Efficient Term Size Computation for Granularity Control [1]

M. Hermenegildo and **P. López-García**
herme@fi.upm.es *pedro@dia.fi.upm.es*
Computer Science Department
Technical University of Madrid (UPM), Spain

Abstract

Knowing the size of the terms to which program variables are bound at run-time in logic programs is required in a class of optimizations which includes granularity control and recursion elimination. Such size is difficult to even approximate at compile time and is thus generally computed at run-time by using (possibly predefined) predicates which traverse the terms involved. We propose a technique which has the potential of performing this computation much more efficiently. The technique is based on finding program procedures which are called before those in which knowledge regarding term sizes is needed and which traverse the terms whose size is to be determined, and transforming such procedures so that they compute term sizes "on the fly". We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We also discuss the advantages and applications of our technique (specifically in the task of granularity control) and present some performance results.

Keywords: Granularity Analysis and Control, Parallelism, Term Size Computation.

1 Introduction

The need to know the size of the terms to which program variables are bound at run-time in logic programs arises in a class of applications related to program optimization which includes recursion elimination, granularity control, and selection among different algorithms or control rules whose performance may be dependent on such size. By term size we refer to measures such as list length, term depth, number of nodes in a term, etc. We address the problem of term size calculation, with special emphasis on its application in granularity control. We start by describing this application in more detail, since it is the fundamental motivation of our work.

It has been shown (see e.g. [6]) that several types of parallelism can be exploited in logic programs while preserving correctness (i.e. the parallel execution obtains the same results as the sequential) and efficiency (i.e. the amount of work performed is not greater or, at least, there is no slow-down). However, such results assume an idealized execution environment in which a number of practical overheads are ignored, such as those associated with task creation, possible task migration of tasks to remote processors, the associated communication overheads, etc. Due to these

[1] The work presented in this paper is supported in part by ESPRIT project 6707 "PARFORCE" and CICYT project number TIC93-0976-CE. The authors would like to thank M. Bruynooghe, Saumya Debray, Lee Naish and all the members of the CLIP group (Computational Logic Implementation and Parallelism) at the Computer Science Department of Technical University of Madrid (UPM) for useful comments.

overheads, and if the granularity of parallel tasks, i.e. the "work available" underneath them, is too small, it may happen that the costs are larger than the benefits in their parallel execution. This makes it desirable to devise a method whereby the granularity of parallel goals and their number can be controlled. Granularity control has been studied in the context of traditional programming [10, 12], functional programming [7], and also logic programming [9, 3, 18, 11].

The aim of granularity control is to change parallel execution to sequential execution or vice-versa based on some conditions related to grain size and overheads. However, granularity control itself can induce new overheads, which should obviously be minimized. As pointed out in [3], granularity analysis for a set of non-recursive procedures is relatively straightforward. However, recursive procedures are somewhat more problematic: the amount of work done by a recursive call depends on the depth of recursion, which in turn depends on the size of the input. Reasonable estimates for the granularity of recursive predicates can thus be made only with some knowledge of the size of the input. In [3] a technique was presented for solving this problem in the context of logic programs. In [11] a complete granularity control system for logic programs based on these ideas is described. The technique is based on performing a compile-time analysis which reduces granularity analysis work at run-time to evaluating simple functions of term sizes. However, the actual determination of those sizes in order to evaluate such functions is necessarily postponed until runtime. A similar technique has been also proposed by Rabhi and Manson in the context of functional programs [15]. An alternative is to determine only the *relative* cost of goals [18], which can be useful for optimizing an on-demand run-time scheduler, but may not be as effective in reducing task creation cost.

The postponement of accurate term size computation to run-time appears inevitable in general. This based on the fact that even sophisticated compile-time techniques such as abstract interpretation are based on computing approximations of variable substitutions for generic executions corresponding to general classes of inputs. In contrast, size is clearly a quite specific characteristic of an input. Although the approximation approach can be useful in some cases we would like to tackle the more general case in which actual sizes have to be computed dynamically at run-time. Of course computing term sizes at run-time is quite simple but at the same time it can involve a significant amount of overhead. This overhead includes both having to traverse significant parts of the term (often the entire term) and the counting process done during this traversal.

The objective of this paper is to propose a novel and more efficient way of computing such sizes. The essential idea is based on the observation that terms are often already traversed by procedures which are called in the program before those in which knowledge regarding term sizes is needed, and thus that such sizes can often be computed "on the fly" by the former procedures after performing some transformations to them. While the counting overhead is not eliminated, overhead is reduced because additional traversals of terms are not needed. We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We have omitted proofs for the sake of conciseness. They can be found in [5].

2 Overview of the Approach

As mentioned in the introduction, we are interested in transforming some predicates in such a way that they will compute some of their argument data sizes at run-time, in addition to performing their normal computation. It is often the case that because of previous transformations or other reasons, the size of certain terms is already known and it can be used as a starting point in the dynamic computation of those that we need to determine at a given point. Thus, we will be interested in the general problem of transforming programs to determine the sizes of one set of terms given that the sizes of the terms in another (disjoint) set are known. For example, consider the predicate `append/3`, defined as:

```
append([], L, L).
append([H|L], L1, [H|R])  :-  append(L, L1, R).□
```

