PhD Thesis

Non-failure Analysis and Granularity Control in Parallel Execution of Logic Programs

presented at the
Facultad de Informática
Universidad Politécnica de Madrid
in partial fulfillment of the degree of
Doctor en Informática

Author: Pedro López García
Master in Computer Science
Universidad Politécnica de Madrid – UPM

Advisor: Manuel de Hermenegildo y Salinas

Madrid, June, 2000

To my parents

Acknowledgements

It is very difficult to express in words great feelings, as the gratitude a feel with people that helped me to make this work possible. To begin with, I'm specially grateful to Manuel Hermenegildo (the advisor and person), without his help, this thesis would not have been possible. He gave me all the support and resources I needed. I thank him for giving me its time, effort, advices, friendship, and above all, for opening me the doors to this fascinating (although sometimes hard) world of research. He has been contributing for many years, and in different ways, as for example by creating the CLIP lab (Computational Logic, Implementation, and Parallelism Lab), to make it possible high quality research in Spain, and has also importantly contributed to spread (Constraint) Logic Programming in Spain and also over the world.

I also thanks my other colleagues at the CLIP lab: Francisco Bueno, Daniel Cabeza, Manuel Carro, María José García de la Banda, and Germán Puebla, for their friendship and help whenever I needed it, their always useful comments, and because of them, I can say that I have a magnificent and friendly work environment. Special mention deserves Francisco Bueno, for its invaluable work on the integration of the tools that we have implemented, and the type analyzers into the ciaopp system, for reading several versions of this dissertation, and contributing to improve its presentation with many advices. Also, Manuel Carro implemented built-in predicates used in this work, and together with Daniel Cabeza collabo-

rated in implementation of the Ciao Prolog compiler.

I would also like to express my gratitude to Saumya Debray and Nai-Wei Lin for implementing the CASLOG system, having it available, having collaborated in many of the works of this thesis and for being coauthors of most of the publications resulting from it.

I'm also grateful to James Lipton, Juan José Moreno, Mario Rodríguez, José Jaime Ruz and Fernando Sáenz for their useful comments.

Last, but not least, thanks also to my parents, my family in general, and all people which in some way, have helped and encouraged me to finish this work.

Abstract

Logic Programming Languages offer an excellent framework for the application of automatic parallelization techniques. On the other hand, there are theoretical results that ensure when parallel(ized) programs are correct (i.e. obtain the same results as their corresponding sequential ones), and when execution of parallel(ized) programs do not take longer than that of the sequential ones. However, such results do not take into account a number of overheads which appear in practice, such as process creation and scheduling, which can induce a slow-down, or, at least, limit speedup, if they are not controlled in some way. In this dissertation, we have developed (an integrated in an advanced system for program analysis, debugging and optimization) a complete automatic granularity control system for logic programs whereby the granularity of parallel tasks, i.e. the work needed for their complete execution, is efficiently estimated and used to limit parallelism so that the effect of the mentioned overheads is controlled. The system is based on a program analysis and transformation scheme, where as much work is done at compile time as possible in order to avoid the introduction of runtime overheads. For this purpose we have developed some program transformation techniques, so that transformed programs perform an efficient granularity control at runtime, and also have developed some program analysis techniques able to infer the information needed for the program transformation phase, such as (lower bounds on) cost of procedures, which calls will not fail (non-failure analysis), etc.

The run-time overhead associated with the approach is usually quite small. Moreover, a static analysis of the overhead associated with the granularity control process is performed in order to decide its convenience. The performance improvements resulting from the incorporation of grain size control are shown to be quite good, specially for systems with medium to large parallel execution overheads.

The non-failure analysis that we have developed can detect procedures and goals that can be guaranteed not to fail (i.e., to produce at least one solution or not terminate), given mode and (upper approximation) type information. The technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a set of variables (i.e. for any element that belongs to the type, at least one test will succeed). We show that the problem of determining a covering is undecidable in general, and give decidability and complexity results for the Herbrand and linear arithmetic constraint systems. We give sound algorithms for determining covering that are precise and efficient in practice. Based on this information, we show how to identify goals and procedures that can be guaranteed to not fail at runtime.

Non-failure information is useful not only for estimating lower bounds on the computational costs of goals (used for granularity control), but also, is useful for many other applications, such as: avoiding speculative parallelism, programming error detection, and program transformation.

Contents

Chapter 1

Introduction

Logic Programming has traditionally been used for implementing typical Artificial Intelligence (AI) applications, and nowadays, it (and its extensions, such as Constraint Logic Programming) has become a very powerful tool in the AI field [Kow74, vEK76, Kow80, Col87, RN95]. This is not surprising, since Logic Programming has some features which make it appropriate for the implementation of typical applications in this field, such as knowledge based systems, knowledge bases, etc. Such applications are in general complex, involving complex search processes and with a strong symbolic component. Moreover, logic constitute a knowledge representation paradigm itself which has the important feature of separating knowledge representation from the reasoning processes on this knowledge. In terms of logic programming, we can say that logic is separated from control. This feature (shared with declarative languages in general) allows to tackle complex problems in a successful way.

On the other hand, regarding efficiency, most of the interesting applications of AI require a great deal of computing capabilities, since in general they are large, complex, with costly search and inference processes and with a mixture of symbolic, numeric and data base processing, that often approximate or exceed the limits of computing capabilities of current systems. The use of computers available nowadays in the marketplace with high computing and storage capabilities (such as mainframes or supercomputers), is an expensive solution to this efficiency problem. However, a better solution can be to exploit parallelism in programs, in order to achieve low-cost parallel architectures with a good performance (i.e. good performance/prize ratio).

The parallelism existing in a logic program can basically be of two types[Con83]: *and-parallelism* and *or-parallelism*. *Or-parallelism* consists on the simultaneous execution, performed in different processors, of different paths of the resolution tree. That is, is the parallel execution of different clauses whose head unify with a particular goal. *Or-parallelism* is typical in non-deterministic problems, i.e., search problems. Conceptually, this kind of parallelism is very simple, an thus, its implementation is practically solved [LH85, War87b, Lus88, Kal87, Sze89, War87a, AK90, CH83, Hau90].

And-parallelism[DeG84, Kal87, BSY88, CDD85, Her86, Lin88, WR87, WW88, BR86, Con83, Fag87, Hua85, LK88, PK88, Kow79] consists on the parallel execution of different body literals of a clause. In contrast with *or-parallelism*, it arises in both, deterministic and non-deterministic problems, which broaden its application field. However, its implementation is a particular complex tasks due to the possible dependencies/interactions between candidate goals for parallel execution.

It is possible to exploit the aforementioned types of parallelism in logic programs [Kal87]. However, parallelization in general is a very complex problem to be performed manually by programmers. Thus, it is desirable to perform it automatically. A lot of theoretical work has been done in automatic parallelization, and most of the problems to achieve it have been solved (see for example [CC94, HR95]):

- correctness: the parallel execution obtains the same results as the sequential, and
- "theoretic" efficiency: the amount of work performed is not greater or, at least, there is no slow-down.

However, results on "theoretic" efficiency assume an idealized execution environment in which a number of practical overheads are ignored, such as those associated with task creation, possible migration of tasks to remote processors, the associated communication overheads, etc. Due to these overheads, and if the granularity of parallel tasks, i.e. the work necessary for their complete execution, is "too small", it may happen that the costs are larger than the benefits in their parallel execution. Of course, the concept of "small granularity" is relative: it depends on the concrete system where parallel programs are running. We then have a problem that we have to face: devise a method whereby the granularity of parallel goals and their number can be controlled. Thus, 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.

The benefits from controlling parallel task size will obviously be greater for systems with greater parallel execution overheads. In fact, in many architectures (e.g. distributed memory multiprocessors, workstation "farms", etc.) such overheads can be very significant and, in them, automatic parallelization cannot in general be done realistically without granularity control. In some other architectures where the overheads for spawning goals in parallel are small (e.g. in small shared memory multiprocessors) granularity control is not essential but it can also achieve important improvements in speedup.

1.1 State of the Art

Granularity control has been studied in the context of traditional programming [KL88, MG89], functional programming [Hue93, HLA94], and also logic programming [Kap88, DLH90, ZTD$^+$92, DL93, LGHD94, LGHD96]. Performing an accurate granularity control at compile-time is difficult since most of the information needed, as for example, input data size, is only known at run-time. An useful strategy can be to do as much work as possible at compile-time, and postpone some final decisions at run-time. This can be achieved by generating at compile-time cost functions which estimate task costs as a function of input data size, which are then evaluated at run-time when such size is known. Then, after comparing costs of parallel and sequential executions, it can be determined which of these types of executions are performed. This scheme was proposed by [DLH90] in the context of logic programs and by [RM90] in the context of functional programs. However, the central topic of the approach proposed in [DLH90] was really the problem of estimating upper bounds to task execution times, leaving as future work the determination of how that information was to be used.

One of the challenges of this dissertation is to fill this gap by illustrating and offering solutions for the many problems involved, for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model. Such problems include on one hand estimating the cost of goals, of the overheads associated with their parallel execution, and of the granularity control technique itself. On the other hand there is also the problem of devising, given that information, efficient compile-time and run-time granularity control techniques. In this work we propose solutions to the many problems previously mentioned.

An alternative approach is to determine only the *relative* cost of goals [ZTD$^+$92], which can be specially useful for optimizing an on-demand run-time scheduler, but may not be as effective in reducing task creation cost. These approaches are in contrast with others, such as that of [Sar89] who bases his algorithm on information obtained via runtime profiling rather than compile-time analysis. [GH85] considers "serial combinators" with reasonable grain sizes, but does not discuss the compile-time analysis necessary to estimate the amount of work that may be done by a call. Serial combinators express fairly low-level control information (e.g. evaluating functions in parallel, waiting for some variables to get a value, etc.) as pseudo-functions in a functional language. They are not intended for direct use by programmers, but are intended to be inserted by an automatically partitioning compiler.

We address the granularity control problem by using the overall approach originally sketched by [DLH90] of computing complexity functions and performing program transformations at compile-time based on such functions, so that the transformed program automatically controls granularity. Taking this approach as a starting point, we have proposed a general granularity control system model able to use information on lower/upper bounds on cost of procedures, and have particularized such model to the case of logic programming and to or/and parallelism [LGHD94, LGHD96].

In the context of logic programming, most of the work on compile-time cost estimation of logic programs has focused on the estimation of upper bounds on costs. However, in many applications, such as parallel implementations on distributed-memory machines, one would prefer to work with lower bounds instead [DLGHL97]. For this reason, we have developed an analysis that obtains functions which estimate lower bounds on the cost of procedures as a function of input data size. This work has been developed in the context of the ESPRIT Project PARFORCE (1.992 – 1.995) and published in [DLGHL97]. Later on, another work like ours has been developed [KSB97], however, instead of determining cost functions, it aims at determining data size thresholds for which it can be ensured that the cost of a given procedure will be greater that a given grain size. Thus, our work is more general and offers a broader application spectrum.

The biggest problem with the inference of lower bounds on the computational cost of logic programs is the possibility of failure. Any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0. Thus, we need a non-failure analysis [DLGH97] if we want non trivial lower bounds. In this dissertation, we have tackled this challenge by developing a non-failure analysis such that given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail [DLGH97]. Our approach to non-failure analysis is inspired in some developed works on regular types [DZ92], equation and disequation over the Herbrand Universe domain solving algorithms [CL89, Kun87, LM87, LMM88, LMM91], and arithmetic constraint solving algorithms [Pug92, PW93], so that we have achieved an algorithm able to deal with a broad number of cases, and whose correctness can be proved based on theoretical works already developed.

There is a related work with non-failure analysis, called cardinality analysis, that infers lower and upper bounds on the number of solutions produced by a call [BCMH94]. However, such analysis is more appropriate for determinism analysis instead of determining which calls will not fail.

On the other hand, a significant shortcoming of the approach to cost estimation presented in [DL93] is its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates are dependent. Since this kind of programs are frequently used in practice, their analysis was another challenge [DLGH95]. For this reason we have proposed an improvement of the analysis of divide-and-conquer predicates, for both cases, lower and upper bound cost analysis [DLGHL97].

Most of the work on granularity control for logic programs has been devoted to and-parallel systems. Thus, we found the need of extending the approach based on program analysis and transformation to the or-parallelism case. This is a problem that we tackled in [LGHD94, LGHD96].

In [AR94, AR97] a distributed parallel system is presented which incorporates a simple granularity control for or–parallelism. However, it is not based on cost analysis, but in the "age" of created choice points.

There are some related works on granularity analysis and control of concurrent logic programs [GT94, GT95, Gal97], which are also based in the approach originally sketched by [DLH90] of computing cost functions and performing program transformations at compile-time based on such functions, so that the transformed program automatically controls granularity. However, such works do not describe a granularity control system in the generality that we do, instead, they are centered on concurrent logic languages and shared memory multiprocessor systems. Consequently, the proposed solutions in these works are quite particular. In contrast, we propose a general model for granularity control, and also particularized it to and/or-logic programs. Thus, although both works are related and have some common points, they solve very different particular problems, or solve the same problems in a different way.

Another difference is that we use program transformation techniques to generate efficient code that performs granularity control at run-time (as for example, program transformations to perform input data size computations), while the other works do not perform any of this kind of optimizations (they are mainly centered in several analysis of concurrent logic programs.)

Moreover, our work, is the only existing one that describe (in the generality that we present it) and implements a complete and totally automatic granularity control system for logic programs. In fact, the first time we describe our complete granularity control system that we propose is in [LGHD94], and the first version of our size computation technique is described in [LGH93].

A further work on granularity control is [SCK98], however, it is based in the use of a metric called "distance" and its objective is to limit the parallelism in a program by keeping track of the the time distance between program points in which parallelizations are performed, limiting this way the number of parallel tasks created in the system.

1.2 Thesis Objectives

This thesis aims at giving answers to the challenges previously mentioned. Thus, one important objective of the thesis is the development of efficient granularity control techniques in order to justify even more its importance as an additional method to be considered in the optimization of parallel execution of logic programs and to extend its scope of applicability to systems for which existing schemas of granularity control do not mean any optimization.

We pretend to completely develop an automatic granularity control system for logic programs and evaluate such a system in different parallel systems. We pursue the objective that the techniques previously mentioned perform an efficient and more accurate granularity control than the existing ones, and also, that the run-time overhead associated to the technique be small, since most of the granularity control will be performed at compile-time, relegating at run-time, only strictly necessary work.

In order to achieve the mentioned objectives, the thesis develops a granularity control system based on the program analysis and transformation schema initially proposed in [DLH90]. In the program transformation phase, we pursue the objective of minimizing the run-time overhead [LGH95], while in program analysis, the objective is inferring all needed information for this program transformation.

It is worth of mentioning that some of the techniques that we have developed for granularity control, have other important applications. In particular, non-failure information is very useful for the avoidance of speculative parallelism, program error detection, and program transformation. For this reason, and also because once that we had begun to work on non-failure analysis, the results were very promising, we devoted very much effort on the development of the non-failure analysis, not only having in mind its application to granularity control, but as an objective by itself. For this reason, the reader may notice that we have devoted more effort to non-failure analysis than to other parts that compose the granularity control system, as for example, the program transformation techniques, or the determination of conditions for deciding between parallel and sequential execution.

We also aims at studying the efficiency and accuracy of the developed techniques, and finally, that these techniques be enough general to being applied to other programming languages.

1.3 Main Contributions

We now enumerate the main contributions of this thesis. Since we have done some parts of the work in collaboration with other researchers, we also mention them. Moreover, we comment on the publications resulting from our work (for brevity we do not mention the many technical reports of the ESPRIT and CICYT projects, nor other technical reports also resulting of our work):

1. We have proposed a general model for a granularity control system which is able to use lower and upper bounds on the cost of procedures, and have particularized such model to the logic programming case in the context

of to and/or-parallelism. The model is based in a program analysis and transformation framework. This work has been done in collaboration with Dr. Saumya Debray from University of Arizona and has been published at the First International Symposium on Parallel Symbolic Computation (PASCO) in 1994 [LGHD94], and also in the Journal of Symbolic Computation in 1966 [LGHD96] (the late is an improved and extended version of the former).

2. We have completely implemented (and integrated in a real program precompilation/optimization system: `ciaopp` [HBPLG99, HBC$^+$99]), an automatic granularity control system for logic programs following the model above-mentioned.

3. In order to develop the complete granularity control system that we have proposed, we have solved two fundamental problems: a) Determining tasks costs and overheads, and b) Controlling parallelism using this information. We have described the solutions to these problems with enough generality so that they could be applied to different systems (besides logic programming systems), and to different execution models.

4. Regarding the problem of determining tasks costs and overheads:

   (a) We have completed the upper bound cost estimation performed by the CASLOG [DL93] system. In order to achieve this, we have integrated the CASLOG system in the `ciaopp` system in such a way that the information on types, modes and data size metrics needed by CASLOG is automatically supplied by `ciaopp` analyzers, allowing this way that the cost analysis be totally automatic.

   (b) In order to achieve the aforementioned, we have adapted and integrated in `ciaopp` (in collaboration with Dr. Francisco Bueno of the

Technical University of Madrid) an implementation of Gallagher's regular types analysis [GdW94]. Moreover, we have added the possibility of using parametric types and of performing simplifications of type definitions given both, as predicates or as a set of type rules (it is also possible to perform a translation of such definitions in both ways).

(c) We have developed an analysis that infers lower bounds on the cost of procedures as functions of input data size. We have implemented this cost analysis and integrated it in the `ciaopp` system. We have performed experiments that show that the analysis is accurate and efficient. This work has also been done in collaboration Dr. Saumya Debray (University of Arizona) and with Dr. Nai-Wei Lin (National Chung Cheng University of Taiwan), and has been described in two publications. The first one in the International Static Analysis Symposium (SAS) in 1994 [DLGHL94], which was also an invited talk given by Dr. Saumya Debray, and where, besides describing our initial work on lower bound cost estimation, we also discuss upper bound cost estimation and different problems related to the estimation of costs in general. The second paper in question has been published at the International Logic Programming Symposium (ILPS) in 1997 [DLGHL97].

(d) We have proposed an improvement of both types of cost analysis (upper and lower bound) in the presence of divide-and-conquer programs. The work has been published in the above-mentioned paper.

(e) We have developed a non-failure analysis able to infer which (calls to) predicates will not fail. We have implemented such analysis and integrated in the `ciaopp` system. We have shown that the analysis is fairly accurate and precise (more accurate than the existing ones). This work has also been done in collaboration Dr. Saumya Debray

(University of Arizona) and has been published at the International Conference on Logic Programming (ICLP) in 1.997 [DLGH97].

