time with no granularity control.			
$\mathbf{T_1}$: execution time with granularity control.			

Table 2: Execution times for benchmarks on &-Prolog

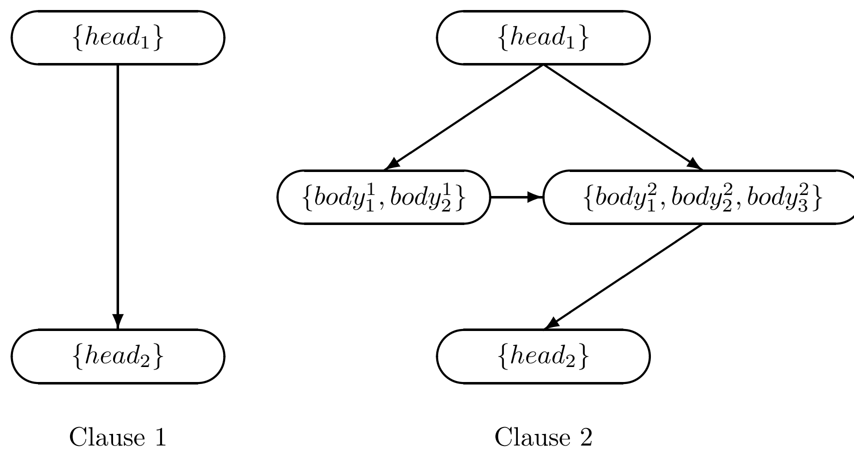

Figure 1: Data dependency graphs for the clauses of predicate `nrev/2`.