Suppose that we want to transform this predicate in such a way that it computes the length of its third argument. Observing the base case we can infer that the length of the term appearing in the third argument of the head is equal to that of the term appearing in the second argument after any successful computation. We can express this size relation as follows: $head[3] = head[2]$, where $head[i]$ denotes the size of the term appearing at the i^{th} argument position in the head. Thus, a transformation of this base case can be performed by adding two additional arguments, which stand for the size of the term appearing in the second and third arguments, respectively: `append3i2([], L, L, S, S).`

In this way, if we call the base case supplying the size of the second argument, we will obtain that of the third one. Observing the recursive clause, we can see that the size of the third argument of the head is equal to the size of the third argument of the first body literal plus one. We express this size relation as follows: $head[3] = body_1[3] + 1$, where $body_j[i]$ denotes the size of the term appearing at i^{th} argument position in the j^{th} literal of the body (literals are numbered from left to right, starting by assigning "1" to the literal just after the head). Then we can think of using a transformed version of this body literal in order to compute $body_1[3]$. But to do this it is necessary that the size of the second argument of this body literal ($body_1[2]$) be supplied at the call (so that $body_1[3]$ can be computed when recursion finishes). Since we already have the $body_1[2] = head[2]$ size relation, we can conclude that it is possible to compute the size of the third argument of `append/3` if the size of the second one is supplied at the call.

The recursive clause can be trivially transformed as follows with the knowledge of the previous size relations:[2]

```
append3i2([],L,L,S,S).
append3i2([H|L],L1,[H|R],S2,S3) :-  append3i2(L,L1,R,S2,Sb3),
                                    S3 is Sb3 + 1.
```

We can see that the problem can be reduced to finding what we will call a "size dependency graph" for each clause of the predicate to be transformed. Figure 1 shows the size dependency graphs corresponding to the previous example. In this figure, the graphs `G2` and `G1` correspond to the base case and recursive clause of `append/3` respectively.

Informally, the set of size dependency graphs contains the information needed to transform a predicate, and is represented by means of what we call a *transformation node*. In general it is necessary to transform more than one predicate to perform a particular size computation. In this case, transformation nodes are viewed as nodes

[2]For clarity, this class of transformations is used in the examples even if they are not ideal, given that they destroy tail recursion optimization. However it is quite straightforward to perform the equivalent transformation which preserves tail recursion optimization by using an accumulating parameter. These are the transformations performed in practice. Note also that although presenting the technique proposed in terms of source-to-source transformations is useful both didactically and as a viable implementation technique, the transformation can also be implemented at a lower level in order to reduce the run-time overheads involved even further.

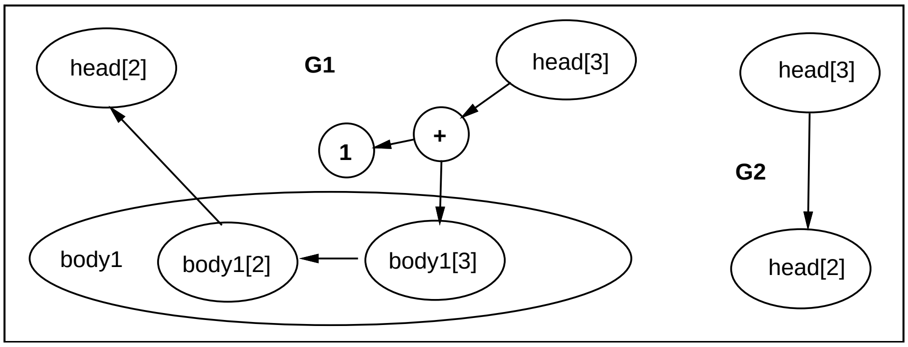

Figure 1: Size dependency graphs for predicate `append/3`.

in a search tree which will have to be explored with the objective of finding a set of such nodes leading to a program transformation which correctly computes the desired term sizes.

In essence, the proposed approach involves first inferring all possible size relations between arguments of the program clauses which can be involved in the desired size computation,[3] constructing all possible transformation nodes from these size relations, and, finally, finding the set of transformation nodes leading to correct size computations.

The static inference of argument size relations have been widely studied [16, 17, 3]. In particular, we refer to the size relations described in [3]. Consider the function $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ (as defined in [3]), that maps ground terms to their sizes under a specific measure m (various measures can be used, e.g., term-size, term-depth, list-length, integer-value, etc.), 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". For example, $|[\mathtt{a},\mathtt{b}]|_{\mathrm{list\_length}} = 2$, but $|f(a)|_{\mathrm{list\_length}} = \perp$. In [3], argument size relations are classified as either "intra-literal" or "inter-literal". The former refer to size relations between the argument positions of a single literal. They hold between the sizes of arguments of all atoms in the success set for the predicate corresponding to the literal and are similar to those described in [17]. The latter refer to relations between argument positions of different literals in a clause or the clause head. For example $size_3 = size_1 + size_2$ is an intra-literal size relation for the predicate `append/3` which states that the length of its third argument is the sum of the lengths of its two first arguments. However $head[3] = body_1[3] + 1$ is an inter-literal size relation corresponding to the recursive clause of `append/3` , and states that for every substitution that makes the terms appearing at positions $head[3]$ and $body_1[3]$ ground, the size of the term appearing at position $head[3]$ is equal to the size of the term appearing at position $body_1[3]$ plus one, i.e. $|\ \mathtt{[H|R]}\ |_{\mathrm{list\_length}} = |\ \mathtt{R}\ |_{\mathrm{list\_length}} + 1$ holds for every substitution that makes `H` and `R` ground.