5. Regarding the problem of controlling parallelism using cost and overhead information:

   (a) We have proposed (efficient) conditions based on both upper and lower bounds on task granularity to choose between parallel and sequential execution and for a general execution model. This work has been published in the papers mentioned in point 1 [LGHD94, LGHD96].

   (b) We have proposed techniques for transforming programs so that they perform an efficient run-time granularity control, as for example, a technique for dynamic term size computation "on the fly". A preliminary version of this technique has been published at the Second Spanish Conference on Declarative Programming (ProDe) in 1993 [LGH93], and an improved and extended version of this paper has been published at the International Conference on Logic Programming (ICLP) in 1995 [LGH95].

6. We have performed an assessment of the accuracy and efficiency of techniques that we have developed.

1.4 Structure of the Work

The structure of the rest of this work is as follows: in Chapter 2 we describe the granularity control model that we propose. We comment on the many problems that arise (some of them more subtle than they appear at first sight) and provide general solutions to them. Finally, we show experimental results of the granularity control techniques that we have developed.

In Chapter 3 we describe our lower bound cost analysis and its improvement to deal with divide-and-conquer programs that we mentioned in the previous

Section. We also comment on the implementation of such analysis that we have performed and show experimental results.

In Chapter 4 we describe the non-failure analysis that we have developed (needed for the lower bound cost analysis above-mentioned), we comment on its implementation and also show experimental results.

In Chapter 5 we describe in detail one of the techniques for performing an efficient run-time granularity control: the technique for dynamic input data size computation “on the fly”. We also show experimental results of the gain achieved by this optimization.

Finally, main conclusions and directions for future work are given in Chapter 6.

Chapter 2

A Methodology for Granularity Control

As mentioned in the introduction, several types of parallelism can be exploited in logic programs while preserving correctness and efficiency, i.e. ensuring that the parallel execution obtains the same results as the sequential one and the amount of work performed is not greater. However, such results do not take into account a number of overheads which appear in practice, such as process creation and scheduling, which can induce a slow-down, or, at least, limit speedup, if they are not controlled in some way. This chapter describes a methodology whereby the granularity of parallel tasks, i.e. the work available under them, is efficiently estimated and used to limit parallelism so that the effect of such overheads is controlled. The run-time overhead associated with the approach is usually quite small, since as much work is done at compile time as possible. Also, a number of run-time optimizations are proposed. Moreover, a static analysis of the overhead associated with the granularity control process is performed in order to decide its convenience. The performance improvements resulting from the incorporation of grain size control are shown to be quite good, specially for systems with medium to large parallel execution overheads.

We know of no other work which describes a complete granularity control system for logic programs, discusses the many problems that arise (some of them more subtle than they appear at first sight) and provides solutions to them in the generality that we present our work.

We do not discuss in detail the different types of overheads which may appear in a parallel execution when comparing it to a sequential execution, which may include not only execution time-related overheads but also, for example, memory consumption overheads, for conciseness, and because we are more concerned with speedups, we concentrate mainly on time-related overheads. However, we conjecture that a similar treatment to that which we propose can be applied to the analysis and control of memory-related overheads.

2.1 A General Model

We start by discussing the basic issues to be addressed in our general approach to granularity control, in terms of a generic execution model. In the following sections we will particularize to the case of logic programs.

2.1.1 Deriving Sufficient Conditions

We first discuss how conditions for deciding between parallel and sequential execution can be devised. We consider a generic execution model: let $g = g_1, \ldots, g_n$ be a task such that subtasks $g_1, \ldots, g_n$ are candidates for parallel execution, T_s represents the cost (execution time) of the sequential execution of g, and T_i represents the cost of the execution of subtask g_i.

There can be many different ways to execute g in parallel, involving different choices of scheduling, load balancing, etc., each having its own cost (execution time). To simplify the discussion, we will assume that T_p represents in some way all of the possible costs. More concretely, $T_p \leq T_s$ should be understood as "T_s is greater or equal than any possible value for T_p".

In a first approximation, we assume that the points of parallelization of g are fixed. We also assume, for simplicity, and without loss of generality, that no tests — such as, perhaps, "independence" tests [CC94, HR95] — other than those related to granularity control are necessary.

Thus, the purpose of granularity control will be to determine, based on some conditions, whether the g_i's are to be executed in parallel or sequentially. In doing this, the objective is to improve the ratio between the parallel and sequential execution times. An interesting goal is to ensure that $T_p \leq T_s$. In general, this condition cannot be determined before executing g, while granularity control should intuitively be carried out ahead of time. Thus, we are forced to use approximations. At this point one clear alternative is to give up on strictly ensuring that $T_p \leq T_s$ and use some heuristics that have good average case behavior. On the other hand, it is not easy to find such heuristics and, also, it is of obvious practical importance to be able to ensure that parallel execution will not take more time than the sequential one. This suggests an alternative solution: evaluating a simpler condition which nevertheless can be proved to ensure that $T_p \leq T_s$. Such a condition can be based on computing an upper bound for T_p and a lower bound for T_s. Ensuring $T_p \leq T_s$ corresponds to the case where the action taken when the condition does not hold is to run sequentially, i.e. to a philosophy were tasks are executed sequentially unless parallel execution can be shown to be faster. This is useful when "parallelizing a sequential program." This approach is discussed in the following section. The converse case of "sequentializing a parallel program", in which detecting when the opposite condition $T_s \leq T_p$ holds is the objective, is considered in Section 2.1.1.

Parallelizing a Sequential Program

In order to derive a sufficient condition for the inequality $T_p \leq T_s$, we derive upper bounds for its left-hand-side and lower bounds for its right-hand-side, i.e. a sufficient condition for $T_p \leq T_s$ is $T_p^u \leq T_s^l$, where T_p^u denotes an upper bound of T_p and T_s^l a lower bound of T_s. We will use the superscripts l and u to denote lower and upper bounds respectively throughout the discussion.

Assume that there are p free processors in the system at the instant in which task g is about to be executed. Assume also that $p \geq 2$ (if there is only one processor, then execution is performed sequentially) and let m be the lowest integer which is greater or equal than n/p, i.e. the ceiling of $\frac{n}{p}$, denoted $m = \lceil \frac{n}{p} \rceil$. We have that $T_p^u = Spaw^u + C^u$, where $Spaw^u$ is an upper bound on the cost of creating the n parallel subtasks, and C^u an upper bound on the execution of g itself. $Spaw^u$ will be dependent on the particular system in which task g is going to be executed. It can be a constant, or a function of several parameters, such as input data size, number of input arguments, number of tasks, etc. and can be experimentally determined. We now consider how C^u can be computed. Let C_i^u be an upper bound on the cost of subtask g_i, and assume that $C_1^u, \ldots, C_n^u$ are ordered in descending order of cost. Two possible ways of computing C^u are the following: $C^u = \sum_{i=1}^{m} C_i^u$; or $C^u = m\ C_1^u$. Each C_i^u can be considered as the sum of two components: $C_i^u = Sched_i^u + T_i^u$, $Sched_i^u$ denotes the time taken from the point in which the parallel subtask g_i is created until its execution is started by a processor (possibly the same processor that created the subtask), i.e. the cost of task preparation, scheduling, communication overheads, etc.[1] T_i^u denotes the time taken by the execution of g_i disregarding all the overheads mentioned

[1] Note that in some parallel systems, such as &-Prolog [HG91], $Sched_i^u$ can in some cases be zero, since there is no overhead associated with the preparation of a parallel task if it is executed by the same processor as the one which created the task.

before. We assume that the tasks $g_1, \ldots, g_n$ are guaranteed to not fail. We also assume that T^l_s can be computed as follows: $T^l_s = T^l_{s_1} + \cdots + T^l_{s_n}$, where $T^l_{s_i}$ is a lower bound of the cost of the (sequential) execution of subtask g_i.

The following two lemmas summarize the previous discussion:

Lemma 2.1.1 *If* $Spaw^u + \sum_{i=1}^{m} C^u_i < T^l_{s_1} + \cdots + T^l_{s_n}$, *then* $T_p \leq T_s$.
Proof: Trivial.

Lemma 2.1.2 *If* $Spaw^u + m\ C^u_1 < T^l_{s_1} + \cdots + T^l_{s_n}$ *then* $T_p \leq T_s$
Proof: Trivial.

As mentioned in the introduction, bounds on execution costs often need to be evaluated totally or partially at run-time, and thus also the conditions above. It would be desirable to make this evaluation be as efficient as possible. There is clearly a tradeoff between the evaluation cost of such a sufficient condition and its accuracy. A sufficient condition with a simpler evaluation than those in lemmas 2.1.1 and 2.1.2 is given below, based on a series of reasonable further assumptions.

Assume that it is ensured that the tasks $g_1, \ldots, g_n$ will not take longer than they would in a sequential execution, ignoring the time to spawn them and all the associated parallel execution overheads[2] and that $Sched^u_1, \ldots, Sched^u_n$ are ordered in descending order of cost. Let $Thres^u$ be a threshold computed using either one of the following expressions: $Thres^u = Spaw^u + m\ Sched^u_1$; or $Thres^u = Spaw^u + \sum_{i=1}^{m} Sched^u_i$.

Theorem 2.1.3 *If there exist at least* $m+1$ *tasks* $t_1, \ldots, t_{m+1}$ *among* $g_1, \ldots, g_n$, *such that for all* i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts^l_{t_i}$, *where* $Ts^l_{t_i}$ *denotes a lower bound of the cost of the sequential execution of task* t_i, *then* $T_p \leq T_s$.

[2] This can be ensured for certain execution platforms, for example if the tasks are "independent". However in some cases, if the tasks are "dependent", they may take longer than they would in a sequential execution.

Proof: Assume that $T_{s_1}, \ldots, T_{s_n}$ are ordered in descending order of cost, where T_{s_i} denotes the cost of the sequential execution of task g_i. Consider the following condition:

$$T_p^u \leq T_{s_1} + \cdots + T_{s_m} + T_{s_{m+1}} + \cdots + T_{s_n} \tag{2.1}$$

where $T_p^u = Thres^u + T_{s_1} + \cdots + T_{s_m}$. We have that if this condition holds then $T_p \leq T_s$, since its left hand side is an upper bound of T_p. Simplifying condition 2.1 we obtain:

$$Thres^u \leq T_{s_{m+1}} + \cdots + T_{s_n} \tag{2.2}$$

If there are at least $m+1$ tasks $t_1, \ldots, t_{m+1}$ among $g_1, \ldots, g_n$, such that for all i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts_{t_i}^l$, then $Thres^u \leq Ts_{t_i}$ (where Ts_{t_i} denotes the cost of the sequential execution of task t_i), and there is some t_i, $1 \leq i \leq m+1$ which is equal to some g_j, $m+1 \leq j \leq n$ and condition 2.2 holds because $Thres^u \leq T_{s_j}$.

We treat now a slightly more complex case in which we also consider other costs, including the cost of granularity control itself: assume now that the execution of g_i takes T_i time steps, such that $T_i = T_{s_i} + W_i$, where W_i is some "extra" work due to either parallel execution itself (for example the cost of accessing remote references) or granularity control or both of them. Let l $(0 \leq l \leq n)$ be the tasks for which we know that $W_i \neq 0$ (equivalently, $T_i > T_{s_i}$). Assume that $W_1^u, \ldots, W_l^u$ are ordered in descending order of cost, and let $r = min(l, m)$. Then, we can compute a new threshold, $Thres_w^u$, by adding W $(Thres_w^u = Thres^u + W)$ to the previous threshold $(Thres^u)$. W can be computed in two possible ways: $W = \sum_{i=1}^{r} W_i^u$; or $W = r\ W_1^u$.

Theorem 2.1.4 *If there exist at least $m+1$ tasks $t_1, \ldots, t_{m+1}$ among $g_1, \ldots, g_n$, such that for all i, $1 \leq i \leq (m+1)$, $Thres_w^u \leq Ts_{t_i}^l$, where $Ts_{t_i}^l$ denotes a lower bound of the cost of the sequential execution of task t_i, then $T_p \leq T_s$.*

Proof: The proof is similar to that of theorem 2.1.3. Since $Thres^u + W + T_{s_1} + \cdots + T_{s_m}$, is also an upper bound of T_p, we can follow the same argument in this proof replacing condition 2.1 by $Thres^u + W + T_{s_1} + \cdots + T_{s_m} \leq T_{s_1} + \cdots + T_{s_m} + T_{s_{m+1}} + \cdots + T_{s_n}$

Suppose now that we cannot ensure that for all i, $1 \leq i \leq n$, g_i is not going to fail. Assume that g_k is the leftmost task for which non-failure is not ensured, for some $1 \leq k \leq n$. We can modify the previous lemmas (2.1.1 and 2.1.2) and theorems (2.1.3 and 2.1.4) slightly as follows.

Lemmas 2.1.1 and 2.1.2 can be reformulated as:

Lemma 2.1.5 *If* $Spaw^u + \sum_{i=1}^{m} C_i^u < T_{s_1}^l + \cdots + T_{s_k}^l$, *then* $T_p \leq T_s$.
Proof: Trivial.

Lemma 2.1.6 *If* $Spaw^u + m\ C_1^u < T_{s_1}^l + \cdots + T_{s_k}^l$ *then* $T_p \leq T_s$
Proof: Trivial.

The only difference is that we consider $T_{s_1}^l + \cdots + T_{s_k}^l$ on the right hand side of the respective inequation instead of $T_{s_1}^l + \cdots + T_{s_n}^l$.

Theorems 2.1.3 and 2.1.4 can be reformulated by assuming as hypothesis that the tasks which have the m greatest costs are among $g_1, \ldots, g_k$. The proofs are similar.

Theorem 2.1.7 *If there exist at least* $m+1$ *tasks* $t_1, \ldots, t_{m+1}$ *among* $g_1, \ldots, g_k$, *such that for all* i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts_{t_i}^l$, *where* $Ts_{t_i}^l$ *denotes a lower bound of the cost of the sequential execution of task* t_i, *and the tasks with the* m *greatest costs are among* $g_1, \ldots, g_k$, *then* $T_p \leq T_s$.

Theorem 2.1.8 *If there exist at least* $m+1$ *tasks* $t_1, \ldots, t_{m+1}$ *among* $g_1, \ldots, g_k$, *such that for all* i, $1 \leq i \leq (m+1)$, $Thres_w^u \leq Ts_{t_i}^l$, *where* $Ts_{t_i}^l$ *denotes a lower bound of the cost of the sequential execution of task* t_i, *and the tasks with the* m *greatest costs are among* $g_1, \ldots, g_k$, *then* $T_p \leq T_s$.

Sequentializing a Parallel Program

Assume now that we want to detect when $T_s \leq T_p$ holds, because we have a parallel program and want to profit from performing some sequentializations. In this case we can compute T_s^u and T_p^l. Let T_i^l be a lower bound on the execution time of g_i. T_p^l can be determined in several ways:

1. If $n \leq p$ then: $T_p^l = Spaw^l + \max(T_1^l, \ldots, T_n^l)$ else: $T_p^l = Spaw^l + \lceil \frac{n}{p} \rceil \min(T_1^l, \ldots, T_n^l)$.
2. $T_p^l = Spaw^l + \sum_{i=1}^{k} T_i^l$, where $k = \lceil \frac{n}{p} \rceil$ and $T_1^l, \ldots, T_n^l$ are ordered in ascending order.
3. $T_p^l = Spaw^l + \frac{T_{s_1}^l + \cdots + T_{s_n}^l}{p}$

The determination of T_i^l will depend, of course, on the way g_i is going to be executed. If the execution is going to be performed in parallel with no granularity control, with granularity control, or sequentially, we compute $T_{p_i}^l$, $T_{g_i}^l$, or $T_{s_i}^l$ respectively. The determination of $T_{p_i}^l$ and $T_{g_i}^l$ is discussed in Section 2.7.

We can choose the maximum of the different possibilities for computing T_p^l. In general, if there are n different choices $x_1, \ldots, x_n$ for computing T_p^l (T_p^u, respectively) we will choose $T_p^l = \max(x_1, \ldots, x_n)$ ($T_p^u = \min(x_1, \ldots, x_n)$, respectively).

2.1.2 Compile-time vs. Run-time Control

The evaluation of the sufficient conditions proposed in the previous sections can in principle be performed totally at run-time, compile-time or partially at each of them. For example, it might be possible to determine at compile time if the condition expressed in Theorem 2.1.3 will always be true when evaluated at run-time. Let C^l be a lower bound of the cost of each g_i, $1 \leq i \leq n$, then if $Thres^u \leq (n-m)C^l$ the condition of the theorem holds, since $(n-m)C^l$ is a lower bound on $T_{s_{m+1}} + \cdots + T_{s_n}$. Clearly, in this case it is not necessary to perform any granularity control and tasks can always be executed in parallel. The converse case is also possible where tasks can be statically determined to be better executed sequentially. Thus, from the granularity control point of view program parts can be classified as *parallel* (all the performed parallelizations are unconditional), *sequential* (there are no parallel tasks), and *performing granularity control* (tests based on granularity information are performed at run-time in order to decide between parallel or sequential execution). Whether it is done at compile-time or at run-time, in order to perform granularity control two basic issues have to be addressed: how the bounds on the costs and overheads which are the parameters of the sufficient conditions are computed (*cost and overhead analysis*) and how the sufficient conditions are used to control parallelism (*granularity control*). They are the subjects of the following sections. Both of these issues imply in general both compile-time and run-time techniques in our approach.

Task Cost Analysis

Since *task cost* is not in general computable at compile-time, we are forced to resort to approximations and, possibly, to performing some work at run-time. In fact, as pointed out by [DLH90], since the work done by a call to a recursive procedure often depends on the size of its input, such work cannot in general be estimated in any reasonable way at compile time and for such calls some run-time work is necessary. The basic approach used is as follows: given a call p, an expression $\Phi_p(n)$ is computed that a) it is relatively easy to evaluate, and b) it approximates $\texttt{Cost}_p(n)$, where $\texttt{Cost}_p(n)$ 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 then evaluated at run-time, when the size of the input is known, yielding an estimate of the cost of the call. We point out that the evaluation of $\Phi_p(n)$ will be simplified as much as possible by the compiler. In many cases it will be possible to simplify the cost function (or, more precisely, the test to be performed) to the point of being able to statically derive a threshold size for one of the input size arguments. In that case, at runtime, such input size is simply compared against the (precomputed) threshold, and thus the function does not need to be evaluated. This simplification is discussed in Section 2.5.1. If after simplification, the resulting expression is costly to evaluate, the compiler may decide to compute a safe approximation with a smaller evaluation cost. We would also like to point out that the cost of evaluating tests, and, in general, of performing granularity control, is also taken into account, as described in Section 2.6. In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*.