3 Transforming Procedures

A *size dependency graph* is a directed, acyclic graph whose nodes can be of the following types: **a)** A *position* in a clause: $head[i]$ or $body_j[i]$, as described in Section 2; **b)** A binary *arithmetic operator* ($+$, $-$, etc.); or **c)** A non-negative integer *number*.

We distinguish two classes of edges:

[3]We can consider only predicates in the strongly connected component of the call graph corresponding to the predicate which is the entry point of the transformation.

- *Intra-literal* edges are those from a position in a body literal to another position in the same body literal, more formally, from $body_i[k]$ to $body_j[n]$ where $i = j$ and $k \neq n$. Their meaning is the following: the size of the term appearing at the k^{th} argument position in the i^{th} literal of the body is computed by a transformed version of the predicate of this literal. In order to perform such size computation this version requires that the size of the term appearing at its n^{th} argument position be supplied at the call.
- *Inter-literal* edges are those which are not intra-literal.

There is an inter-literal edge from a position x to another position y, if the size of the term appearing at position x is equal to the size of the term appearing at position y. Arithmetic operator nodes and number nodes are used to express arithmetic relations between the size of argument positions, as illustrated in Figure 1. Regarding the number and type of outgoing and incoming edges allowed, we establish a classification of nodes as follows:

- Only two cases are allowed for head positions nodes, namely:
  - Input size nodes, which have one or more inter-literal incoming edges and no outgoing edges.
  - Output size nodes, which have exactly one outgoing inter-literal edge and no incoming edges.
- For body positions, also only two cases are allowed, namely:
  - Supplied size nodes, which have one outgoing inter-literal edge and one or more incoming intra-literal edges. They correspond to those arguments whose size is supplied at the call of a transformed body literal.
  - Computed size nodes, which have one or more incoming inter-literal edges and zero or more outgoing intra-literal edges. They correspond to those arguments whose size is computed by transformed body literals.
- A binary arithmetic operator node has two outgoing inter-literal edges and one incoming inter-literal edge.
- A non-negative integer number node has only one inter-literal incoming edge and no outgoing edges.

Consider the size dependency graph `G1` in Figure 1. $head[2]$ is an input size node, $head[3]$ is an output size node, $body_1[2]$ is a supplied size node and $body_1[3]$ is a computed size node. A *transformation node* for a predicate $Pred$ is a pair $(Label, Graphs)$, where $Graphs$ is a set of size dependency graphs. There is exactly one graph for each clause defining the predicate. Suppose that there are n clauses in the definition of predicate $Pred$. Let G_i be the size dependency graph for clause i, and I_i and O_i the set of input and output size arguments of G_i respectively. Let $I = \bigcup_{i=1}^{n} I_i$ and $O = \bigcup_{i=1}^{n} O_i$. Then $Label$, the label of the transformation node, is a tuple $(Pred, Is, Os)$, where $Is = \{i \mid head[i] \in I\}$ and $Os = \{i \mid head[i] \in O\}$. With the above defined label we can express which predicate $Pred$ is transformed and which argument sizes will be computed as a function of which others. The transformed version of $Pred$ will have an additional argument for each item $i \in Is$ (which will be bound to the size of the term appearing at the i^{th} argument position in the head at the predicate call) and $j \in Os$ (which will be bound to the size of the term appearing at the j^{th} argument position in the head once the call succeeds). For example, $(append/3, \{2\}, \{3\})$ is a label which states that the predicate `append/3` will be transformed to compute the size of its third argument, provided that the size of the second one is supplied at the procedure call. This means that it is necessary

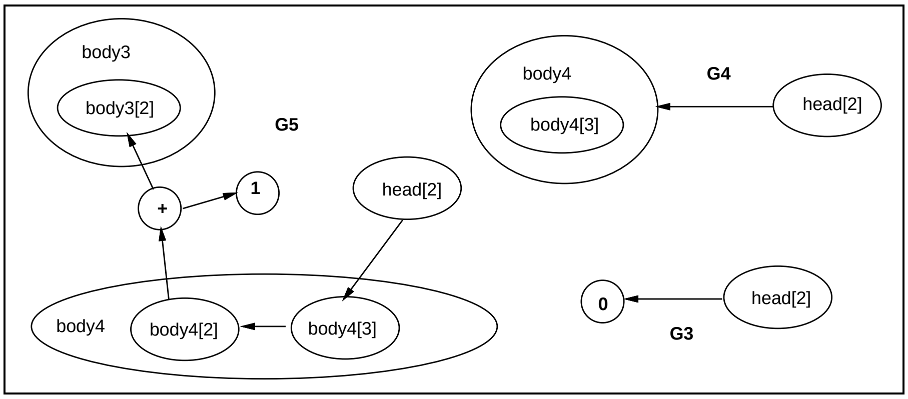

Figure 2: Size dependency graphs for predicate `qsort/2`.

to add two extra arguments to the transformed predicate which will stand for the sizes of the second and third arguments of `append/3`.

Example 3.1 Figure 1 represents the transformation node composed by the size dependency graphs `G1` and `G2`, namely $((append/3, \{2\}, \{3\}), \{\texttt{G1}, \texttt{G2}\})$. $\square$

We require that the size dependency graphs meet the following condition: if there is an inter-literal edge from a supplied size node $body_i[k]$ to a computed size node $body_j[n]$ then $j < i$. This condition ensures that the sizes supplied to a transformed literal are computed only by previous literals of the body. This requirement is due to the fact that the sizes supplied have to be "ground" at the call, because we are interested in using built-ins similar to "is/2" (in fact, more efficient and specialized versions) to perform the arithmetic operations needed to compute sizes and these built-ins require all but one of their arguments to be ground. It is important to note that this condition may be relaxed if the target language is for example a Constraint Logic Programming language [8] which can solve linear equations. However actual equation solving would probably incur in significant overhead. Thus we enforce the condition both for efficiency reasons and for allowing the transformed programs to be executed without requiring any constraint solving capabilities in the target language.

In a size dependency graph the set of all the nodes corresponding to a literal with number i (i.e. those of the form $body_i[j]$) is referred to as the *literal node* $body_i$. As an example, consider the size dependency graph `G1` in Figure 1. There, the set $\{body_1[2], body_1[3]\}$ is the literal node $body_1$. We also group the supplied size nodes and computed size nodes corresponding to a particular literal node into the sets S and C respectively (in the example $S = \{body_1[2]\}$ and $C = \{body_1[3]\}$). We associate with the literal node the label $(Pred, Is, Os)$, where $Pred$ is the predicate name and arity of the literal and $Is = \{j \mid body_i[j] \in S\}$ and $Os = \{j \mid body_i[j] \in C\}$ (in the example, the label associated with literal node $body_1$ is $(append/3, \{2\}, \{3\})$). The label of the literal node indicates which transformed version of the predicate of the literal corresponds to such literal. This is the version which performs the size computation that is also expressed by such label. Then, when the clause where the literal appears is transformed, the literal will be replaced by a call to the predicate that performs the size computation.

4 Transforming Sets of Procedures

In this section we address the problem of transforming a set of procedures which are part of a call-graph, in order that they perform a size computation. To this end, it is necessary to have at least a transformation node for some of those procedures and these nodes have to meet some conditions that are explained below.

Definition 4.1 [Transformation] Is a graph composed by a set N of transformation nodes and a set of edges. There is a distinguished transformation node $E \in N$ which is called the *entry point* of the transformation and:

1. Let G be any size dependency graph of T_1, where T_1 is a transformation node $T_1 \in N$, and let l be any literal node of G, then l has exactly one outgoing edge and no incoming edges. This edge goes from l to some transformation node $T_2 \in N$ such that the label of T_2 is equal to the label associated with the literal node l (note that T_1 and T_2 can be the same transformation node). The intuition behind this edge is the following: suppose that L_1 is the literal corresponding to l in the source clause corresponding to G, and L_2 is the transformed version of L_1 which perform the size computation indicated by the label associated with l. The edge states that the predicate of L_1 can be transformed according to the information represented in T_2 yielding the predicate of L_2.
2. There is an edge from transformation node $T_1 \in N$ to a transformation node $T_2 \in N$ if and only if there is an edge from some literal node l of T_1 to T_2. Intuitively, this edge states that the transformed predicate corresponding to T_1 calls the transformed predicate corresponding to T_2.
3. All the transformation nodes $T \in N$ are reachable from E. $\Box$

Definition 4.2 [Size Computation Specification] We define a *size computation specification* as a pair $(Pred, Os)$, where $Pred$ is the name and arity of the predicate to be transformed, and Os is a set of argument numbers whose sizes are computed by the transformed predicate at run-time. $\Box$

Definition 4.3 [Transformation for a size computation specification] A Transformation for a size computation specification $(Pred, Os)$ is a transformation T such that the label of the entry point of T is of the form $(Pred, Is, Os)$. $\Box$

Theorem 1 *If there is a Transformation T for a size computation specification $(Pred, Os)$ such that the label of the entry point of T is $(Pred, Is, Os)$ then it is possible to transform the clauses of $Pred$ to obtain a transformed Predicate $Pred'$, such that $Pred'$ computes the sizes of the arguments indicated in Os, provided that the sizes of arguments indicated in Is are supplied (while still also performing the same computations originally performed by $Pred$).* $\Box$

5 Irreducible/Optimal Transformations

Since there may be many possible transformations for a given size computation specification, we are interested in those involving the least amount of overhead at run-time. Such overhead is dependent on the system, since it depends on the cost of argument passing and that of arithmetic operations. Reducing this overhead suggests considering transformations having the minimum number of transformation nodes and each of them having the minimum number of items in Is, where $(Pred, Is, Os)$ is the label of any node in the transformation. That is, to transform a predicate to make it compute the sizes of some of its arguments we would like to know which are the arguments whose sizes are strictly necessary to perform this computation (in order to add only the absolutely necessary additional arguments and operations to the transformed predicates) and also what is the minimum number of predicates which have to be transformed. We first introduce the concept of *irreducible transformation* and show that in order to determine whether it is possible to transform a predicate we only need to consider irreducible transformations. Then we present some ideas regarding the generation of optimal irreducible transformations.

Definition 5.1 [Ordering between labels] Given two labels, $X = (Pred, Is_x, Os)$ and $Y = (Pred, Is_y, Os)$, we say that $X <_l Y$ if and only if $Is_x \subset Is_y$. □

For example: $(append/3, \{2\}, \{3\}) <_l (append/3, \{1,2\}, \{3\})$, but
$(append/3, \{2\}, \{3\}) \not<_l (append/3, \{1\}, \{3\})$

Definition 5.2 [Irreducible Transformation] A transformation T is *irreducible* iff:

1. The labels of transformation nodes in T are unique.
2. There are no two transformation nodes in T, labeled with the labels X and Y respectively, such that $X <_l Y$. □

We represent an irreducible transformation as a pair (L, T), where T is a set of transformation nodes and L is the label of the transformation node that is the entry point of the transformation (recall that the labels of the transformation nodes in T are unique). The entry point belongs to the set T. Since the labels of the transformation nodes are unique, it is not necessary to explicitly represent any edges in the irreducible transformation (they can be determined from conditions in Definition 4.3 without ambiguity). Thus, all edges are omitted.