As mentioned in Section 2.1 the approximation of the condition used to decide between parallelization and sequentialization can be based either on some heuristics or on a *safe* approximation (i.e. an upper or lower bound). For the latter approach we were able to show sufficient conditions for parallel execution while preserving efficiency. Because of these results, we will in general require $\Phi_p(n)$ to be not just an approximation, but also a bound on the actual execution cost. Fortunately, as mentioned before, much work has been presented on (time) complexity analysis of programs [Met88, Wad88, Ros89, BH89, Sar89, ZZ89, FSZ91]. The most directly applicable are the methods presented by [DL93] and [DLGHL97] for statically estimating cost functions for predicates in a logic program. The two approaches have much in common but they differ in the way the approximation is done. In the first one upper bounds of task costs are computed, that is $(\forall n)\mathtt{Cost}_p(n) \leq \Phi_p(n)$, while in the second one, to be discussed later in this chapter and also in more detail in chapter 3, the converse approximation is done: $(\forall n)\mathtt{Cost}_p(n) \geq \Phi_p(n)$.

Example 2.1.1 Consider the procedure `q`/2 defined as follows:

```
q([],[]).
q([H|T],[X|Y]):- X is H + 1, q(T,Y).
```

where the first argument is an input argument. Assume that the cost unit is the number of resolution steps. In a first approximation, and for simplicity, we suppose that the cost of a resolution step (i.e., procedure call) is the same as that of the `is/2` builtin. With these assumptions, the cost function of `q/2` is $\texttt{Cost}_q(n) = 2\,n + 1$, where n is the size (length) of the input list (first argument). □

Parallelization Overhead Analysis

Regarding the determination of the overheads that appear together with the costs in the sufficient conditions of Section 2.1.1, as mentioned there, this is a more or less trivial task in systems where such costs can be considered constant. However, it is often the case that such costs have, in addition to a constant component, other components which can be a function of several parameters, such as input data size, number of input arguments, number of tasks, number of active processors in the system, type of processor, etc., in which case some run-time evaluation will be needed. For example, in a distributed system, task spawning cost is often proportional to data size, since in many models a complete closure (a call plus its arguments) is sent to the remote processor. Thus, the evaluation of the overheads also implies in general the generation at compile-time of a cost function, to be evaluated at run-time when parameters (such as data size in our previous example) are known.

Performing Granularity Control

Let us assume that techniques, such as those described in general terms above, for determining task costs and overheads are given. Then, the remainder of the granularity control task is to devise a way to actually compute such costs and then control task creation using such information.

We take again the approach of doing as much of the work as possible at compile-time. We propose performing 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 idea is to postpone the actual computations and decisions 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.

2.2 Cost Analysis in Logic Programming

We now further discuss the cost analysis problem in the context of logic programs. We distinguish between the cases of and-parallelism and or-parallelism.

2.2.1 Cost Analysis for And-parallelism

In (goal level) and-parallelism the units being parallelized are goals. We have developed a lower bound goal cost analysis (which also includes a non-failure analysis) which we briefly sketch — see the work of [DLGHL97] for details. The problem when estimating lower bounds is that in general it is necessary to account for the possibility of failure of head unification, leading a naive analysis to always derive a trivial lower bound of 0. Given (an upper approximation of) mode and type information, the analysis can detect procedures and goals which can be guaranteed not to fail. The technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a variable. Conceptually, we can think of a clause as consisting of a set of primitive tests on the actual parameters of the call, followed by body goals. The tests at the beginning determine whether the clause should be executed or not, and in general may involve pattern matching, arithmetic tests, type tests, etc. A type refers to a set of terms. For any given clause, we refer to the conjunction of the primitive tests that determine whether it will be executed as "the tests of the clause". The disjunction of all the tests of the clauses that define a particular predicate will be referred to as "the test of that predicate." Informally, the test of a predicate covers the type of a variable if binding this variable to any value in the type, the test of the predicate succeeds (the extension of this notion to tuples of variables is straightforward).

An upper-bound cost analysis of goals has been described by [DL93]. It is very similar and simpler than that of lower bounds, since the fact that an upper bound on the actual run-time cost is being computed allows assuming that each literal in the body of the clause succeeds and also that all clauses are executed (independently of whether all solutions are required or not).

2.2.2 Cost analysis for Or-parallelism

The case of or-parallelism is similar to that of and-parallelism except that the units being parallelized are branches of the computation rather than goals. However, the cost analyses of the previous sections can be adapted by simply taking into account the "continuation" of the choice points being considered. As an example, consider a clause $h :- \ \ldots, L, L_1, \ldots, L_n.$. Assume that the predicate of literal L is p, and the definition of predicate p contains "c" "eligible" clauses $\{Cl_1, \ldots, Cl_c\}$, where $Cl_i = h_i :- \ b_i$. In the or-parallel execution of literal L, the "c" choices (each one corresponding to a clause of predicate p) and their continuations (the rest of the L_i, $1 \leq i \leq n$, and the other goals L_{n+1} to L_k

that may appear after them in the resolvent at the time L is leftmost) are executed in parallel. Let $Cost_{cl_i}(x)$ and $Cost_{L_i}(x)$ denote the cost of clause Cl_i and literal L_i respectively, then the cost of the choice corresponding to clause Cl_i, denoted by $Cost_{ch_i}$ can be computed as follows: if we are computing lower bounds we have that $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x) + \sum_{j=1}^{m} Cost^l_{L_j}(x)$, if non-failure is ensured for clause Cl_i and m is the leftmost literal for which non-failure is not ensured; or, alternatively, $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x)$, if non-failure is not ensured for clause Cl_i. On the other hand, when computing upper bounds we have that $Cost^u_{ch_i}(x) = Cost^u_{cl_i}(x) + \sum_{j=1}^{k} Cost^u_{L_j}(x)$.

The determination of L_{n+1} to L_k, the continuations of the clause under consideration, cannot be obtained directly from the call graph in the presence of last call optimization. The problem is that while non-tail-calls in the body of a procedure return to the caller, because of last call optimization, a tail call does not return to its caller, but rather to the nearest ancestor procedure that made a non-tail call. Thus, while for non-tail calls the transfer of control from the caller to the callee and back is evident from the program's call graph, this is not the case for tail calls. To address this problem, given a program we construct a context-free grammar as follows: for each program clause

$$p(\bar{t}) :- \ \mathit{Guard} \mid q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$$

the grammar contains a production

$$p \longrightarrow q_1 \ \mathtt{L}_1 \ q_2 \ \mathtt{L}_2 \ \ldots \ \mathtt{L}_{n-1} \ q_n$$

, where the $\mathtt{L}_i$, which are labels corresponding to procedure continuations, are the terminal symbols of the grammar. We then compute FOLLOW sets for this grammar [ASU86]: for any predicate p, FOLLOW(p) gives the set of possible continuations for p.

2.3 Granularity Control in Logic Programming: the And-Parallelism Case

We use an example in this section to explain the basic program transformation intuitively since a formal presentation would unnecessarily make it more complex.[3]

Example 2.3.1 Consider the predicate `q`/2 defined in Example 2.1.1, the predicate `r`/2 defined as follows:

```
r([],[]).
r([X|RX],[X2|RX1]) :- X1 is X * 2, X2 is X1 + 7, r(RX,RX1).
```

and the parallel goal: `..., q(X,Y) & r(X), ...`, in which literals `q(X,Y)` and `r(Z)` are executed in parallel, as described by the & (parallel conjunction) connective [HG91].

The cost functions of `q`/2 and `r`/2 are $\texttt{Cost}_q(n) = 2\ n + 1$ and $\texttt{Cost}_r(n) = 3\ n + 1$ respectively. Assume a number of processors $p \geq 2$. According to Theorem 2.1.3, the previous goal can safely be transformed into the following one:

```
..., length(X, LX), Cost_q is LX*2+1, Cost_r is LX*3+1,
     (Cost_q > 15, Cost_r > 15 -> q(X,Y) & r(X); q(X,Y), r(X)), ...
```

where a value for the threshold ($Thres^u$) of 15 units of computation is assumed, the variables `Cost_q` and `Cost_r` denote the cost of the (sequential) execution of goal `q(X,Y)` and `r(Z)` respectively, and `LX` denotes the length of the list `X`. □

[3] Although presenting the technique proposed in terms of a source-to-source transformation is convenient for clarity and also a viable implementation technique, the transformation can also obviously be implemented at a lower level in order to reduce the run-time overheads involved even further.

2.4 Granularity Control in Logic Programming: the Or-Parallelism Case

Consider the clause body $\ldots, L, L_1, \ldots, L_n$. in the example in Section 2.2.2. This body can be transformed in order to perform granularity control as follows: $\ldots, (cond \rightarrow L' \ ; \ L), L_1, \ldots, L_n$. Where L' is the parallel version of L, and is created by replacing the predicate name of L (p) by another one, say p', such that p' is the parallel version of p, and is obtained from p by replacing predicate name p with p' in all clauses of p. p' is then declared as "parallel" by means of the appropriate directive. If *cond* holds, then the literal L' (parallel version of L) is executed otherwise L is executed.

A problem with the use of a predicate level parallelism directive is that either all or none of its clauses are executed in parallel. Since there can be differences of costs between clauses, this can lead to worse load-balancing, so a better choice can be the use of some declaration which allows us to specify clusters of clauses such that within each cluster clauses are executed sequentially, and the different clusters are executed in parallel. That way, we can have several parallel versions of a predicate, each of them executed if a particular condition holds. This is illustrated in the following example, where a call to `p` in `...,p, q, r.` and predicate `p` are transformed as follows:

```
..., (cond_1 -> p1 ; cond_2 -> p2; p), q, r.

p:- q1, q2, q3.  p1:- q1, q2, q3 //   p2:- q1, q2, q3 //
p:- r1, r2.      p1:- r1, r2 //       p2:- r1, r2.
p:- s1, s2.      p1:- s1, s2.         p2:- s1, s2.
p.               p1.                  p2.
```

Here, the directive `//` declares three clusters for the predicate `p1`: the first and second ones composed of the first and second clauses respectively, and the third cluster composed of the third and fourth clauses (these two clauses are executed or explored sequentially). Also, for the predicate `p2` we have two clusters: the first one composed of the first clause and the second one composed of the second, third and fourth clauses.

2.5 Reducing Granularity Control Overhead

The transformations proposed inevitably introduce some new overheads in the execution. It would be desirable to reduce this run-time overhead as much as possible. We propose optimizations which include test simplification, improved term size computation, and stopping granularity control, where if it can be determined that a goal will not produce tasks which are candidates for parallel execution, then a version which does not perform granularity control is executed.

In order to discuss the optimizations we need to introduce some terms. We first recall the notion of "size" of a term. Various measures can be used to determine the "size" of an input, e.g., term-size, term-depth, list-length, integer-value, etc. [DL93]. The measure(s) appropriate in a given situation can generally be determined by examining the operations performed in the program. Let $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ be a function that maps ground terms to their 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 $|f(a)|_{\text{list\_length}} = \perp$. We extend the definition of $|\cdot|_m$ to tuples of terms in the obvious way, by defining the function $Siz_m : \mathcal{H}^n \mapsto {\mathcal{N}_\perp}^n$, such that $Siz_m((t_1, \ldots, t_n)) = (|t_1|_m, \ldots, |t_n|_m)$.

Let I and I' denote two tuples of terms, Φ a set of substitutions and θ a substitution. We also define the set of states corresponding to a certain clause point as those states whose leftmost goal corresponds to the literal after that program point. We define the set of substitutions at a clause point in a similar way.

Definition 2.5.1 [*Comp* function] Given a state s_1 corresponding to a clause point p_1, the current substitution θ corresponding to that state, and another clause point p_2, we define $comp(\theta, p_2)$ as the set of substitutions at point p_2 which correspond to states that are in the same derivation as s_1. ■

Definition 2.5.2 [Directly computable sizes] Consider a set Φ of substitutions at a clause point p_1 and another clause point p_2. $Siz_m(I')$ is directly computable at p_2 from $Siz_m(I)$ with respect to Φ if exists a (computable) function ψ such that for all θ, θ', $\theta \in \Phi$, and $\theta' \in comp(\theta, p_2)$, $Siz_m(I\theta)$ is defined and $Siz_m(I'\theta') = \psi(Siz_m(I\theta))$. ■

Definition 2.5.3 [Equivalence of expressions] Two expressions E and E' are equivalent with respect to the set of substitutions Φ if for all $\theta \in \Phi$ $E\theta$ yields the same value as $E'\theta$ when evaluated. ■

2.5.1 Test Simplification

Informally, we can view test simplification as follows: the starting point is an expression which is a function of the size of a set of terms. We try to find an expression which is equivalent to it but which is a function of a smaller set of terms. Also, we apply standard arithmetic simplifications to this expression. Since this new expression will have less variables, simplification will be easier and the corresponding simplified expression will be less costly to compute.

Let us now formally describe the notion of simplification of expressions. Consider the set of substitutions Φ' at clause point p_2, just before execution of goal g.

Assume that we have an expression $E(Siz_m(I'))$ to evaluate at p_2. The objective is to find a program point p_1 and a tuple of terms I such that $Siz_m(I')$ is directly computable at p_2 from $Siz_m(I)$ with respect to Φ with the function ψ, where Φ is the set of substitutions at clause point p_1 and either $p_1 = p_2$ or p_1 precedes p_2 and $E(Siz_m(I'))$ appear after p_1. We have that $E(\psi(Siz_m(I))$ is equivalent to $E(Siz_m(I'))$ with respect to Φ'. Then we can compute an expression E' which is equivalent to $E(\psi(Siz_m(I))$ (by means of simplifications) with respect to Φ' and its evaluation cost is less than that of $E(\psi(Siz_m(I))$. The following example illustrates this kind of optimization.

Example 2.5.1 Consider the goal `..., q(X,Y) & r(X), ...` in Example 2.3.1. In this example $I = I' = (X)$; $Siz(I')$ is directly computable from $Siz(I)$ with respect to Φ with ψ, where ψ is the identity function. $Siz(I\theta)$ is defined for all θ in Φ, since X is bound to a ground list. Thus, we have that for all $\theta \in \Phi$ and for all $\theta' \in comp(\theta, p_2)$, $Siz(I'\theta') = \psi(Siz(I\theta))$. $E(Siz(I)) \equiv max(2\ Siz(X) + 1, 3\ Siz(X) + 1) + 15 \leq 2\ Siz(X) + 1 + 3\ Siz(X) + 1$. Let us now compute E'. We have that for all $\theta \in \Phi$, $max(2\ Siz(X) + 1, 3\ Siz(X) + 1) \equiv 3\ Siz(X) + 1$, so we have $3\ Siz(X) + 1 + 15 \leq 2\ Siz(X) + 1 + 3\ Siz(X) + 1$ which is simplified to $15 \leq 2\ Siz(X) + 1$ and then to $7 \leq Siz(X)$ which is E'. Using this expression we get a more efficient transformed program than in Example 2.3.1:

```
  ..., length(X, LX), (LX > 7 -> q(X, Y) & r(X) ; q(X, Y), r(X)), ...
```

□

In some cases test simplification avoids evaluating cost functions, so that term sizes are compared directly with some threshold. Assume that we have a test of the form $Cost_p(n) > G$ where G is a number and $Cost_p(n)$ is a monotone cost function on one variable for some predicate p. In this case, a value k can be found such that $Cost_p(k) \leq G$ and $Cost_p(k + 1) > G$, so that the expression $Cost_p(n) > G$ can be simplified to $n > k$.

2.5.2 Stopping Granularity Control

An important optimization aimed at reducing the cost of granularity control is based on detecting when an invariant holds recursively on the condition to perform parallelization/sequentialization and executing in those cases a version of the predicate which does not perform granularity control and executes in the appropriate way which corresponds to the invariant.

Example 2.5.2 Consider the predicate `qsort`/2 defined as follows:

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

The following transformation will perform granularity control based on the condition given in Theorem 2.1.3 and the detection of an invariant (tests have already been simplified —we omit details— so that the input data sizes are directly compared with a threshold):

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition(First, L1, Ls, Lg),
  length(Ls, SLs), length(Lg, SLg),
  (SLs > 20 -> (SLg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2);
                            g_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))
            ;  (SLg > 20 -> s_qsort(Ls, Ls2) , g_qsort(Lg, Lg2);
                            s_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))),
  append(Ls2, [First|Lg2], L2).



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

Note that if the input size is less than the threshold (20 units of computation in this case[4]) then a (sequential) version which does not perform granularity control is executed. This is based on the detection of a recursive invariant: in subsequent recursions this goal will not produce tasks with input sizes greater or equal than the threshold, and thus, for all of them, execution should be performed sequentially and obviously no granularity control is needed. [GH91] presented techniques for detecting such invariants. □

2.5.3 Reducing Term Size Computation Overhead

With regard to term size computation, the standard approach is to explicitly traverse terms, using builtins such as `length`/2. However such computation can also be carried out in other ways which can potentially reduce run-time overhead:

1. In the case where input data sizes to the subgoals in the body that are candidates for parallel execution are directly computable from those in the clause head (an example of this is the classical “Fibonacci” benchmark – see Example 2.7.1) such sizes can be computed by evaluating an arithmetic operation. Clause heads can supply their input data size through additional arguments.

2. Otherwise term size computation can be simplified by transforming certain procedures in such a way that they compute term sizes “on the fly” [LGH95].

[4] This threshold is determined experimentally, by taking the average value resulting from several runs.

3. In the cases where term sizes are compared directly with a threshold it is not necessary to traverse all the terms involved, but rather only to the point at which the threshold is reached.

2.6 Taking Into Account the Cost of Granularity Control

As a result of the simplifications proposed in the previous sections three different types of specialized versions of a predicate can be generated: sequential, parallel with no granularity control, and parallel with granularity control. In this section we address the issue of how to select among these versions. We can view this as a reconsideration of the original problem of deciding between parallel and sequential execution, addressed in Section 2.1, but where we add the new issue of deciding whether to perform granularity control or not. Let T_s, T_p, and T_g denote the execution time of the sequential, parallel, and granularity control versions for the predicate corresponding to a given call. The original problem tackled in Section 2.1 can be viewed as determining $min(T_s, T_p, T_g)$. Again, this is not computable ahead of the execution of the goals and we are once more forced to compute an approximation based on heuristics or sufficient conditions. We again take the latter approach, i.e. using sufficient conditions, which we would in principle try to compute for each of the six possible cases involved: $T_g \leq T_s$, $T_p \leq T_s$, $T_p \leq T_g$, $T_s \leq T_g$, $T_s \leq T_p$ and $T_g \leq T_p$. Since we can only approximate these conditions an important issue is the decision taken when none of such conditions can be proved to hold. One solution is to have a pre-determined order relation which is used unless another relation can be proven to be true. This corresponds to the two cases of "sequentializing by default" or "parallelizing by default" studied in Section 2.1, where only one condition was considered. For example, a default ordering might be: $T_g \leq T_s \leq T_p$, which essentially expresses a default assumption that the optimal execution time is achieved when execution is performed in parallel with granularity control unless the contrary is proven. Goals are also executed sequentially unless parallel execution is proven to take less time. If the "no-slowdown" condition is to be enforced, i.e. it is required that the sequential execution time not be exceeded, then, in all pre-determined order relations we must have that $T_s \leq T_g$ and $T_s \leq T_p$.