Example 5.1 Consider the predicate qsort/2 defined as follows:

```
C1: qsort([],[]).
C2: qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        qsort(Ls,Ls2),qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

and suppose we want to transform it to compute the length of its second argument. Figure 2 shows size dependency graphs corresponding to the clauses of predicate qsort/2. In this figure, the size dependency graph G3 corresponds to the base case (C1) of this predicate, and G4 and G5 correspond to its recursive clause (C2). Let $N1$ be the transformation node $N1 = ((qsort/2, \emptyset, \{2\}), \{G3, G5\})$. Let $N2$ be the transformation node from Example 3.1. Then, the pair $((qsort/2, \emptyset, \{2\}), \{N1, N2\})$ is an irreducible transformation, with entry point the node $N1$. This irreducible transformation is represented in Figure 3. The pair $((append/3, \{2\}, \{3\}), \{N2\})$ is also an irreducible transformation. □

A note on the generation and nature of transformation nodes: this generation is performed through a mode analysis to determine data flow patterns [2, 13, 1] and an argument size analysis [3]. It is important to note that this combined analysis can in some cases infer intra-literal size relations between arguments of a predicate. This information can be used to generate transformation nodes which can be part of a transformation, but which need to traverse less data because a size computation can be performed directly in one operation, rather than by counting during the execution of the predicate. For example, suppose that the analysis infers the intra-literal size relation $size_3 = size_1 + size_2$ for append/3 (which states that the length of its third argument is the sum of the lengths of its

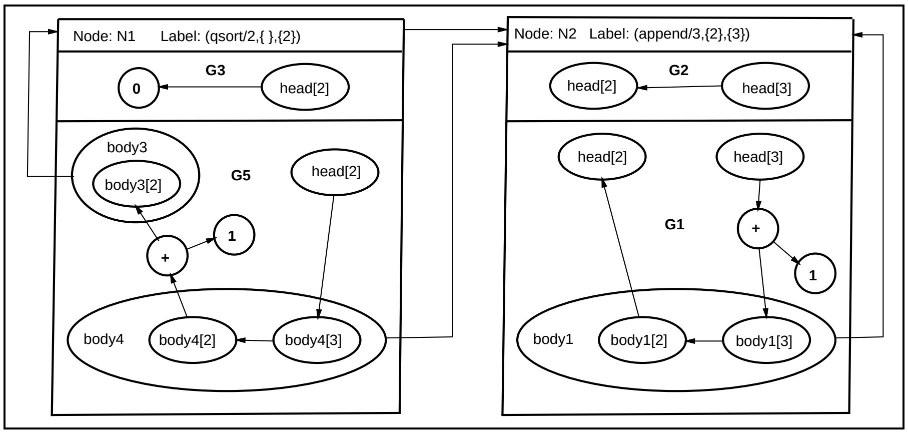

Figure 3: An irreducible transformation.

two first arguments), and the intra-literal size relation $size_2 = size_1$ for predicate `qsort/2`. Consider the clause `C2` in Example 5.1. Using $size_3 = size_1 + size_2$ for `append/3` we have that $|\mathtt{L2}|_{\text{list\_length}} = |\mathtt{Ls2}|_{\text{list\_length}} + |[\mathtt{First}|\mathtt{Lg2}]|_{\text{list\_length}}$ holds for every substitution that makes all the terms appearing in it ground, and also $|\mathtt{L2}|_{\text{list\_length}} = |\mathtt{Ls2}|_{\text{list\_length}} + |\mathtt{Lg2}|_{\text{list\_length}} + 1$ holds. Thus we can infer the following inter-literal size relation $head[2] = body_2[2] + body_3[2] + 1$ which doesn't imply any transformation of predicate `append/3` but of the predicate `qsort/2`. Moreover, using $size_2 = size_1$ for `qsort/2` we have that $|\mathtt{Ls2}|_{\text{list\_length}} = |\mathtt{Ls}|_{\text{list\_length}}$ and $|\mathtt{Lg2}|_{\text{list\_length}} = |\mathtt{Lg}|_{\text{list\_length}}$ also holds. Thus, we can infer another inter-literal size relation $head[2] = body_1[3] + body_1[4] + 1$ (which implies the transformation of predicate `partition/4`)).

Theorem 2 *If there is a transformation T for a size computation specification X then there is an irreducible transformation T' for X.*$\Box$

Theorem 2 implies that we only need to find irreducible transformations to determine whether a procedure is transformable to compute sizes. Obviously, irreducible transformations will result in transformed procedures with potentially less overhead at run-time than the transformations they have been obtained from, but now the problem is to decide which irreducible transformation will have less overhead, or, in other words, which of them will be optimal. The problem of finding such optimal irreducible transformations lies in the fact that we need to use two parameters (number of transformation nodes and number of arguments needed) in the comparison and some transformations may be incomparable, in the sense that one is smaller than the other one on one criteria but the converse is true on the other criteria. In practice we can always assign costs or weights to both argument passing steps and arithmetic operations so that for each transformation we can obtain a function which gives its cost or overhead as a function of the input data sizes. In this case we can compare the cost of irreducible transformations and decide which of them is optimal. In the same way, we can compare the cost of irreducible transformations with the cost of performing the standard size computation, i.e. the one using predefined predicates such as `length/2`, in order to see how convenient performing the transformation to compute sizes is.

Predicate: `find_trans(SCS, S, Trans)`

Input: a size computation specification `SCS` and the information `S` about size relations between arguments in the different clauses of a program for the predicate in `SCS`, derived through size analysis.

Output: an irreducible transformation `Trans` for `SCS`.

Definition:
```
find_trans(SCS, S, Trans) ←
    generate_label(SCS, L), search([L], S, nil, T), Trans=(L, T).
```

Predicate: `generate_label(SCS, L)`

Description: generates a label `L` for `SCS`. Fails when all possible labels have been generated via backtracking.

Predicate: `search(LabelList, SizeRel, InTrans, OutTrans)`

Definition:
```
search(nil, SizeRel, Trans, Trans).
search([Label|LabList], SizeRel, InTrans, OutTrans) ←
    generate_node(Label,SizeRel,[Label|LabList],InTrans,Node,LL),
    append(LL, LabList, NewLabList), Trans = [Node|InTrans],
    search(NewLabList, SizeRel, Trans, OutTrans).
```

Predicate: `generate_node(Label, SizeRel, LabList, InTrans, Node, LL)`

Description: Generates a transformation node `Node` with label `Label`, using the information about size relations `SizeRel` in such a way that the following condition is met: Let S_t be the set of labels of the transformation nodes in the current transformation `InTrans`. Let S_l be the set of labels in `LabList`. Let S_n be the set of labels associated with literal nodes in `Node`. Then, there are no two labels l_1 and l_2, $l_1 \in S_n$ and $l_2 \in (S_t \cup S_l \cup S_n)$, such that $l_2 <_l l_1$.

If it is not possible, or all possible transformation nodes have been generated previously via backtracking, then it fails. Otherwise, it creates a list `LL` containing the labels in the set $S_n - (S_t \cup S_l)$ and succeeds. We omit the detailed description of the generation of `Node` for the sake of brevity.

Figure 4: A top-down algorithm for finding irreducible transformations.

6 Searching for Irreducible Transformations

Since the number of transformation nodes for a given size computation specification is finite, a possible algorithm to find transformations may be to simply generate all possible sets of transformation nodes and test which of them are irreducible transformations. Note that the number of transformation nodes is in any case restricted by the number of size relations that can be inferred by size analysis [3] (in fact, if the algorithm does not find any transformation it does not mean that a transformation does not exist, but rather that it is impossible to find a transformation with the inferred information by size analysis). However, some other more efficient approaches are possible.

In Figure 4 we propose a simple, goal directed algorithm (for which we will later propose some optimizations) which performs a top-down search starting from a given size computation specification (a bottom-up algorithm is also possible). The search space is described by the `find_trans/3` predicate. Note that the irreducible transformations generated still have to be checked in order to determine which of them has the least overhead in the size computation process.