Note that these pre-determined order relations can be partial. In that case at some point a heuristic has to be applied. The order between two costs T_1 and T_2 can then be determined as follows: If T_1 and T_2 are related in the pre-determined order relation, then compute a sufficient condition to prove the opposite order; otherwise, if some sufficient condition to prove either of the relations $T_1 \leq T_2$ or $T_2 \leq T_1$ holds then we choose the corresponding one; otherwise the order can be determined by means of some heuristics. A good heuristic can be to use the average of the lower and upper bound which are already computed or take the average of the computed costs of the different clauses of a predicate.

2.7 Determining T_p and T_g of a call

The determination of a bound for T_s has already been addressed in the previous sections. There, T_p was simply assumed to be the same as T_s, taking as its approximation the opposite bound to that used for T_s. We now address the issue of determining T_p more precisely and also determining T_g. For conciseness, we present the techniques by means of an example.

Example 2.7.1 Let us consider a transformed version `gfib`/2 of the `fib`/2 predicate which performs run-time granularity control:

```
gfib(0, 0).
gfib(1, 1).
gfib(N, F):- N1 is N - 1, N2 is N - 2,
            (N > 15 -> gfib(N1, F1) & gfib(N2, F2)
                    ; sfib(N1,F1), sfib(N2,F2)),
             F is F1 + F2.


sfib(0, 0).
sfib(1, 1).
sfib(N, F):- N > 1, N1 is N - 1, N2 is N - 2,
             sfib(N1, F1), sfib(N2, F2),
             F is F1+F2.
```

□

2.7.1 Cost of parallel execution without granularity control: T_p

Upper bounds

In general it is difficult to give a non-trivial upper bound on the cost of the parallel execution of a given set of tasks, since it is difficult to predict the number of free processors that will be available to them at execution time. Note that a trivial upper bound can be computed in some cases by assuming that all the potentially parallel goals are created as separate tasks but they are all executed by one processor.

Consider the predicate `fib`/2 defined in Example 2.7.1. Let Is denote the size of the input (first argument) and $T_p(Is)$ the cost of the parallel execution without granularity control of a call to predicate `fib`/2 for an input of size Is. The following difference equation can be set up for the recursive clause of `fib`/2:
$T_p^u(Is) = C_b^u(Is) + Spaw^u(Is) + Sched^u(Is) + T_p^u(Is-1) + T_p^u(Is-2) + C_a^u(Is)$

for $Is > 1$, where $C_b(Is)$ and $C_a(Is)$ represent the costs of the sequential execution of the literals before and after the parallel call respectively, that is, $C_b(Is)$ represents the cost of `N1 is N-1,N2 is N-2` and $C_a(Is)$ the cost of `F is F1+F2`. The solution to this difference equation gives the cost of a call to `fib`/2 for an input of size Is. The following boundary conditions for the equation are obtained from the base cases: $T_p^u(0) = 1$ and $T_p^u(1) = 1$.

Lower bounds

A trivial lower bound — taking non-failure into account, as discussed by [DLGHL97] — can be computed as follows: $T_p^l(Is) = \frac{W_p^l(Is)}{p}$, where W_p^l represents the work performed by the parallel execution with no granularity control of a call to predicate `fib`/2 for an input of size Is, and can be computed by solving the following difference equation: $W_p^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_p^l(Is-1) + W_p^l(Is-2) + C_a^l(Is)$ for $Is > 1$, with the boundary conditions: $W_p^l(0) = 1$ and $W_p^l(1) = 1$.

As an alternative, another value for $T_p^l(Is)$ can be obtained by solving the following difference equation: $T_p^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + T_p^l(Is-1) + C_a^l(Is)$ for $Is > 1$, with the boundary conditions: $T_p^l(0) = 1$ and $T_p^l(1) = 1$. In this case, an infinite number of processors is considered. Since in each "fork" there are two branches, the longest of them ($T_p^u(Is-1)$) is chosen.

2.7.2 Cost of the execution with granularity control: T_g

Upper bounds

The following difference equation can be set up for the recursive clause of `fib`/2: $T_g^u(Is) = C_b^u(Is) + Test^u(Is) + Spaw^u(Is) + Sched^u(Is) + T_g^u(Is-1) + T_g^u(Is-2) + C_a^u(Is)$ for $Is > 15$. We assume that all the potentially parallel goals are created as separate tasks but they are all executed by one processor, as is done in Section 2.7.1.

For a call with $Is = 15$ there is no overhead associated with parallel execution since it is performed sequentially, so that the following boundary conditions are obtained: $T_g^u(15) = Test^u(15) + T_s^u(15)$; and $T_g^u(Is) = T_s^u(15)$ for $Is \leq 15$, where $T_s^u(15)$ denotes an upper bound on the sequential execution time of a call to `fib`/2 with an input of size 15.

Lower bounds

A trivial lower bound (taken non-failure into account) can be computed as follows: $T_g^l(Is) = \frac{W_g^l(Is)}{g}$, where W_g^l represents the work performed by the execution with granularity control of a call to `fib`/2 for an input of size Is, which can be computed by solving the following difference equation: $W_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_g^l(Is-1) + W_g^l(Is-2) + C_a^l(Is)$ for $Is > 15$, with the boundary conditions: $W_g^l(15) = Test^l(15) + T_s^l(15)$, and $W_g^l(Is) = T_s^l(15)$ for $Is \leq 15$, where $T_s^l(15)$ denotes a lower bound on the sequential execution time of a call to `fib`/2 with an input of size 15.

Another value for $T_g^l(Is)$ can be obtained by solving the following difference equation: $T_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + T_g^l(Is-1) + C_a^l(Is)$ for $Is > 15$, with the boundary conditions: $T_g^l(15) = Test^l(15) + T_s^l(15)$, and $T_g^l(Is) = T_s^l(15)$ for $Is \leq 15$.

2.8 Experimental Results

We have developed a prototype of a granularity control system based on the ideas presented. The current prototype has some shortcomings: it only covers the case of (independent, goal level) and-parallelism and the builtin type analyzer is comparatively simple. Despite this, it can achieve effective fully automatic granularity

Table 2.1: Experimental results for benchmarks on &-Prolog.

programs	**seq**	**ngc**	**gc**	**gct**	**gcts**	**gctss**
fib(19)	1.839	0.729	1.169	0.819	0.819	0.549
(**O**=m)		1	-60%	-12%	-12%	+24%
fib(19)	1.839	0.970	1.389	1.009	1.009	0.639
(**O**=5)		1	-43%	-4.0%	-4.0%	+34%
hanoi(13)	6.309	2.509	2.829	2.419	2.399	2.399
(**O**=m)		1	-12.8%	+3.6%	+4.4%	+4.4%
hanoi(13)	6.309	2.690	2.839	2.439	2.419	2.419
(**O**=5)		1	-5.5%	+9.3%	+10.1%	+10.1%
unb_matrix	2.099	1.009	1.339	1.259	0.870	0.870
(**O**=m)		1	-32.71%	-24.78%	+13.78%	+13.78%
unb_matrix	2.099	1.039	1.349	1.269	0.870	0.870
(**O**=5)		1	-29.84%	-22.14%	+16.27%	+16.27%
qsort(1000)	3.670	1.399	1.790	1.759	1.659	1.409
(**O**=m)		1	-28%	-20%	-19%	-0.0%
qsort(1000)	3.670	1.819	2.009	1.939	1.649	1.429
(**O**=5)		1	-11%	-6.6%	+9.3%	+21%

control on three out of the four and-parallel benchmarks (fib, hanoi, and qsort). The results are given in Table 2.1. For the other benchmarks (unb_matrix) and for or-parallelism we have hand-annotated the programs following the algorithms presented and assuming state of the art type inference technology. The results are given in Tables 2.1 and 2.2. We believe that by completing the prototype implementation, and incorporating existing analysis technology, the development of a fully automatic granularity control system is possible, and that our results show that such a system can result in substantial benefit in execution time.

Table 2.2: Experimental results for benchmarks on Muse.

programs	**seq**	**ngc**	**gctss**	**opt**	$\mathbf{e_1}$	$\mathbf{e_2}$
queens(8)	17.019	2.090	1.759	1.702	+15.84 %	+86.83 %
domino(12)	37.049	4.459	4.139	3.705	+7.18 %	+42.43 %
series	22.429	7.360	4.860	2.243	+33.97 %	+48.86 %
farmer	17.929	2.170	2.149	1.793	+0.97 %	+5.57 %

We have first tested the granularity control system with &-Prolog [HG91], a parallel Prolog system, on a Sequent Symmetry multiprocessor using 4 processors. Table 2.1 presents results of granularity control (showing execution times in seconds) for four representative benchmarks and for two levels of task creation and spawning overhead (**O**): minimal (**m**), representing the default overhead found in the &-Prolog shared memory implementation (which is very small – a few microseconds), and an overhead (the &-Prolog system allows adding arbitrary overheads to task creation via a run-time switch) of 5 milliseconds (**5**), which should be representative of a hierarchical shared memory system or of an efficient implementation on a multicomputer with a very fast interconnect. The program `unb_matrix` performs the multiplication of 4×2 and 2×1000 matrices. Results are given for several degrees of optimization of the granularity control process: naive granularity control (`gc`), adding test simplification (`gct`), adding stopping granularity control (`gcts`), and adding "on-the-fly" computation of data size (`gctss`). Results are also given for the sequential execution (`seq`) and the parallel execution without granularity control (`ngc`) for comparison. The obtained speedups have been computed with respect to `ngc`. The importance of the optimizations proposed is underlined by the fact that they result in steadily increasing performance as they are added. Also, except in the case of `qsort` on a very low overhead system, the fully optimized versions show substantial improvements w.r.t. performing no granularity control. Note that the situations studied are on a small shared memory machine and actually imply very little parallel task overhead, i.e. the conditions under which granularity control offers the least advantages. Thus the results can be seen as lower bounds on the potential improvement. Obviously, on systems with higher overheads, such as distributed systems, the benefits can be much larger.

Regarding or-parallelism, Table 2.2 presents results of granularity control (showing execution times in seconds) for some benchmarks on the Muse [AK90] system using 10 workers, and running on a Sequent Symmetry multiprocessor with 10 processors. **queens(8)** computes all the solutions to the 8 queens problem. **domino(12)** computes all the legal sequences of 12 dominoes. **series** computes a series whose expression is a disjunction of series. **farmer** is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC. Results are given for the fully optimized versions which perform granularity control (`gctss`), the sequential execution (`seq`) and the parallel execution without granularity control (`ngc`) for comparison. **opt** is a lower bound on the optimal time, i.e. $\mathbf{opt} = \frac{\mathbf{seq}}{10}$. $\mathbf{e_1} = \frac{\mathbf{ngc-gctss}}{\mathbf{ngc}} \times 100$, and $\mathbf{e_2} = \frac{\mathbf{ngc-gctss}}{\mathbf{ngc-opt}} \times 100$ indicate the percentage of the saved time, with respect to the parallel execution time without granularity control and the ideal parallel execution time respectively, when granularity control is performed. Note that some programs do not exhibit the necessary inherent parallelism to achieve this ideal execution time even if there were no overheads associated with their parallel executions. The reason for introducing these two metrics is that the Muse system showed very good performance in the execution of the selected benchmarks. This is because the Muse scheduler performs an implicit control of parallelism depending on the load of the system. Thus, the potential benefits from applying our granularity control techniques to these benchmarks were more limited. This metric allows us to conclude that our results are in fact quite good, since in general they achieve a significant portion of the potential benefits. Note also that the situations studied are on a small shared memory machine, and, thus, the results, as in the and-parallelism case, can be seen as lower bounds on the potential improvement.

2.9 Chapter Conclusions

We have developed (and integrated in the `ciaopp` system [HBPLG99, HBC$^+$99]) a complete granularity control system for logic programs, which is based on a program analysis and transformation scheme, where as much work is done at compile time as possible in order to avoid the introduction of runtime overheads.

We have discussed the many problems that arise (for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model) and provided solutions to them. We believe that the results are general enough to be of interest to researchers working on other parallel languages. We have also assessed the developed granularity control techniques for and-parallelism and or-parallelism on the &-Prolog and Muse systems respectively, and have obtained what we believe are quite encouraging results.

It appears from the sensitivity of the results that we have observed in our experiments 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 control can usefully be performed automatically by a compiler.

We can conclude that granularity analysis/control is a particularly promising technique because it has the potential of making feasible to automatically exploit low-cost parallel architectures, such as workstations on a (possibly high speed) local area network.

Chapter 3

Lower Bound Cost Analysis for Logic Programs

It is generally recognized that information about the runtime cost of computations can be useful for a variety of applications. For example, it is useful for granularity control, in logic and functional parallel languages [LGHD96, DLH90, ZTD$^+$92, HLA94, RM90, Kap88], and for query optimization in deductive databases [DL90]. In the context of logic programming, the work on cost estimation has generally focused on upper bound cost analyses [DL93]. However, in many cases one would prefer to work with lower bounds instead. As an example, consider a distributed memory implementation of Prolog: suppose that the work involved in spawning a task on a remote processor takes 1000 instructions, and that we infer that a particular procedure call in a program will execute no more than 5000 instructions. This suggests that it *may be* worth executing this call on a remote processor, but provides no assurance that doing so will not actually produce a performance degradation relative to a sequential execution (the call might terminate after executing only a small number of instructions). On the other hand, if we know that a call will execute at least 5000 instructions, we can be assured that spawning a task on a remote processor to execute this call is worthwhile. Thus, while upper bound cost information is better than no information at all, lower bounds may be more useful than upper bounds.

The biggest problem with the inference of lower bounds on the computational cost of logic programs is the possibility of failure. Any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0. For this reason, we had to develop a non-failure able to infer which calls and procedures will not fail. This analysis is described in detail in Chapter 4.

The main contributions of this Chapter are as follows: (*i*) we show how non-failure information can be used to infer non-trivial lower bounds on the computational costs of goals; (*ii*) discuss how to bound the chromatic polynomial of a graph from below, and thereby show how to infer lower bounds on the number of solutions a predicate can generate (this information is useful, for example, for estimating communication costs in distributed-memory implementations); (*iii*) show how information about the number of solutions computed can be used to improve lower bound estimates when all solutions to a goal are required; and (*iv*) show how to obtain improved lower bound estimates for a simple but common class of divide-and-conquer programs.

Our ideas have been implemented and the resulting lower bound cost estimates, given in Section 3.6, can be seen to be quite precise, especially for an automatic analysis tool.

3.1 Lower-Bound Cost Analysis: The One-Solution Case

If only one solution is required of any computation, it suffices to know whether a computation will generate at least one solution, i.e., will not fail. Assuming that this information is available, for example by using the technique mentioned in the previous section, cost analysis for a particular predicate can then proceed as follows:

1. We first determine the relative sizes of variable bindings at different program points in a clause by computing lower bounds on output argument sizes as functions of input argument sizes. This is done by solving (or estimating lower bound solutions to) the resulting difference equations: the approach is very similar to that discussed in [DL93], the only difference being that whereas [DL93] estimated upper bounds on argument sizes using the *max* function across the output sizes of different clauses in a cluster, we use the *min* function across clauses to estimate lower bounds on argument sizes.

2. The (lower bound) computational cost of a clause is then expressed as a function of the input argument size, in terms of the costs of the body literals in that clause.

   Consider a clause $C \equiv$ '$H :- B_1, \ldots, B_m$'. Let n be the r–tuple which represents the sizes of the r input arguments for the head of the clause, and let (lower bounds on) the input argument sizes for the body literals $B_1, \ldots, B_m$ be $\phi_1(n), \ldots, \phi_m(n)$ respectively. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_i. Then, if B_k is the rightmost body literal that is guaranteed to not fail, the following gives a lower bound on the cost $Cost_C(n)$ of the clause C on an input of size n:

$$h(n) + \sum_{i=1}^{k} Cost_{B_i}(\phi_i(n)) \ \leq \ Cost_C(n).$$

If the clause C corresponds to a non-failing predicate, then we take $k = m$.

3. A lower bound on the cost $Cost_p(n)$ of a predicate p on an input of size n is then given by

$$\min\{Cost_C(n) \mid C \text{ is a clause defining } p\} \ \leq \ Cost_p(n).$$

As discussed in [DL93], recursion is handled by expressing the cost of recursive goals symbolically as a function of the input size. From this, we can obtain a set of difference equations that can be solved (or approximated) to obtain a lower bound on the cost of a predicate in terms of the input size.

Given a predicate defined by m clauses $C_1, \ldots, C_m$, we can improve the precision of this analysis by noting that clause C_i will be tried only if clauses $C_1, \ldots, C_{i-1}$ fail to yield a solution. For an input of size n, let $\delta_i(n)$ denote the least amount of work necessary to determine that clauses $C_1, \ldots, C_{i-1}$ will not yield a solution and that C_i must be tried: the function δ_i obviously has to take into account the type and cost of the indexing scheme being used in the underlying implementation. In this case, the lower bound for p can be improved to:

$$\min\{Cost_{C_i}(n) + \delta_i(n) \mid 1 \leq i \leq m\} \ \leq \ Cost_p(n).$$

The pruning operator can also be taken into account, so that clauses which are after the first clause, say C_i, which has a non-failing sequence of literals just before the cut, are ignored, and the lower bound on the cost of the predicate is then the minimum of the costs of the clauses preceding the clause C_i and this clause itself.

3.2 Lower-Bound Cost Analysis: All Solutions

In many applications, it is reasonable to assume that all solutions are required. For example, in a distributed memory implementation of a logic programming system, the cost of sending or receiving a message is likely to be high enough that it makes sense for a remote computation to compute all the solutions to a query and return them in a single message instead of sending a large number of messages, each containing a single solution. For such cases, estimates of the computational cost of a goal can be improved greatly if we have lower bounds on the number of solutions—indeed, as the example of a distributed memory system suggests, in some cases the number of solutions may itself be a reasonable measure of cost.