Example 6.1 Consider the predicate `qsort/2` as defined in Example 5.1, and suppose we want to transform it to compute the length of its second argument, that is, we want to find a transformation for the size computation specification $(qsort/2, \{2\})$. We assume a depth-first search (as obtained when the `find_trans/3` predicate is executed in Prolog).

1. The search starts by calling `find_trans(SCS, S, Trans)`, where `SCS` = $(qsort/2, \{2\})$ and `S` is the information about size relations for the predicates in the `quick-sort` program (i.e. `qsort`/2, `partition`/4, and `append`/3).
2. Suppose that `generate_label(SCS, L)` generates the label `L` $= (qsort/2, \emptyset, \{2\})$.
3. Then `search([L], S, nil, T)` is called. Suppose that `generate_node(L, S, [L], nil, Node, LL)` succeeds generating the transformation node `Node` = `N1`, where `N1` = $((qsort/2, \emptyset, \{2\}), \{G3, G4\})$, where $G3$ and $G4$ are the size dependency graphs in Figure 2, and making `LL` = `[L1]`, where `L1`= $(append/3, \emptyset, \{3\})$.
4. A recursive call `search([L1], S, [N1], OutTrans)` is made. This call fails because of the failure of `generate_node(L1, S, [L1], [N1], Node2, LL2)`. Thus, backtracking occurs and `generate_node(L, S, [L], nil, Node, LL)` is retried. Suppose that this call succeeds generating the transformation node `Node` = `N2`, where `N2` = $((qsort/2, \emptyset, \{2\}), \{G3, G5\})$, and $G3$ and $G5$ are the size dependency graphs in Figure 2, and making `LL` = `[L2]`, where `L2`= $(append/3, \{2\}, \{3\})$.
5. A recursive call `search([L2], S, [N2], OutTrans)` is made. Suppose that `generate_node(L2, S, [L2], [N2], Node3, LL3)` succeeds generating the node `Node3` = `N3`, where `N3` = $((append/3, \{2\}, \{3\}), \{$`G1`, `G2`$\})$, where `G1` and `G2` are the size dependency graphs in Figure 1, and making `LL3` = `nil`.
6. Finally a recursive call `search(nil, nil, [N3, N2], OutTrans)` is made. This call succeeds making `OutTrans` = `[N3, N2]`. Thus `Trans` = `(L, [N3, N2])`. □

The efficiency of the previous top-down algorithm can be improved if certain information is used during the generation of transformation nodes performed by `generate_node`/3. In particular, knowledge regarding which of the labels associated with literal nodes in the generated transformation node are likely to make the `generate_node`/3 predicate fail further on while trying to find transformation nodes for such labels. This can prune the search space considerably. It is sometimes possible to detect such labels by examining facts in the program. For example, it is possible to detect that `generate_node`/3 will not find any transformation node for $(append/3, \emptyset, \{3\})$, since, examining the fact which appears in the definition of `append`/3, we can infer that at least it is necessary to supply the size of the second argument of `append`/3 at the call. Thus, no transformation node will be generated having the label $(append/3, \emptyset, \{3\})$ associated with some literal node. We have built a prototype implementation in Prolog along these lines which makes use of the built-in search capabilities of Prolog to perform such a top-down search.

It should be noted that our transformation algorithm can be classified as a "rules + strategies" approach – see [14] and its references– and thus, can be described in terms of applying certain folding and unfolding rules in a particular order. In fact, what our algorithm expresses is a particular "strategy" tailored to finding optimal transformations, in the sense that, if several possible transformations are suitable, it constructs those which have the least runtime overhead, based on the criteria of choosing those which traverse less data and perform less arithmetic operations. This is useful for implementation reasons since it avoids having to implement a full partial evaluator which would be an overkill for the task in hand.

In some simple cases similar transformations to the ones we propose can be obtained by adding to the original program some code that would perform the size computation in a naive way, and then applying a general purpose transformation strategy (e.g. partially evaluating a "length/2" predicate into a previous recursive

bench	$\mathbf{T_{wsc}}$	$\mathbf{T_{st}}$	$\mathbf{T_{pt}}$	$\mathbf{T_{st} - T_{wsc}}$	$\mathbf{T_{pt} - T_{wsc}}$	gain
c/2	202.90	405.69	277.99	202.79	75.09	63.0 %
qsort/2	1218.00	1495.00	1343.90	277.00	125.90	55.3 %
q/2	52.59	90.20	61.69	37.61	9.10	76.7 %
deriv/2	119.00	3349.00	239.00	3110.00	120.00	92.9 %

Table 1: Execution times (ms) for benchmarks.

loop). However, the need for our algorithm comes from the fact that the general purpose strategies used in program transformation systems are less powerful *in this particular application* than our algorithm, in the sense that a general strategy would not ensure obtaining transformations for some cases that our algorithm does, and, also, it would not ensure the optimality of the transformations if they are found. Note, for example, that there are certain transformations which are based on detecting that some term sizes need to be known and used as a starting point for other size computations. This can only be done by reasoning at the "strategy" level.

7 Experimental Results and Advantages

We have run a series of experiments using SICStus PROLOG running on a SUN IPC workstation to measure the gain obtained with our predicate transformation technique with respect to what we will call the "standard approach" to computing term sizes, that is, by introducing new calls to predicates that explicitly compute them. An example is by using the Prolog `length/2` built-in to compute lengths of lists. Theoretically this gain can be up to 100%. To measure this in practice we have chosen a few benchmarks which we feel represent either worst or typical cases and which we argue allow getting some feeling for the performance gain which can be obtained in practice. Table 1 shows execution times for the experiments performed with these benchmarks. $\mathbf{T_{wsc}}$ is the execution time without size computation. $\mathbf{T_{st}}$ is the execution time with size computation using the standard approach. $\mathbf{T_{pt}}$ is the execution time of the predicate transformation approach. $\mathbf{T_{st} - T_{wsc}}$ and $\mathbf{T_{pt} - T_{wsc}}$ are then the overheads due to size computation with the standard and predicate transformation approach, respectively. The last column shows the gain achieved by the predicate transformation approach with respect to the standard one. This gain is computed according to the following expression: $gain = \frac{(\mathbf{T_{st} - T_{wsc}}) - (\mathbf{T_2 - T_{wsc}})}{\mathbf{T_{st} - T_{wsc}}} 100$ For brevity only a brief description of the benchmarks is provided. A more complete description (including the program text) can be found in [5]. The first benchmark that we have chosen contains only the predicate `c/2` followed by a call to `length/2`. `c/2` performs the standard, simplest possible form of list traversal, performing no work during the iterations. Thus, the transformation approach will incur in maximum overhead.

The second benchmark is the predicate `qsort/2`, in which the lengths of the two output lists of `partition/4` are computed. This size computation is useful when transforming the predicate `qsort/2` in order to perform granularity control.

The third benchmark is the predicate `q/2` defined as follows:

```
q([], []).
q([X|Y], [X,X|Y1]) :- X > 7, !, q(Y, Y1).
q([X|Y], [X,X,X|Y1]) :- X =< 7, q(Y, Y1).
```

Execution times have been measured for different lengths of the input list for these three benchmarks, and the observed gain is approximately constant in each case.

Finally, the fourth benchmark is the predicate deriv/2, also performing size computation. Note that in this case the size measure is not list length, but rather term size (we do not include the corresponding transformation for the sake of brevity).

The observed gain arise from two factors: avoiding additional term traversal and performing less arithmetic operations. In both deriv/2 and q/2 the "standard approach" has to traverse more data and thus the number of arithmetic operations is greater than in the predicate transformation approach.

Note that another advantage of our approach is that it can take profit of previous size computations so that no recomputation is performed.

On the other hand, there are also certain cases in which the predicate transformation approach can be more expensive than the standard one. Such cases may appear in connection with backtracking – if there is frequent failure and backtracking within a predicate which has been transformed to perform term size computation it may be better to compute term sizes once and for all using the standard approach upon success. Also, one can construct predicate transformations which perform redundant size computations.

8 Applying the Technique in Granularity Control

As mentioned in the introduction, the approach we propose to the problem of granularity control, and in which the size calculation proposed is instrumental, is by computing cost functions and performing program transformations at compile-time based on such functions, so that the transformed program automatically controls granularity at run-time [3, 11]. The idea is to perform a transformation of the program in such a way that the cost computations and spawning decisions are encoded in the program itself, and in the most efficient way possible. The actual computations and decisions are postponed until run-time when the parameters missing at compile-time, such as data sizes or processor load, are available. In particular, the transformed programs will perform the following tasks: compute input data sizes; use those sizes to evaluate the cost functions; estimate the spawning and scheduling overheads; decide whether to schedule tasks in parallel or sequentially; decide whether granularity control should be continued or not, etc. This is illustrated in the following example, which presents actual output obtained from an implementation of the proposed techniques.

Example 8.1 Consider a parallel version of the definition of the qsort/2 predicate given in Example 5.1:

```
qsort([],[]).
qsort([First|L1],L2) :- partition(First,L1,Ls,Lg),
                        qsort(Ls,Ls2) & qsort(Lg,Lg2),
                        append(Ls2,[First|Lg2],L2).
```

in which qsort(Ls,Ls2) and qsort(Lg,Lg2) are executed in parallel, as expressed by the & symbol [4]. The cost analysis performed at compile-time would provide a function that gives an approximation on the cost of predicate qsort/2 in terms of the size of its first argument, assuming that this argument is ground at procedure invocation. Then a predicate transformation can be done automatically for predicate qsort/2 in order to perform granularity control. As a result of this transformation the following code is obtained:

```
% Version of qsort/2 that performs granularity control.
g_qsort([],[],_).
```

```
g_qsort([First|L1],L2,Size1) :-
      % compute upper-bound of execution time.
      qsorttime(Size1,Time),
      Time < 10 ->
         (partition(First,L1,Ls,Lg),
          s_qsort(Ls,Ls2),s_qsort(Lg,Lg2));
         (trpartition(First,L1,Ls,Lg,0,SizeLs,0,SizeLg),
          g_qsort(Ls,Ls2,SizeLs)&g_qsort(Lg,Lg2,SizeLg)),
      append(Ls2,[First|Lg2],L2).
% Sequential version for qsort/2.
s_qsort([],[]).
s_qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        s_qsort(Ls,Ls2),s_qsort(Lg,Lg2),
        append(Ls2,nFirst|Lg2],L2).

trpartition(F,[],[],[],S1,S1,S2,S2).
trpartition(F,[X|Y],Y1,[X|Y2],ISize1,OSize1,ISize2,OSize2):-
        X > F, !, ISize3 is ISize2 + 1,
        trpartition(F,Y,Y1,Y2,ISize1,OSize1,ISize3,OSize2).
trpartition(F,[X|Y],[X|Y1],Y2,ISize1,OSize1,ISize2,OSize2) :-
        ISize3 is ISize1 + 1,
        trpartition(F,Y,Y1,Y2,ISize3,OSize1,ISize2,OSize2).
```

where the literal `qsorttime(Size1,Time)` computes an estimation of the cost of executing the clause body sequentially, evaluating the function inferred through analysis at compile-time. We have omitted it for the sake of conciseness. The constant `10` represents some experimentally determined threshold which is directly related to the cost of creating a parallel task (note that this could also be a function). The literal `trpartition(First,L1,Ls,Lg,0,SizeLs,0,SizeLg)` is the transformed version of `partition(First,L1,Ls,Lg)`, which is obtained automatically and computes the sizes of its third and fourth arguments (`SizeLs` and `SizeLg` represent the sizes of `Ls` and `Lg` respectively). □

Note that in the cases where term sizes are compared directly with a threshold it is not necessary that the transformed predicates which compute those sizes traverse all the terms involved, but rather only to the point at which the threshold is reached. Thus it is possible to perform more subtle transformations so that when the computed size is greater or equal than the threshold a flag is activated and the size computation is stopped (by executing predicate versions which do not perform any size computation).

We have incorporated the techniques proposed in a prototype granularity control system that we are developing in the context of the &-Prolog system [4]. We have also performed some experiments–which are described in [11]–for several degrees of optimization of the granularity control process. These experiments show that the most significant improvement comes from the incorporation of the dynamic term size computation technique that we have proposed. While the technique needs to be tested on a much larger set of benchmarks to draw firm conclusions, we argue that the results obtained so far are encouraging.

References

[1] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[2] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[3] 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.

[4] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[5] M. Hermenegildo and P. López-García. Dynamic Term Size Computation in Logic Programs. Technical Report CLIP 15/94, Computer Science Faculty, Technical University of Madrid, November 1994.

[6] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[7] 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.

[8] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

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

[10] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[11] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[12] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[13] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[14] A. Pettorossi and M. Proietti. Transformations of Logic Programs: Foundations and Techniques. *Journal of Logic Programming, Special Issue: Ten Years of Logic Programming*, 19/20, May/July 1994.

[15] 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.

[16] J.D. Ullman and A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal ACM*, 35(2):345–373, 1988.

[17] K. Verschaetse and D. De Schreye. Derivation of Linear Size Relations by Abstract Interpretation. In *Fourth International Symposium PLILP'92, Programming Language Implementation and Logic Programming*, pages 296–310, Leuven, Belgium, August 1992. Springer Verlag.

[18] 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.