If we obtain lower bounds on the number of solutions that can be generated by the literals in a clause (this problem is addressed in next section), we can use this information to improve lower bound cost estimates for the case where all solutions to a predicate are required. Consider a clause '$p(\bar{x}) :- B_1, \ldots, B_n$' where B_k is the rightmost literal that is guaranteed to not fail. Let the input argument size for the head of the clause be n, and let (lower bounds on) the input argument sizes for the body literals $B_1, \ldots, B_m$ be $\phi_1(n), \ldots, \phi_m(n)$ respectively. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_i. Now consider a body literal B_j, where $1 \leq j \leq k+1$, i.e., all the predecessors of B_j are guaranteed to not fail. The number of times B_j will be executed is given by the total number of solutions generated by its predecessors, i.e., the literals $B_1, \ldots, B_{j-1}$. Let this number be denoted by N_j : we can estimate N_j using Theorem 3.4.1 (or extensions thereof), e.g., by considering a clause whose body consists of the literals $B_1, \ldots, B_{j-1}$, and where the output variables in the head are given by $vars(B_1, \ldots, B_{j-1}) \cap vars(B_j, \ldots, B_n)$. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_j. Then, a lower bound on the execution cost of the clause to obtain all solutions is given by

$$h(n) + \sum_{i=1}^{k}(N_i \times Cost_{B_i}(\phi_i(n)) \ \leq \ Cost_C(n).$$

3.3 Number of Solutions: The Single-Clause Case

In this section we address the problem of estimating lower bounds on the number of solutions which a predicate can generate.

3.3.1 Simple Conditions for Lower Bound Estimation

It is tempting to try and estimate a lower bound on the number of solutions generated by a clause '$H :- B_1, \ldots, B_n$' from lower bounds on the number of solutions generated by each of the body literals B_i, possibly using techniques analogous to those used in [DL93] for the estimation of upper bounds on the number of solutions. Unfortunately, this does not work. For example, given a clause '$p(X) :- q(X), r(X)$', where X is an output variable, and assuming that q and r generate n_q and n_r bindings, respectively, for X, then $min(n_q, n_r)$ is not a lower bound on the number of solutions the clause can generate. To see this, consider the situation where q can bind X to either a or b, while r can bind X to either b or c: thus, $min(n_q, n_r) = min(2, 2) = 2$, but the number of solutions for the clause is 1.

The following gives a simple sufficient condition for estimating a lower bound on the number of solutions generated by a clause.

Theorem 3.3.1 *Let $x_1, \ldots, x_m$ be distinct unaliased output variables in the head of a clause such that each of the x_i occurs at most once in the body of the clause, and x_i and x_j do not occur in the same body literal for $i \neq j$. If n_i is a lower bound on the number of bindings that can be generated for x_i by the clause body, then $\prod_{i=1}^{m} n_i$ is a lower bound on the number of solutions that can be generated by the clause.*

This result can be generalized in various ways: we do not pursue them here due to space constraints. The utility of this theorem is shown in Example 3.4.1.

3.3.2 Handling Equality and Disequality Constraints

This section presents a simple algorithm for computing a lower bound on the number of solutions for predicates which can be "unfolded" into a conjunction of binary equality and disequality constraints on a set of variables. The constraints are in the form of $X = Y$ or $X \neq Y$ for any two variables X and Y. The types of the variables in a predicate are assumed to be the same and to be given as a finite set of atoms. The problem of computing the number of bindings that satisfy a set of binary equality and disequality constraints on a set of variables with the same type can be transformed into the problem of computing the *chromatic polynomial* of a graph G, denoted by $C(G, k)$, which is a polynomial in k and represents the number of different ways G can be colored by using no more than k colors (see [DL93]).

Unfortunately, the problem of computing the chromatic polynomial of a graph is NP-hard, because the problem of k-colorability of a graph G is equivalent to the problem of deciding whether $C(G, k) > 0$ and the problem of graph k-colorability is NP-complete [Kar72]. Therefore, we will develop an approximation algorithm to compute a lower bound on the chromatic polynomial of a graph. The basic idea is to start with a subgraph that consists of only a single vertex of the graph, then repeatedly build larger and larger subgraphs by adding a vertex at a time into the previous subgraph. When a vertex is added, the edges connecting that vertex to vertices in the previous subgraph are also added. At each iteration, a lower bound on the number of ways of coloring the newly added vertex can be determined by the number of edges accompanied with the vertex. Accordingly, a lower bound on the chromatic polynomial for the corresponding subgraph can be determined using the bound on the polynomial for the previous subgraph and the bound on the number of ways of coloring the newly added vertex.

We now describe the algorithm more formally. The *order* of a graph $G = (V, E)$, denoted by $|G|$, is the number of vertices in V. Let G be a graph of order n. Suppose $\omega = v_1, \ldots, v_n$ is an ordering of V. We define two sequences of subgraphs of G according to ω. The first is a sequence of subgraphs $G_1, \ldots, G_n$, called *accumulating subgraphs*, where $G_i = (V_i, E_i)$, $V_i = \{v_1, \ldots, v_i\}$, and E_i is the set of edges of G that join the vertices of V_i, for $1 \leq i \leq n$, The second is a sequence of subgraphs $G'_2, \ldots, G'_n$, called *interfacing subgraphs*, where $G'_i = (V'_i, E'_i)$, V'_i is the set of vertices of G_{i-1} that are adjacent to vertex v_i, and E'_i is the set of edges of G_{i-1} that join the vertices of V'_i, for $2 \leq i \leq n$.

The algorithm for computing the chromatic polynomial of a graph, based on the construction of accumulating subgraphs and interfacing subgraphs, is shown in Figure 3.1. This algorithm constructs the accumulating subgraphs according to an ordering of the set of vertices. At each iteration, the number of ways of coloring the newly added vertex is computed based on the order of the corresponding interfacing subgraph.

Theorem 3.3.2 *Let $G = (V, E)$ be a graph of order n and ω be an ordering of V. Suppose the interfacing subgraphs of G corresponding to ω are $G'_2, \ldots, G'_n$. Then:*

$$k \prod_{i=2}^{n} (k - |G'_i|) \leq C(G, k).$$

Let $G = (V, E)$ be a graph of order n. The algorithm proceeds as follows:

```
begin
    compute the degree for each vertex in $V$;
    generate an ordering $\omega = v_1, \ldots, v_n$ of $V$ by sorting the vertices in
        decreasing order on their degrees using the radix sort;
    $C(G, k) := k$;
    $G_1 := (\{v_1\}, \emptyset)$;
    for $i := 2$ to $n$ do
        compute the order $|G'_i|$ of the interfacing subgraph $G'_i$;
        $C(G, k) := C(G, k) \times (k - |G'_i|)$;
        construct the accumulating subgraph $G_i$;
    od
end
```

Figure 3.1: An approximation algorithm for computing the chromatic polynomial of a graph

The proof of this theorem is given in [Lin93]; we omit it here due to space constraints. Since the bound obtained from this may depend on the ordering chosen for the vertices in the graph, we use a heuristic to find a "good" ordering. The intuition behind the heuristic is that if the maximum order of the interfacing subgraphs is smaller, then we can get a nontrivial lower bound ($\neq 0$) on $C(G, k)$ for more values of k. Therefore, we use the ordering that sorts the vertices in the decreasing order on the degrees of vertices.

Let the graph under consideration have n vertices and m edges. First, the computation for the degrees of vertices in the graph can be performed in $O(n+m)$. Second, since the degrees of vertices in the graph are at most $n - 1$, we can sort the vertices using radix sort in $O(n)$. Third, the total cost for the construction of accumulating subgraphs G_i, $i \leq i \leq n$, is $O(n+m)$ because each edge in the graph is examined only twice. Finally, since only the orders of the interfacing subgraphs are needed to compute the chromatic polynomial, it is not necessary to construct the interfacing graphs. The orders of the interfacing subgraphs can be obtained as a by-product of constructing the accumulating graphs. Therefore, the complexity of the whole algorithm is $O(n+m)$.

3.4 Number of Solutions: Multiple Clauses

The previous section discussed the estimation of lower bounds on the number of solutions computed by a single clause. In this section we discuss how we can estimate the number of solutions for a group of clauses.

Theorem 3.4.1 *Consider a set of clauses $S = \{C_1, \ldots, C_n\}$ that all have the same head unification and tests. If n_i is a lower bound on the number of solutions generated by C_i, $1 \leq i \leq n$, then $\sum_{i=1}^{n} n_i$ is a lower bound on the total number of (not necessarily distinct) solutions generated by the set of clauses S.*

The restrictions in this theorem can be relaxed in various ways: we do not pursue this here due to space constraints. We can use the result above to estimate a lower bound on the number of solutions generated by a predicate for an input of size n as follows: partition the clauses for the predicate into clusters such that the clauses in each cluster have the same head unification and tests, so that Theorem 3.4.1 is applicable, and compute lower bound estimates of the number of solutions for each cluster. Then, if a number of different clusters—say, clusters $C_1, \ldots, C_k$, with number of solutions at least $n_1, \ldots, n_k$ respectively, may be applicable to an input of size n, then the number of solutions overall for an input of size n is given by $min(n_1, \ldots, n_k)$. The utility of this approach is illustrated by the following example.

Example 3.4.1 Consider the following predicate to generate all subsets of a set represented as a list:

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

As discussed in Section 3.1, recursion is handled by initially using a symbolic representation to set up difference equations, and then solving, or estimating solutions to, these equations. In this case, let (a lower bound on) the number of solutions computed by `subset`/2 on an input of size n be symbolically represented by $S(n)$. The first clause for the predicate yields the equation

$$S(0) = 1.$$

From Theorem 3.3.1, on an input of size n, $n > 0$, the second and third clauses each yield at least $S(n-1)$ solutions. Since they have the same head unification and tests, Theorem 3.4.1 is applicable, and the number of solutions given by these two clauses taken together is therefore at least $S(n-1) + S(n-1) = 2S(n-1)$. Thus, we have the equation

$$S(n) = 2S(n-1).$$

These difference equations can be solved to get the lower bound $S(n) = 2^n$ on the number of solutions computed by this predicate on an input of size n. □

3.5 Cost Estimation for Divide-and-Conquer Programs

A significant shortcoming of the approach to cost estimation presented is its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates are dependent. In the familiar quicksort program (see Example 3.5.1), for example, since either of the outputs of the partition predicate can be the empty list, the straightforward approach computes lower bounds under the assumption that both output can *simultaneously* be the empty list, and thereby significantly underestimates the cost of the program. In some sense, the reason for this loss of precision is that the approach outlined so far is essentially an independent attributes analysis [JM81]. However, even if we came up with a relational attributes analysis that kept track of relationships between the sizes of different output arguments of a predicate, it is not at all obvious how we might, systematically and from first principles, use this information to improve our lower bound cost estimates. For the quicksort program, for example, if the input list has length n, then the two output lists of the partition predicate have lengths m and $n - m - 1$ for some m, $0 \leq m < n$. The resulting cost equation for the recursive clause is of the form

$$C(n) = C(m) + C(n - m - 1) + \ldots \qquad (0 \leq m \leq n - 1)$$

In order to determine a worst-case lower bound solution to this equation we need to determine the value of m that minimizes the function $C(n)$, and doing this automatically, when we don't even know what $C(n)$ looks like, seems nontrivial. As a pragmatic solution, we argue that it may be possible to get quite useful results simply by identifying and treating common classes of divide-and-conquer programs specially.

In many of these programs, the sum of the sizes of the input for the "divide" predicates in the clause body is equal to the size of the input in the clause head minus some constant. This size relationship can be derived in some cases by the approach presented in [DL93]. However, this is not possible in other cases, since in this approach the size of each output argument is treated as a function only of the input sizes, independently of the sizes of other output arguments, and, as a result, relationships between the sizes of different output arguments are lost (consider for example the `partition`/4 predicate defined in example 3.5.1). Although the analysis does not break down for these cases, it can lose precision. A possible solution to improve precision is to use one of the recently proposed approaches for inferring size relationships for this class of programs [BK96, GDL95].

Assuming that we have the mentioned size relationship for these programs, in the cost analysis phase we obtain an expression of the form:

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(n)$ for $n > 0$, where k is an arbitrary value such that $0 \leq k \leq n-1$, C is a constant and $g(n)$ is any function.

where $y(n)$ denotes the cost of the divide-and-conquer predicate for an input of size n and $g(n)$ is the cost of the part of a clause body which does not contain any call to the divide-and-conquer predicate.

For each particular computation, we obtain a succession of values for k. Each succession of values for k yields a value for $y(n)$.

In the following we discuss how we can compute lower/upper bounds for expressions such as that for $\texttt{Cost}_{\texttt{qsort}}(n)$ in Example 3.5.1. Consider the expression:

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k)$ for $n > 0$, where k is an arbitrary value such that $0 \leq k \leq n-1$ and C is a constant.

A *computation tree* for such an expression is a tree in which each non-terminal node is labeled with $y(n)$, $n > 0$, and has two children $y(n-1-k)$ and $y(k)$ (left- and right-hand-side respectively), where k is an arbitrary value such that $0 \leq k \leq n{-}1$. Terminal nodes are labeled with $y(0)$ and have no children. Assume that we construct a tree for $y(n)$ following a depth-first traversal. In each non-terminal node, we (arbitrarily) chose a value for k such that $0 \leq k \leq n-1$. We say that the *computation succession* of the tree is the succession of values that have been chosen for k in chronological order, as the tree construction proceeds.

Lemma 3.5.1 *Any computation tree corresponding to the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$ *and* C *is a constant,*

has $n+1$ *terminal nodes and* n *non-terminal nodes.*

Proof By induction on n. For $n = 0$ the theorem holds trivially. Let us assume that the theorem holds for all m such that $0 \leq m \leq n$, then, we can prove that for all m such that $0 \leq m \leq n+1$ the theorem also holds by reasoning as follows: we have that $y(n+1) = y(n-k) + y(k)$, where k is an arbitrary value such that $0 \leq k \leq n$. Since $0 \leq k \leq n$, we also have that $0 \leq n-k \leq n$, and, by induction hypothesis, the number of terminal nodes in any computation tree of $y(n-k)$ (respectively $y(k)$) is $n-k+1$ (respectively $k+1$). The number of terminal nodes in any computation tree of $y(n+1)$ is the sum of the number of terminal nodes in the children of the node labeled with $y(n+1)$, i.e. $(n-k+1)+(k+1) = n+2$. Also, the number of non-terminal nodes in any computation tree of $y(n-k)$ (respectively $y(k)$) is $n-k$ (respectively k). The number of non-terminal nodes of any computation tree of $y(n+1)$ is the sum of the number of non-terminal nodes of the children of the node labeled with $y(n+1)$ plus one (the node $y(n+1)$ itself, since it is non-terminal) i.e. $1 + (n-k) + k = n+1$. ■

Theorem 3.5.2 *For any computation tree corresponding to the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$ *and* C *is a constant,*

it holds that $y(n) = (n+1) \times C$.

Proof By Lemma 3.5.1, any computation tree has $n+1$ terminal nodes labeled with $y(0)$ and the evaluation of each of these terminal nodes is C. ∎

Theorem 3.5.3 *Given the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(k)$ *for* $n > 0$*, where* k *is an arbitrary value such that* $0 \leq k \leq n-1$*,* C *is a constant and* $g(k)$ *a function,*

for any computation tree corresponding to it, it holds that $y(n) = (n+1) \times C + \sum_{i=1}^{n} g(k_i)$*, where* $\{k_i\}_{i=1}^{n}$ *is the computation succession of the tree.*

Proof By Lemma 3.5.1, any computation tree has $n+1$ terminal nodes and n non-terminal nodes. The evaluation of each terminal node yields the value C and each time a non-terminal node i is evaluated, $g(k_i)$ is added. ∎

In order to minimize (respectively maximize) $y(n)$ we can find a succession $\{k_i\}_{i=1}^{n}$ that minimizes (respectively maximizes) $\sum_{i=1}^{n} g(k_i)$. This is easy when $g(k)$ is a monotonic function, as the following corollary shows.

Corollary 3.5.1 *Given the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(k)$ *for* $n > 0$*, where* k *is an arbitrary value such that* $0 \leq k \leq n-1$*,* C *is a constant and* $g(k)$ *an increasing monotonic function,*

Then, the succession $\{k_i\}_{i=1}^{n}$*, where* $k_i = 0$ *(respectively* $k_i = n-1$*) for all* $1 \leq i \leq n$ *gives the minimum (respectively maximum) value for* $y(n)$ *of all computation trees.*

Proof It follows from Theorem 3.5.3 and from the fact that $g(k)$ is an increasing monotonic function. ∎

It follows from Corollary 3.5.1 that the solution of the difference equation (obtained by replacing k by 0):

$$\begin{aligned}&y(0) = C,\\&y(n) = y(n-1) + y(0) + g(0) \text{ for } n > 0,\end{aligned}$$

i.e. $(n+1) \times C + n \times g(0)$ is the minimum of $y(n)$, and the solution of the difference equation:

$$\begin{aligned}&y(0) = C,\\&y(n) = y(0) + y(n-1) + g(n-1) \text{ for } n > 0,\end{aligned}$$

i.e. $(n+1) \times C + n \times g(n-1)$ is the maximum of $y(n)$.

Note that we can replace $g(k)$ by any lower/upper bound on it to compute a lower/upper bound on $y(n)$. We can also take any lower/upper bound on each $g(k_i)$. For example, if $g(k)$ is an increasing monotonic function then $g(k_i) \leq g(n-1)$ and $g(k_i) \geq g(0)$ for $1 \leq i \leq n$, thus, $y(n) \leq (n+1) \times C + n \times g(n-1)$ and $y(n) \geq (n+1) \times C + n \times g(0)$.

Let's now assume that the function g depends on n and k:

Corollary 3.5.2 *Given the expression:*

$$\begin{aligned}&y(0) = C,\\&y(n) = y(n-1-k) + y(k) + g(n,k) \textit{ for } n > 0, \textit{ where } k \textit{ is an arbitrary}\\&\textit{value such that } 0 \leq k \leq n-1,\ C \textit{ is a constant and } g(n,k) \textit{ a function.}\end{aligned}$$

Then, the solution of the difference equation:

$$\begin{aligned}&f(0) = C,\\&f(n) = f(n-1) + C + L \textit{ for } n > 0,\end{aligned}$$

where L is a lower/upper bound on $g(n,k)$, is a lower/upper bound on $y(n)$ for all $n \geq 0$ and for any computation tree corresponding to $y(n)$. In particular, if $g(n,k)$ is an increasing monotonic function, then $L \equiv g(1,0)$ (respectively $L \equiv g(n,n-1)$) is a lower (respectively upper) bound on $g(n,k)$.

Example 3.5.1 Let us see how, using the described approach for divide-and-conquer programs, the lower-bound cost analysis can be improved. We first consider the analysis without the incorporation of the optimization, and then we compare with the result obtained when the optimization is used.

Consider the predicate `qsort`/2 defined as follows:

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

partition([], F, [], []).
partition([X|Y], F, [X|Y1], Y2) :-
        X =< F,
        partition(Y, F, Y1, Y2).
partition([X|Y], F, Y1, [X|Y2]) :-
        X > F,
        partition(Y, F, Y1, Y2).


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

Let $\texttt{Cost}_{\texttt{p}}(n)$ denote the cost (number of resolution steps) of a call to predicate `p` with an input of size n (in this example, the size measure used for all predicates is list length [DL93]). The estimation of cost functions proceeds in a "bottom-up" way as follows:

The difference equation obtained for append/3 is:

$$\texttt{Cost}_{\texttt{append}}(0, m) = 1 \text{ (the cost of head unification)},$$
$$\texttt{Cost}_{\texttt{append}}(n, m) = 1 + \texttt{Cost}_{\texttt{append}}(n - 1, m).$$

where $\texttt{Cost}_{\texttt{append}}(n, m)$ is the cost of a call to append/3 with input lists of lengths n and m (first and second argument, respectively). The solution to this equation is: $\texttt{Cost}_{\texttt{append}}(n, m) = n + 1$. Since this function depends only on n, we use the function $\texttt{Cost}_{\texttt{append}}(n)$ instead.

The difference equation for partition/4 is:

$$\texttt{Cost}_{\texttt{partition}}(0) = 1 \text{ (the cost of head unification)},$$
$$\texttt{Cost}_{\texttt{partition}}(n) = 1 + \texttt{Cost}_{\texttt{partition}}(n - 1).$$

where $\texttt{Cost}_{\texttt{partition}}(n)$ gives the cost of a call to partition/4 with an input list (first argument) of length n. The solution to this equation is: $\texttt{Cost}_{\texttt{partition}}(n) = n + 1$.

For qsort/2, we have:

$$\texttt{Cost}_{\texttt{qsort}}(0) = 1 \text{ (the cost of head unification)},$$
$$\texttt{Cost}_{\texttt{qsort}}(n) = 1 + \texttt{Cost}_{\texttt{partition}}(n - 1) + 2 \times \texttt{Cost}_{\texttt{qsort}}(0) + \texttt{Cost}_{\texttt{append}}(0)$$

because the computed lower bound for the size of the input to the calls to qsort and append is 0. Thus, the cost function for qsort/2 is given by:

$$\texttt{Cost}_{\texttt{qsort}}(0) = 1,$$
$$\texttt{Cost}_{\texttt{qsort}}(n) = n + 4, \text{ for } n > 0.$$

Now, we use the described approach for divide-and-conquer programs. Assume that we use the expression:

$$\begin{aligned}
&\texttt{Cost}_{\texttt{qsort}}(0) = 1,\\
&\texttt{Cost}_{\texttt{qsort}}(n) = 1 + \texttt{Cost}_{\texttt{partition}}(n-1) + \texttt{Cost}_{\texttt{qsort}}(k)\\
&\qquad + \texttt{Cost}_{\texttt{qsort}}(n-1-k) + \texttt{Cost}_{\texttt{append}}(k), \text{ for } 0 \leq k \leq n-1 \text{ and } n > 0.
\end{aligned}$$

Replacing values, we obtain:

$$\begin{aligned}
&\texttt{Cost}_{\texttt{qsort}}(n) = n + k + 2 + \texttt{Cost}_{\texttt{qsort}}(k) + \texttt{Cost}_{\texttt{qsort}}(n-1-k),\\
&\qquad \text{for } 0 \leq k \leq n-1.
\end{aligned}$$

According to Corollary 3.5.2, by giving to n and k the minimum possible value, i.e. 1 and 0 respectively, we have that $n + k + 2 \geq 3$, and thus we replace $n + k + 2$ by 3 in order to obtain a lower bound on the former expression, which yields:

$$\texttt{Cost}_{\texttt{qsort}}(n) = 3 + \texttt{Cost}_{\texttt{qsort}}(k) + \texttt{Cost}_{\texttt{qsort}}(n-1-k), \text{ for } 0 \leq k \leq n-1.$$

which is equivalent to the difference equation:

$$\texttt{Cost}_{\texttt{qsort}}(n) = 3 + 1 + \texttt{Cost}_{\texttt{qsort}}(n-1), \text{ for } n > 0.$$

The solution of this equation is $\texttt{Cost}_{\texttt{qsort}}(n) = 4n+1$, which is an improvement on the former lower bound. □

The previous results can be easily generalized to cover multiple recursive divide-and-conquer programs and programs where the sum of the sizes of the input for the “divide” predicates in the clause body is equal to the size of the input in the clause head minus some constant which is not necessarily 1.

3.6 Implementation

We have implemented a prototype of a lower bound size/cost analyzer, by recoding the version of CASLOG [DL93] currently integrated in the *CiaoPP* system [HBPLG99, HBC$^+$99]. The analysis is fully automatic, and only requires type information for the program entry point. Types, modes and size measures are automatically inferred by the system. Table 3.1 shows some accuracy and efficiency results of the lower bound cost analyzer. The second column of the table shows the cost function (which depends on the size of the input arguments) inferred by the analysis. T_{tms} is the time required by the type, mode, and size measure analysis (SPARCstation 10, 55MHz, 64Mbytes of memory), T_{nf} the time required by the non-failure analysis, and T_s and T_{ca} are the time required by the size and cost analysis respectively. **Total** is the total analysis time ($Total = T_{tms} + T_{nf} + T_s + T_{ca}$). All times are given in milliseconds.

Program	**Cost function**	T_{tms}	T_{nf}	T_s	T_{ca}	**Total**
fibonacci	$\lambda x.1.447 \times 1.618^x + 0.552 \times (-0.618)^x - 1$	90	10	20	20	140
hanoi	$\lambda x.x2^x + 2^{x-1} - 2$	430	30	60	60	580
qsort	$\lambda x.4x + 1$	420	50	70	50	590
nreverse	$\lambda x.0.5x^2 + 1.5x + 1$	220	20	30	30	300
mmatrix	$\lambda\langle x, y\rangle.2xy + 2x + 1$	350	90	90	90	620
deriv	$\lambda x.x$	1010	80	170	120	1,380
addpoly	$\lambda\langle x, y\rangle x + 1$	220	70	40	30	360
append	$\lambda x.x + 1$	100	20	10	10	140
partition	$\lambda x.x + 1$	175	30	30	20	255
substitute	$\lambda\langle x, y, z\rangle.x$	70	50	110	100	330
intersection	$\lambda\langle x, y\rangle.x + 1$	150	130	20	30	260
difference	$\lambda\langle x, y\rangle.x + 1$	140	90	20	40	290

Table 3.1: Accuracy and efficiency of the lower bound cost analysis

3.7 Application to Automatic Parallelization

We have interfaced the cost analysis stage (see Section 3.6) with the granularity control system described in Chapter 2 (which is also integrated in the CIAO system as another stage, and which includes an annotator which transforms programs to perform granularity control). The result is a complete program parallelizer with (lower bound cost based) granularity control. Since our objective herein is simply to study the usefulness of the lower bound estimates produced, only a very simple granularity control strategy has been selected: goals are always executed in parallel provided their grain sizes are estimated to be greater than a given fixed threshold, which is a constant for all programs. Also, the versions of the programs that perform granularity control are simple source-to-source transformations which add granularity control tests to the original versions. A discussion of more advanced strategies that include variable thresholds (which depend on parameters such as data transfer cost, number of processor, system load, etc.), lower level transformations, and performing goal groupings to increase granularity can be found in Chapter 2

programs	**seq**	**ngc**	**gclb(175)**	**gclb(959)**
mmatrix(100)	52.389	74.760 (0.70)	29.040 (1.80)	27.981 (1.87)
mmatrix(50)	6.469	5.978 (1.08)	3.378 (1.92)	3.758 (1.72)
fib(19)	0.757	1.458 (0.52)	0.128 (5.93)	0.103 (7.32)
hanoi(13)	1.442	1.464 (0.98)	0.677 (2.13)	0.619 (2.33)
qsort(1000)	0.475	0.414 (1.15)	0.230 (2.06)	0.314 (1.51)
qsort(3000)	4.142	2.423 (1.71)	1.094 (3.79)	1.575 (2.63)

Table 3.2: Granularity control results for benchmarks on ECL^{i}PS^e.

We have performed some preliminary experiments in which a series of benchmarks have been parallelized automatically, with the compiler option corresponding to inclusion of granularity control both enabled and disabled. The resulting programs have been executed on the ECL^{i}PS^e system using 10 workers, and running on a SUN SPARC 2000 SERVER with 10 processors. We have chosen this system, which implements and-parallelism on top of or-parallelism, because it has considerably greater parallel task execution overhead than systems which implement and-parallelism natively (such as, for example, the &-Prolog engine used in the CIAO system). As a result, this system offers an interesting challenge – it proved very difficult to achieve and-parallel speedups on it automatically with previous parallelizers.

Table 3.2 presents the results. It shows wall-clock execution times in seconds. Results are given for the sequential execution (**seq**), the parallel execution without granularity control (**ngc**), and the versions which perform granularity control (**gclb(175)** and **gclb(959)**). The two numbers correspond to two different choices of threshold, and illustrate the comparatively low sensitivity of the results to the choice of this parameter that we have observed.

The results of the experiments appear promising, in the sense that the granularity control does improve speedups in practice, in a quite challenging situation. On systems with higher overheads, such as distributed systems, the benefits can be much larger, although it may be difficult to achieve actual speedups in some cases (i.e., given high enough overheads, the result of the granularity analysis can often be simply a sequential program). In any case, we believe that it is possible to improve these results significantly by using more sophisticated control strategies, as mentioned above.

3.8 Chapter Conclusions

We have developed an analysis that infers lower bounds on the cost of procedures as functions of input data size (and that is able to properly deal with divide-and-conquer programs). We have implemented this cost analysis and integrated it in the `ciaopp` system. Finally, we have performed experiments that show that the analysis is accurate and efficient.

Despite a suggestive similarity in names, our work is quite different from Basin and Ganzinger's work on complexity analysis based on ordered resolution [BG96]. They consider resolution based on a well-founded total ordering on ground atoms, and use this to examine the complexity of determining, given a set of Horn clauses N and a ground Horn clause C, whether $N \models C$. Our work, by contrast, is based on an operational formulation of logic program execution that is not restricted to ground queries (or, for that matter, to Horn programs, since it is easy to handle features such as cuts and negation by failure). Because operational aspects of program execution are modeled more accurately in our approach, the results obtained are considerably more precise.

Finally, it is worth of mentioning that information about the runtime cost of computations can be useful for a variety of applications (besides granularity control, which is the main motivation for which we developed it), such as for example assisting program transformation systems to find the optimal transformations, choosing between different algorithms, program efficiency debugging, and query optimization in deductive databases.

Chapter 4

Non-failure Analysis

In this Chapter we provide a method whereby, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail (i.e., to produce at least one solution or not terminate). The technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a set of variables (i.e. for any element that belongs to the type, at least one test will succeed). We show that the problem of determining a covering is undecidable in general, and give decidability and complexity results for the Herbrand and linear arithmetic constraint systems. We give sound algorithms for determining covering that are precise and efficient in practice. Based on this information, we show how to identify goals and procedures that can be guaranteed to not fail at runtime. Applications of such non-failure information include programming error detection, program transformations and parallel execution optimization, avoiding speculative parallelism and estimating lower bounds on the computational costs of goals, which can be used for granularity control, and that is the main motivation for this dissertation. Finally, we report on an implementation of our method and show that better results are obtained than with previously proposed approaches.

4.1 Motivation

There are two important motivations for considering compile-time analyses to identify non-failure in logic programs. The first is that it is usually very useful to be able to identify badly-behaved programs where possible. For example, in statically typed languages, the behavior one expects is that program components will be used in a way consistent with their types, and compile-time type checking is used to detect departures from this expected behavior. While this does not rule out programming errors, it makes it a lot simpler to identify and localize certain kinds of common programming errors. Similarly, in logic programs, the usual expectation is that a predicate will succeed and produce one or more solutions. In most logic programming systems, however, the only checking that is done is a rather simplistic—though useful—check about the naming of singleton variables. The second reason is that knowledge of non-failure can be used to aid a number of program transformations and optimizations. For example, we may want to execute possibly-failing goals ahead of non-failing goals where possible; and in parallel systems, knowledge of non-failure can be used to avoid speculative parallelism and to estimate lower bounds on the computational costs of goals [DLGHL94, DLGHL97], which can be used for granularity control of parallel tasks [LGHD96], and that is the main motivation for this dissertation.

The problem with naive attempts to infer non-failure is that, in general, it is always possible for a goal to fail because "bad" argument values cause a failure during head unification. An obvious solution would be to try and rule out such argument values by considering the types of predicates. However, most existing type analyses provide upper approximations, in the sense that the type of a predicate is a superset of the set of argument values that are actually encountered at runtime. Unfortunately, straightforward attempts to address this issue, for example by trying to infer lower approximations to the calling types of predicates, fail to yield nontrivial lower bounds for most cases.

4.2 Preliminaries

We assume an acquaintance with the basic notions of logic programming. In order to reason about non-failure, it is necessary to distinguish between unification operations that act as tests (and which may fail), and output unifications that act as assignments (and always succeed). To this end, we assume that programs are *moded*, i.e., for each unification operation in each predicate, we know whether the operation acts as a test or creates an output binding (note that this is weaker than most conventional notions of moding in that it does not require input arguments to be ground, and allows an output argument to occur as a subterm of an input argument). Where it is necessary to emphasize the input tests in a clause, we write the clause in "guarded" form, as

$$p(x_1, \ldots, x_n) :- \mathit{input\_tests}(x_1, \ldots, x_n) \parallel \mathit{Body}.$$

Consider a predicate defined by the clauses:

$$\mathit{abs}(X, Y) :- X \geq 0 \parallel Y = X.$$
$$\mathit{abs}(Y, Z) :- Y < 0 \parallel Z = -Y.$$

Suppose we know that this predicate will always be called with its first argument bound to an integer. Obviously, for any particular call, one or the other of the tests '$X \geq 0$' and '$X < 0$' may fail; however, taken together, one of them will succeed. This shows that we cannot rely on examining the tests of each clause separately: it is necessary to collect them together and examine the behavior of the collection as a whole. When collecting tests together, however, we must be careful to make sure that we do not get confused by different variable names in different clauses. For example, in the *abs* predicate defined above, we need to make sure that:

1. we notice that the variable X in the first clause and the variable Y in the second clause actually refer to the same component of the arguments to the predicate; and

2. we do not confuse the variable Y in the first clause with the variable Y in the second clause.

These pitfalls can be avoided by normalizing clauses so that they use variable names consistently and according to a predefined convention. We rely on the usual approach of using sequences of integers to encode paths in ordered trees: the empty sequence ε corresponds to the root node of the tree, and if a sequence π corresponds to a node N, then the sequences $\pi 1, \ldots, \pi k$ correspond to its children $N_1, \ldots, N_k$ taken in order. We adopt the convention that a variable in a clause is designated as X_π, where π is (the sequence encoding) the path from the root of the clause, labeled `:-`/2, to the leftmost occurrence of that variable. To enhance readability, the examples used in this Chapter will not resort to explicitly naming variables in this way unless necessary, with the understanding that the algorithms are defined with respect to normalized clauses.

4.3 Types, Tests, and Coverings

A type refers to a set of terms, and can be denoted by using several type representations: e.g. *type terms* and *regular term grammars* as in [DZ92] (which is the one we use), or *type graphs* as in [JB92]). Thus, we assume the existence of an infinite set of *type symbols*. The type symbol μ denotes the type of the entire Herbrand universe, and the symbol ϕ denotes the empty type. We include the following five definitions taken from [DZ92], because we think it can help to better understand the algorithms we present here, and also makes this text more auto-contained.

Definition 4.3.1 A *type term* is defined inductively as follows:

1. A constant symbol is a type term.

2. A variable is a type term.

3. A type symbol is a type term.

4. If f is a n-ary function symbol and each τ_i is a type terms, $f(\tau_1, \ldots, \tau_n)$ is a type term.

A *pure type term* is a variable-free type term. A *logical term* is a type-symbol-free type term. ∎

We assume (as in [DZ92]) that type symbols are partitioned into non-empty subsets, called *base types*. There are also effective test for membership of a given term in each base type. Examples of *base types* we use in our non-failure analysis are:

1. The set of all constant symbols that represent integer numbers. This base type is denoted as *int*.

2. The set of all constant symbols which do not represent numbers (according to Prolog terminology, the set of all *atoms*). This base type denoted as *atm*.

Definition 4.3.2 A *type rule* is an expression of the form $\alpha \rightarrow \Upsilon$, where α is a type symbol, and Υ is a set of pure type terms. ∎

We denote sets of type rules, that is, *regular term grammars*, by the letter T (as in [DZ92]).

Definition 4.3.3 A type symbol α, is *defined* in, or by, a set of type rules T if there exists a type rule $(\alpha \rightarrow \Upsilon) \in T$. A type term τ is *defined* by a set of type rules T if each type symbol in τ is defined in T. ∎

We assume (as in [DZ92]) that each type symbol ocurring in a set of type rules T is eiher μ, ϕ, a base type symbol, or a type symbol defined in T, and that each type symbol defined in T has *exactly* one defining type rule in T.

Definition 4.3.4 A type rule $\alpha \rightarrow \Upsilon$ is *deterministic* if no element of Υ is a type symbol and there is no pair of pure type terms $\tau_1, \tau_2 \in \Upsilon$, such that $\tau_1 \neq \tau_2$, $\tau_1 = f(\tau_1^1, \ldots, \tau_n^1)$, and $\tau_2 = f(\tau_1^2, \ldots, \tau_n^2)$. ■

The class of types that can be described by deterministic type rules is the same as the class of *tuple-distributive regular types* [DZ92].

More detailed an interesting information on type related issues, may be found in papers such as [DZ92, JB92]. Thus, we do not pursue them further here for the sake of brevity.

The non-failure analysis we describe here is based on *regular types* [DZ92], specified by *regular term grammars* that meet the assumptions above. We show soundness and completeness results for both, the case when the *regular types* are *tuple-distributive*, and when they are not.

In the rest of this section we define some concepts which are useful for describing the algorithms we propose.

Definition 4.3.5 An *infinite function symbol type* Ψ is an infinite set of terms, such that the set $\{f \mid f(t_1, \ldots, t_n) \in \Psi\}$, tah is the set of main function symbols of all the terms in Ψ, is infinite. ■

Let $\mathsf{type}[p]$ denote the type of each predicate p in a given program. In this Chapter, we are concerned exclusively with "calling types" for predicates—in other words, when we say "*a predicate p in a program P has type* $\mathsf{type}[p]$", we mean that in any execution of the program P starting from some class of queries of interest, whenever there is a call $p(\bar{t})$ to the predicate p, the argument tuple $\bar{t}$ in the call will be an element of the set denoted by $\mathsf{type}[p]$.

We denote the Herbrand Universe (i.e., the set of all ground terms) as $\mathcal{H}$, and the set of n–tuples of elements of $\mathcal{H}$ as $\mathcal{H}^n$.

Given a (finite) set of variables V, a *type assignment* over V is a mapping from V to a set of types. A type assignment ρ over a set of variables $\{x_1, \dots, x_n\}$ is written as $(x_1 : a_1, \dots, x_n : a_n)$, where $\rho(x_i) = a_i$, for $1 \leq i \leq n$, and a_i is a type representation. Alternatively, we also use the notation $\bar{x} : \bar{T}$, where $\bar{x}$ is the tuple of variables $(x_1, \dots, x_n)$, and $\bar{T}$ is the tuple of types $(a_1, \dots, a_n)$. Given a term t and a type representation T, we abuse of terminology and say $t \in T$, meaning that t belongs to the set of terms denoted by T.

A *primitive test* is an "atom" whose predicate is a built-in such as the unification or some arithmetic predicate $(<, >, \leq, \geq, \neq$, etc.) which acts as a "test". Define a *test* to be either a primitive test, or a conjunction $\tau_1 \wedge \tau_2$ or a disjunction $\tau_1 \vee \tau_2$, or a negation $\neg\tau_1$, where τ_1 and τ_2 are tests.

Fundamental to our approach to detecting non-failure is the notion of a test "covering" a type assignment:

Definition 4.3.6 A test $S(\bar{x})$ *covers* a type assignment $\bar{x} : \bar{T}$, where $\bar{x}$ and $\bar{T}$ are a tuple of variables and a tuple of nonempty types respectively, if for every $\bar{t} \in \bar{T}$ it is the case that $\bar{x} = \bar{t} \models S(\bar{x})$. ■

Consider a predicate p defined by n clauses, with input tests $\tau_1, \dots, \tau_n$:

$$p(\bar{x}) :- \tau_1(\bar{x}) \parallel Body_1.$$
$$\dots$$
$$p(\bar{x}) :- \tau_n(\bar{x}) \parallel Body_n.$$

We refer to the test $\tau(\bar{x}) \equiv \tau_1(\bar{x}) \vee \dots \vee \tau_n(\bar{x})$ as the *input test of p.* Suppose that the predicate p has type $\mathsf{type}[p]$: in the interest of simplicity, we sometimes abuse terminology and say that the predicate p covers the type $\mathsf{type}[p]$ if the input test $\tau(\bar{x})$ of p covers the type assignment $\bar{x} : \mathsf{type}[p]$.

Define the "calls" relation between predicates in a program as follows: p calls q, written $p \leadsto q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p, and let $\leadsto^\star$ denote the reflexive transitive closure of $\leadsto$. The importance of the notion of covering is expressed by the following result:

Theorem 4.3.1 *A predicate p in the program is non-failing if, for each predicate q such that $p \leadsto^\star q$, q covers* $\mathsf{type}[q]$*.*

Proof Suppose that p can fail, i.e., there is a goal $p(\bar{t})$, with $\bar{t} \in \mathsf{type}[p]$, that fails. We show, by induction on the number of resolution steps n, that in any failed execution sequence for this goal, there is some predicate q such that $p \leadsto^\star q$ and q does not cover its type. The base case is for $n = 0$, i.e., where failure occurs because head unification and tests failed for each clause defining the called procedure. From the definition of a covering, this can happen only if the tests of the called procedure did not cover the type of the actual parameters. For the inductive case, assume that the theorem holds for all predicates that have a goal that fails in less than k resolution steps, and consider an execution that fails in k resolution steps. It must be that for each clause of the called procedure p for which head unification and tests succeed, some body literal L, with predicate symbol q, fails. Now the failure of L must involve fewer than k resolution steps, and so from the induction hypothesis, there is a predicate r such that $q \leadsto^\star r$ and r does not cover $\mathsf{type}[r]$. Since L is a body literal in a clause defining p, we have $p \leadsto q$, which means that $p \leadsto^\star r$ as well. This establishes that p can fail only if there is some predicate q such that $p \leadsto^\star q$ and q does not cover its type. The theorem follows.

Assume that p can fail, i.e., there is a goal $p(\bar{t})$, with $\bar{t} \in \mathsf{type}[p]$, that fails. It is a straightforward induction on the number of resolution steps to show that there is a q such that $p \leadsto^\star q$ and q does not cover its type. ∎

Note that non-failure does not imply success: a predicate that is non-failing may nevertheless not produce an answer because it does not terminate. This is illustrated by the predicate, defined by the single clause given below, which is non-failing and—on most existing Prolog systems—non-terminating:

$$p(X) :- \, p(X).$$

Ideally, we would like to have a decision procedure to determine whether a test covers a given type assignment. Unfortunately, this is impossible in general, as the following result shows:

Theorem 4.3.2 *Given an arbitrary test and type assignment, it is in general undecidable whether the test covers the type assignment.*

Proof The proof is straightforward from a result, due to Matijasevič, that shows that determining the existence of (integer) solutions for arbitrary Diophantine equations is undecidable [Mat70]. Given an arbitrary polynomial $\phi(x_1, \ldots, x_n)$, consider the test $\phi(x_1, \ldots, x_n) \neq 0$. This test covers the type assignment $(x_1 : \texttt{integer}, \ldots, x_n : \texttt{integer})$ if and only if every possible assignment of integers to the variables $x_1, \ldots, x_n$ causes the polynomial ϕ to take on a non-zero value, i.e., if and only if the Diophantine equation $\phi(x_1, \ldots, x_n) = 0$ has no integer solutions. But since the problem of determining the existence of integer roots for an arbitrary Diophantine equation is undecidable, it follows that the problem of determining whether an arbitrary test covers an arbitrary type assignment is also undecidable. ■

We are therefore forced to resort to sound (but, necessarily, incomplete) algorithms to determine coverings. In the remainder of this section we show that covering problems are decidable for most cases arising in practice—in particular, for equality and disequality tests over the Herbrand domain and for linear arithmetic tests—and give algorithms for deciding covering for these cases. Given a test and a type assignment that we want to check for covering, our approach is to:

1. first partition the test such that tests in different resulting partitions involve different constraint systems, and

2. then apply to each partition a covering algorithm particular to the corresponding constraint system.

In this Chapter we consider two commonly encountered constraint systems:

- first order terms with equality and disequality tests, on variables with *tuple-distributive regular types* [DZ92] (types which are specified by regular term grammars in which each type symbol has exactly one defining type rule and each type rule is *deterministic*); and

- linear arithmetic tests on integer variables.

The following example illustrates the partitioning process:

Example 4.3.1 Consider the test:

$$\mathtt{X1} = [\,] \lor (\mathtt{X1} = [\mathtt{H}|\mathtt{L}] \land \mathtt{H} > \mathtt{H1}) \lor (\mathtt{X1} = [\mathtt{H}|\mathtt{L}] \land \mathtt{H} \leq \mathtt{H1}),$$

together with the type assignment $(\mathtt{X1} : \mathtt{intlist}, \mathtt{H1} : \mathtt{integer})$, where:

$$\mathtt{intlist} \rightarrow \{[\,], [\mathtt{integer} \mid \mathtt{intlist}]\}.$$

This is partitioned into two components:

- the Herbrand test $\mathtt{X1} = [\,] \lor \mathtt{X1} = [\mathtt{H}|\mathtt{L}]$ together with the type assignment $(\mathtt{X1} : \mathtt{intlist})$; and

- the linear arithmetic test $\mathtt{H} > \mathtt{H1} \lor \mathtt{H} \leq \mathtt{H1}$ together with the type assignment $(\mathtt{H} : \mathtt{integer}, \mathtt{H1} : \mathtt{integer})$.

□

4.3.1 Covering in the Herbrand Domain

Decidability and Complexity

While covering is undecidable in the presence of arbitrary arithmetic operations, it turns out to be decidable if we restrict ourselves to equations and disequations over Herbrand terms. Before discussing the algorithm for this, we give a result on the complexity of the covering problem for Herbrand:

Theorem 4.3.3 *The covering problem for the Herbrand domain is co-NP-hard. It remains co-NP-hard even if we restrict ourselves to equality tests.*

Proof By reduction from the problem of determining whether a propositional formula in disjunctive normal form, containing at most 3 literals per disjunct, is a tautology ([GJ79], problem LO8). This problem is known to be co-NP-complete (see [GJ79], problem LO8). Given a DNF formula $C \equiv C_1 \vee \cdots \vee C_n$ over a set of variables $\{x_1, \ldots, x_m\}$, where each C_i is a conjunction of at most 3 literals, we generate a test $S \equiv S_1 \vee \cdots \vee S_n$. S_i is generated from C_i as follows: let $C_i \equiv L_{i1} \wedge L_{i2} \wedge L_{i3}$, where the L_{ij} are literals, then $S_i \equiv \widehat{L}_{i1} \wedge \widehat{L}_{i2} \wedge \widehat{L}_{i3}$, with $\widehat{L}_{ij}$ defined as follows, $1 \leq j \leq 3$: if $L_{ij} = x$ for some variable x, then $\widehat{L}_{ij}$ is the test $x = \mathit{true}$; if $L_{ij} = \neg x$ for some variable x, then $\widehat{L}_{ij}$ is the test $x = \mathit{false}$. Let **Bool** denote the type $\{\mathit{true}, \mathit{false}\}$, then we consider the type assignment $(x_1 : \textbf{Bool}, \ldots, x_m : \textbf{Bool})$. It is not difficult to see that the test S covers this type assignment if and only if every truth assignment to the variables of C makes it evaluate to *true*, i.e., if and only if C is a tautology. The theorem follows. ■

A Decision Procedure

The decision procedure presented here is inspired by a result, due to Kunen [Kun87], that the emptiness problem is decidable for Boolean combinations of (notations for) certain “basic” subsets of the Herbrand universe of a program. It also uses straightforward adaptations of some operations described by Dart and Zobel [DZ92].

The reason the covering algorithm for Herbrand is as complex as it is is that we want a *complete* algorithm for equality and disequality tests. It is possible to simplify this considerably if we are interested in equality tests only. Before describing the algorithm, we introduce some definitions and notation.

We use the notions (to be defined in the following) of *type-annotated term*, and in general *elementary set*, as representations which denote some subsets of $\mathcal{H}^n$ (for some $n \geq 1$). Given a representation S (elementary set or type-annotated term), $Den(S)$ refers to the subset of $\mathcal{H}^n$ denoted by S.

Definition 4.3.7 [type-annotated term] A *type-annotated term* is a pair $M = (\bar{t}, \rho)$, where $\bar{t}$ is a tuple of terms, and ρ is a type assignment $(x_1 : T_1, \ldots, x_k : T_k)$. ■

To enhance readability, the type of x_i in M, i.e., $\rho(x_i)$, will sometimes be written as $type(x_i, M)$ or as $type(x_i, \rho)$. Also, given a type-annotated term M, we denote its tuple of terms and its type assignment as $\bar{t}_M$ and ρ_M respectively. A type-annotated term $(\bar{t}, \rho)$ denotes the set of all the ground terms $\bar{t}\theta$ such that $x\theta \in type(x, \rho)$ for each variable in $\bar{t}$.

Given a type-annotated term $(\bar{t}, \rho)$, the tuple of terms $\bar{t}$ can be regarded as a *logical term* (i.e. a type-symbol-free type term) and ρ can be considered to be a type substitution, so that, if we apply this type substitution to $\bar{t}$, we get a pure type term (a variable-free type term). This is useful for defining the "intersection" and "inclusion" operations over of type-annotated terms (that we define later), using the algorithms described by Dart and Zobel [DZ92] for performing these operations over pure type terms. When we have a type-annotated term $(\bar{t}, \rho)$ such that $\rho(x) = \mu$ for each variable x in $\bar{t}$, we omit the type assignment ρ for brevity and use the tuple of terms $\bar{t}$ (recall that μ denotes the type of the entire

Herbrand universe). Thus, a tuple of terms $\bar{t}$ with no associated type assignment can be regarded as a type-annotated term which denotes the set of all ground instances of $\bar{t}$.

Definition 4.3.8 [elementary set] An elementary set is defined as follows:

- Λ is an elementary set, and denotes the empty set (i.e., $Den(\Lambda) = \emptyset$);
- a type-annotated term $(\bar{t}, \rho)$ is an elementary set; and
- if A and B are elementary sets, then $A \otimes B$, $A \oplus B$ and $comp(A)$ are elementary sets that denote, respectively, the sets of (tuples of) terms $Den(A) \cap Den(B)$, $Den(A) \cup Den(B)$, and $\mathcal{H}^n \setminus Den(A)$. ■

We define the following relations between elementary sets:

- $A \sqsubseteq B$ iff $Den(A) \subseteq Den(B)$.
- $A \sqsubset B$ iff $Den(A) \subset Den(B)$.
- $A \simeq B$ iff $Den(A) = Den(B)$.

Definition 4.3.9 [cobasic set] A cobasic set is an elementary set of the form $comp(\bar{t})$, where $\bar{t}$ is a tuple of terms (recall that $\bar{t}$, is in fact a type-annotated term $(\bar{t}, \rho)$ such that $\rho(x) = \mu$ for each variable x in $\bar{t}$). ■

Definition 4.3.10 [minset] A *minset* is either Λ or an elementary set of the form $A \otimes comp(B_1) \otimes \cdots \otimes comp(B_p)$, for some $p \geq 0$, where A is a tuple of terms, $comp(B_1), \ldots, comp(B_p)$ are cobasic sets, and for all $1 \leq i \leq p$, $B_i = A\theta_i$ and $A \not\sqsubseteq B_i$ for some substitution θ_i (i.e. $B_i \sqsubset A$). ■

For brevity, we write a minset of the form $A \otimes comp(B_1) \otimes \cdots \otimes comp(B_p)$ as A/C, where $C = \{comp(B_1), \ldots, comp(B_p)\}$. We also denote the tuple of terms of a cobasic set $Cob \equiv comp(B)$ as $\bar{t}_{Cob}$, i.e. $\bar{t}_{Cob} \equiv B$.

Example 4.3.2 Here are some examples of type-annotated terms (M, N, and S). Let M be the type-annotated term $((x, y), (x : \gamma_1, y : \gamma_2))$, where $\gamma_1 ::= f(\mu)$, and $\gamma_2 ::= g(\mu) \mid h(\mu)$. In this case, ρ_M denotes the type assignment $(x : \gamma_1, y : \gamma_2)$, and $\bar{t}_M$ denotes the tuple of terms (x, y). Let S be a type-annotated term, such that $\bar{t}_S \equiv (f(z), w)$ and $\rho_S \equiv (z : \mu, w : \gamma_2)$. We have that M and S describe the same subset of $\mathcal{H}^n$, i.e., $Den(M) = Den(S)$. Let N be a type-annotated term, where $\bar{t}_N \equiv (f(v_1), g(v_2), v_3, v_4, f(a), f(v_5), v_6)$, $\rho_N \equiv (v_1 : \mu, v_2 : list, v_3 : \gamma_2, v_4 : \gamma_3, v_5 : \gamma_3, v_6 : list)$, and $\gamma_3 ::= a \mid b$ and $list ::= [\,] \mid [\mu|list]$. We have that $\rho(v_1) = \mu$, and $\rho(v_3) = \gamma_2$. $\square$

Definition 4.3.11 [type-annotated term instance] Let α and β be two type-annotated terms. We say that α is an instance of β if $\alpha \sqsubset \beta$ and there is a substitution θ such that $\bar{t}_\alpha = \bar{t}_\beta \theta$. ■

Consider a predicate p defined by n clauses, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$:

$p(\bar{x}) :- \tau_1(\bar{x}) \parallel Body_1.$

$\ldots$

$p(\bar{x}) :- \tau_n(\bar{x}) \parallel Body_n.$

Suppose that the predicate p has type $\mathsf{type}[p]$. Testing whether the input test of p, $\tau(\bar{x})$, covers the type assignment $\bar{x} : \mathsf{type}[p]$ can be reduced to test whether:

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_n \tag{4.1}$$

where M is a type-annotated term which is a representation of $\bar{x} : \mathsf{type}[p]$, and each S_i is a minset, which is the representation of $\tau_i(\bar{x})$. The test $\tau_i(\bar{x})$ can be transformed into the minset S_i as follows:

1. Assume that the test $\tau_i(\bar{x})$ is of the form $E_i \wedge D_i^1 \wedge \cdots \wedge D_i^m$, where E_i is the conjunction of all unification tests of $\tau_i(\bar{x})$ (i.e., a system of equations) and each D_i^j a disunification test (i.e., a disequation).

2. Let θ_i be the substitution associated with the solved form of E_i (this can be computed by using the techniques of Lassez et al. [LMM88]).
3. Let θ_i^j, for $1 \leq j \leq m$, be the substitution associated with the solved form of $E_i \wedge N_i^j$, where N_i^j is the negation of D_i^j.
4. $S_i = A \otimes comp(B_i^1) \otimes \cdots \otimes comp(B_i^m)$, where $A = (\bar{x})\theta_i$ and $B_i^j = (\bar{x})\theta_i^j$, for $1 \leq j \leq m$.

Taking into account condition 4.1, we have that:

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_m \text{ iff } M \otimes comp(S_1) \otimes \cdots \otimes comp(S_m) \simeq \Lambda$$

We then can write $comp(S_1) \otimes \cdots \otimes comp(S_m)$ into disjunctive normal form as $M_1 \oplus \cdots \oplus M_u$, where each M_i is a minset. [1] Since

$$M \otimes (M_1 \oplus \cdots \oplus M_u) \simeq M \otimes M_1 \oplus \cdots \oplus M \otimes M_u,$$

we have that

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_m \text{ iff } M \otimes M_i \simeq \Lambda \text{ for all } 1 \leq i \leq u.$$

Thus, the fundamental problem is to devise an algorithm to test whether $M \otimes S \simeq \Lambda$, where M is a type-annotated term and S a minset. The algorithm that we propose is given by the boolean function *empty*(M, S), defined in Figure 4.1.[2]

We do some definitions before describing the algorithm.

Definition 4.3.12 [*intersection*] Let M and A be a type-annotated term and a tuple of terms respectively. Let $unify(\tau_1, \tau_2, T, \Theta)$ be the function described

[1] Note that $\oplus$, $\otimes$, and *comp* constitute a Boolean algebra, and the operation $\otimes$ is computable for type-annotated terms.

[2] We use the type representation of [DZ92], and assume that there is a common set of rules where type symbols are described. For brevity, we omit such set of rules in the description of the algorithms.

in [DZ92], where τ_1 and τ_2, are type terms, Θ a type substitution for the variables in τ_1 and τ_2, and T a set of deterministic type rules defining τ_1, τ_2, and Θ.

We define the function *intersection* as follows:
$\mathit{intersection}(M, A)$ is the type annotaded term form of the pair (τ_f, T_f, Θ_f) (this translation is trivial, so that details are omitted), where:

- $(\tau_f, T_f, \Theta_f) = \mathit{unify}(\bar{t}_M, A, T, \sigma)$, where T is the set of rules defining the type symbols appearing in ρ_M (note that $\bar{t}_M$ and A are in fact type terms), and σ is a type substitution constructed as follows:

$$x\sigma = \begin{cases} t & \text{if } \ x \in \mathit{vars}(M) \ \text{ and } \ \mathit{type}(x, M) = t \\ x & \text{otherwise.} \end{cases}$$

■

Theorem 4.3.4 *The following holds for the function intersection:*

- $\mathit{intersection}(R, B)$ *terminates,*
- $\mathit{intersection}(R, B) = I$ *iff* $R \otimes B \simeq I$*, and*
- $\mathit{intersection}(R, B) = \Lambda$ *iff* $R \otimes B \simeq \Lambda$.

Proof It follows from Theorem 5.60 of [DZ92]. ■

Definition 4.3.13 [*aliased*] Let R and $\bar{t}$ be a type-annotated term and a tuple of terms respectively, such that for all $x \in \mathit{vars}(R)$, $x\theta$ is a variable, where $\theta = \mathit{mgu}(\bar{t}_R, \bar{t})$, or $\mathit{type}(x, R)$ is an infinite function symbol type. We define the function *aliased* as follows:

$\mathit{aliased}(R, \bar{t}) = \{v \in \mathit{vars}(R) \mid v\theta$ is a variable, and exists $v' \in \mathit{vars}(R)$, $v \neq v'$, such that $v\theta = v'\theta\}$.

■

Definition 4.3.14 [*expansion*] Let R be a type-annotated term, and Cob a cobasic set. Let $mgu(A, B)$ be the most general unifier of the tuples of terms A and B. We define the function *expansion* as follows:

$expansion(R, Cob) = (R', Rest)$, where:

- R' is a type-annotated term;
- $Rest$ is a set of type-annotated terms;
- for all $x \in vars(R')$, it holds that $type(x, R')$ is an infinite function symbol type, or $x\theta$ is a variable, where $\theta = mgu(\bar{t}_{R'}, \bar{t}_{Cob})$;
- $(\cup_{X \in Rest} Den(X)) \cup Den(R') = Den(R)$; and
- for all $X \in Rest$, $X \otimes \bar{t}_{Cob} \simeq \Lambda$.

$(R', Rest)$ is a pair which is a "partition" of R. ■

$\boxed{empty(M, S) :}$

Input: a type-annotated term M and a minset S ($S = A/C$, where A is a tuple of terms, and C a set of cobasic sets).

Output: a boolean value denoting whether $M \otimes S \simeq \Lambda$.

Process:

1. if $S = \Lambda$ then return(**true**), otherwise, let $R = intersection(M, A)$;
2. if $R = \Lambda$ then return(**true**);
3. otherwise, if $included(A, R)$ then return(**false**) else return($empty1(C, R, \emptyset)$).

Figure 4.1: Definition of the function *empty*.

First, it performs the "intersection" of M and the tuple of terms of the minset S (we assume that $S = A/C$). This intersection is implemented by the function *intersection*(R, A), which returns $R \otimes A$ (recall that a tuple of terms is a type-annotated term). Then, if the mentioned intersection (which we call R) is not empty, nor A (recall that $S = A/C$) is "included" in R, it calls *empty1*$(C, R, \emptyset)$, defined in figure 4.2, which checks whether $R/C \simeq \Lambda$. This is done by checking if R is "included" in some tuple of terms of some cobasic set in C (in which case $R/C \simeq \Lambda$). For this purpose, it uses the function *included*$(R, \bar{t}_{Cob})$, where $\bar{t}_{Cob}$ is the tuple of terms of some cobasic set Cob that belongs to C. This function is a straightforward adaptation of the function $subset_T(\tau_1, \tau_2)$ described in [DZ92], that determines whether the type denoted by a pure type term (a variable-free type term) is a subset of the type denoted by another (i.e., *included*$(R, \bar{t})$ returns **true** if and only if $R \sqsubseteq \bar{t}$).

Note that R/C can be seen as a system of one equation (corresponding to R) and zero or more disequations (each of them corresponding to a cobasic set in C). Thus the problem can be seen as determining whether such system has no solutions. We say that a cobasic set Cob is "useless" (for determining the unsatisfiability of the system) whenever if $R/(C - \{Cob\}) \not\simeq \Lambda$, then $R/C \not\simeq \Lambda$. Any useless cobasic set Cob can be removed from C, since $R/(C - \{Cob\}) \simeq \Lambda$ if and only if $R/C \simeq \Lambda$ (note that if $R/(C - \{Cob\}) \simeq \Lambda$, then obviously $R/C \simeq \Lambda$). This is done in step 1 of function *empty1*. If the tuple of terms of a cobasic set Cob in C is "disjoint" with R, then it is useless (however, there can be useless cobasic sets in C whose tuples of terms are not disjoint with R). Once we have removed useless cobasic sets, if the remaining set of cobasic sets is not empty then we go to step 3. In this step, if R is not "included" in none of the tuples of terms of the cobasic sets in C, this means that R is "too big", and thus, it is "expanded" to a set of "smaller" type-annotated terms (with the hope that each of them be "included" in the tuple of terms of some cobasic set in C). This is done in step 4, where a cobasic set Cob of C'' is selected, and R is "expanded" by using the function *expansion*. We have that $expansion(R, Cob) = (R', Rest)$, where R' is an instance of R obtained by expanding R to some "decision depth." This depth allows us to detect if the cobasic set Cob is useless. For example, assume that $R = ((X, Y), (X : list, Y : list))$ and $C = \{C_1, C_2\}$, where $C_1 = comp(([H|L], Z))$ and $C_2 = comp(([\ ], Z))$. R is not included in none of the tuples of terms of the cobasic sets in C, but if we expands R using C_1, i.e., $(R_1, \{R_2\}) = expansion(R, C_1)$, where $R_1 = (([H1|L1], Y1), (H1 : \mu, L1 : list, Y1 : list))$ and $R_2 = (([], Y2), (Y2 : list))$, we have that R_1 and R_2 are included in $\bar{t}_{C_1}$ and $\bar{t}_{C_2}$ respectively (and thus, $R/C \simeq \Lambda$). However, in other situations, the problem cannot be solved by expanding R: assume, for example, that now $C = \{comp((Z, Z))\}$, in this case, R is not included in (Z, Z) because this tuple of terms introduces an equality constraint in $Den(R)$ (note that here R is already expanded to the "decision depth," in which the equality constraints are given by the "aliased variables"). These aliased variables are computed in step 6 by using the function $aliased(R, \bar{t}_{Cob})$.

Also in step 6 (after computing aliased variables), if for some $x \in vars(R')$, it holds that $type(x, R') = \mu$ and either $x \in AVars$, or $x\theta'$ is not a variable, then we can say that Cob useless. This can be proved by using the variable x to construct an instance S of R such that: assuming that there exists an instance I of R, such that $I \otimes\ \bar{t}_{C_1} \simeq \Lambda$ for all $C_1 \in Cset$, where $Cset = C' \cup \{CS \mid (B, A, CS) \in AL\}$, then, S can be constructed from I so that $S \otimes\ \bar{t}_{C_2} \simeq \Lambda$ for all $C_2 \in \{Cob\} \cup Cset$.

The function *empty1*(C, R, AL), defined in figure 4.2, performs a "first round" over the cobasic sets in C. After this round (whose end is detected in step 2 by the condition $C'' = \emptyset$), cobasic sets which have been detected to be useless are ignored (removed) and the rest are stored in AL, which is an accumulation parameter.

In step 7, R' and $AVars$ (besides Cob) are recorded in this parameter, because aliased variables whose type is infinite (or which after having been expanded get bounded to a term containing variables whose type is infinite) allow us to detect useless cobasic sets (which are removed before *empty2*(AL', R) — defined in figure 4.3 — is called in step 2).

The function *empty2*(AL, R), defined in figure 4.3, selects a cobasic set Cob in AL, and, if R is not included in $\bar{t}_{Cob}$, then R is expanded as a set of type-annotated terms $R_1, \ldots, R_n$ by expanding only "decision variables". This ensures that every R_i is either "included" in $\bar{t}_{Cob}$ or "disjoint" with it. It also ensures that R is not infinitely expanded (note that the type of such variables is finite).

Example 4.3.3 Consider the predicate `reverse/2`:

```
reverse(X,Y)  :- X = []  ‖ Y = [].
reverse(X,Y)  :- X = [X1|X2]  ‖ reverse(X2,Y2), append(Y2,[X1],Y).
```

and the type assignment $\rho \equiv (X : list)$, where $list \rightarrow \{[\,], [\mu|list]\}$. Let τ be the input test of the predicate `reverse`/2, i.e., $\tau \equiv X = [\,] \vee X = [X1|X2]$. Let M be the type-annotated term which is a representation of ρ, i.e., $M = ((X), (X : list))$. The minset representations of $X = [\,]$ and $X = [X1|X2]$ are $([\,])$ and $([X1|X2])$ respectively (in this example we deal with unary tuples).

We have that τ covers ρ iff $((X), (X : list)) \sqsubseteq ([\,]) \oplus ([X1|X2])$ iff $((X), (X : list)) \otimes comp(([\,]) \oplus ([X1|X2])) \simeq \Lambda$ iff $((X), (X : list)) \otimes comp(([\,])) \otimes comp(([X1|X2])) \simeq \Lambda$. The disjunctive normal form of $comp(([\,])) \otimes comp(([X1|X2]))$ is $(X3) \otimes comp(([\,])) \otimes comp(([X1|X2]))$, which has only one minset. Now, we have to prove that $((X), (X : list)) \otimes (X3) \otimes comp(([\,])) \otimes comp(([X1|X2])) \simeq \Lambda$, i.e., whether the call *empty*(M, S), where $M = (\bar{t}_M, \rho_M)$, $\bar{t}_M \equiv (X)$, $\rho_M \equiv (X : list)$, and $S \equiv (X3)/\{comp([\,]), comp([X1|X2])\}$ returns **true**. This call proceeds as follows (and in fact returns **true**):

1. *intersection*$(M, (X3))$ returns the type-annotated term $((X4), (X4 : \mathit{list}))$.

2. Since this intersection is not "empty" and $(X3)$—which represents the type-annotated term $((X3), (X3 : \mu))$ —is not "included" in $((X4), (X4 : \mathit{list}))$, the call

   $$\mathit{empty1}(\{\mathit{comp}([\,]), \mathit{comp}([X1|X2])\}, ((X4), (X4 : \mathit{list})), \emptyset)$$

   is performed. This call returns **true** and the computation is as follows:

   (a) We have that $((X4), (X4 : \mathit{list}))$ is not "included" in none of tuples of terms of the cobasic sets in $\{\mathit{comp}([\,]), \mathit{comp}([X1|X2])\}$. Thus, a cobasic set is selected from this set. Assume that $\mathit{comp}([X1|X2])$ is the selected cobasic set;

   (b) $(R', \mathit{Rest}) = \mathit{expansion}(((X4), (X4 : \mathit{list})), \mathit{comp}([X1|X2]))$, where $R' = (([X5|X6]), (X5 : \mu, X6 : \mathit{list}))$, and $\mathit{Rest} = \{(([]), \emptyset)\}$ ($\emptyset$ denotes an empty type assignment, since $([\,])$ has no variables).

   (c) The call $\mathit{included}(R', ([X1|X2]))$ returns **true**, and thus the call $\mathit{empty1}(\{\mathit{comp}([\,])\}, (([\,]), \emptyset), \emptyset)$ is performed. This call also returns **true**, because $(([\,]), \emptyset) \sqsubseteq ([\,])$. Thus, the initial call returns **true**. □

The covering algorithm we present is complete for *tuple-distributive regular types*:

Theorem 4.3.5 *Let M be a type-annotated term in which all types are tuple-distributive regular types, and S a minset. Then empty(M, S) terminates, and returns* **true** *iff $M \otimes S \simeq \Lambda$.*

While sound, the algorithm is not complete for regular types in general (though we believe it is fairly accurate in practice):

Theorem 4.3.6 *Let M be a type-annotated term where all types are regular types, and S a minset. Then empty(M, S) terminates, and if it returns* **true** *then $M \otimes S \simeq \Lambda$.*

One reason for imprecision in the case of non tuple-distributive regular types is that the function *intersection*(M, A), computes a superset of the exact intersection when we deal with general regular types (this result can be derived from the work of Dart and Zobel [DZ92]). Another reason comes from the use of the function *expansion*(R, Cob) to partition the type-annotated term R in the boolean function *empty1*$(C, R, \emptyset)$. Given a pair $(R', Rest)$ where R' is a type-annotated term, and $Rest$ is a set of type-annotated terms, we assume that all type-annotated terms in $Rest$ are disjoint with the tuple of terms of the cobasic set Cob, but this is not true for general regular types, and, consequently, precision may be lost. A possible solution in order to obtain a complete algorithm for general regular types would be to rewrite the type annotated term which represents the input type of a predicate as a union of type annotated terms containing only tuple-distributive types, and then apply the above described covering algorithm for each of the elements of the union.

4.3.2 Covering in Linear Arithmetic over Integers

In this section, we consider linear arithmetic tests over integers (the ideas extend directly to linear tests over the reals, which turn out to be computationally somewhat simpler). Without loss of generality, assume that the tests are in disjunctive normal form, i.e., they are of the form $\Phi(\bar{x}) = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} \phi_{ij}(\bar{x})$ where each of the tests $\phi_{ij}(\bar{x})$ is of the form $\phi_{ij}(\bar{x}) \equiv a_0 + a_1 x_1 + \cdots + a_k x_k \ \textcircled{?}\ 0$, with $\textcircled{?} \in \{=, \neq, <, \leq, >, \geq\}$. Determining whether $\Phi(\bar{x})$ covers the type assignment of `integer` to each variable in $\bar{x}$ amounts to determining whether $\models (\forall \bar{x})\Phi(\bar{x})$. This is true if and only if $(\exists \bar{x})\neg\Phi(\bar{x})$ is unsatisfiable. In other words, we need to determine the unsatisfiability of

$$\neg\Phi(\bar{x}) \ = \ \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \neg\phi_{ij}(\bar{x}) \ = \ \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \psi_{ij}(\bar{x}),$$

where $\psi_{ij}(\bar{x})$ is derived from $\neg\phi_{ij}(\bar{x})$ as follows: let $\phi_{ij}(\bar{x}) = \sum_{i=0}^{k} a_i x_i \text{ \textcircled{?} } 0$. If $\text{\textcircled{?}}$ is a comparison operator other than '=', $\psi_{ij}(\bar{x})$ is simply $\sum_{i=0}^{k} a_i x_i \ \overline{\text{\textcircled{?}}}\ 0$, where $\overline{\text{\textcircled{?}}}$ is the complementary operator to $\text{\textcircled{?}}$, e.g., if $\text{\textcircled{?}} \equiv$ '$>$' then $\overline{\text{\textcircled{?}}} \equiv$ '$\leq$'. If $\text{\textcircled{?}} \equiv$ '=', the corresponding complementary operator is '$\neq$', but this can be written in terms of two tests involving the operators '$>$' and '$<$':

$$\psi_{ij}(\bar{x}) = (\textstyle\sum_{i=0}^{k} a_i x_i > 0) \ \vee \ (\textstyle\sum_{i=0}^{k} a_i x_i < 0).$$

The resulting system, transformed to disjunctive normal form, defines of a set of integer programming problems: the answer to the original covering problem is "yes" if and only if none of these integer programming programs has a solution. Since a test can give rise to at most finitely many integer programs in this way, it follows that the covering problem for linear integer tests is decidable.

Since determining whether an integer programming problem is solvable is NP-complete [GJ79], the following complexity result is immediate:

Theorem 4.3.7 *The covering problem for linear arithmetic tests over the integers is co-NP-hard.*

It should be noted, however, that the vast majority of arithmetic tests encountered in practice tend to be fairly simple: our experience has been that tests involving more than two variables are rare. The solvability of integer programs in the case where each inequality involves at most two variables, i.e., is of the form $ax + by \leq c$, can be decided efficiently in polynomial time by examining the loops in a graph constructed from the inequalities [AS79]. The integer programming problems that arise in practice, in the context of covering analysis, are therefore efficiently decidable.

4.3.3 Covering Analysis: Putting it Together

Let τ be the input test of predicate p and ρ a type assignment. Consider the type assignment ρ written as a type-annotated term M, and τ written in disjunctive normal form, i.e., $\tau = \tau_1 \vee \cdots \vee \tau_n$, where each τ_i is a conjunction of primitive tests (recall that primitive tests are unification, disunification, etc.). Consider the test τ_i written as $\tau_i^H \wedge \tau_i^A$, where τ_i^H and τ_i^A are a conjunction of primitive unification and arithmetic tests respectively (i.e., we write arithmetic tests after unification tests). Consider also τ_i^H written as a minset D_i (recall that D_i is the intersection (conjunction) of a tuple of terms, and zero or more cobasic sets). Let D be the union (disjunction) of the these minsets.

Example 4.3.4 Let p be the predicate `partition`/4 from the familiar quicksort program. Let τ be $X = [\,] \vee (X = [H|L] \wedge H > Y) \vee (X = [H|L] \wedge H \leq Y)$ and let ρ be $(X : \mathit{intlist}, Y : \mathit{integer})$, where $\mathit{intlist} \rightarrow \{[\,], [\mathit{integer}|\mathit{intlist}]\}$. In this case, we have that M is $((X,Y),(X : \mathit{intlist}, Y : \mathit{integer}))$. $\tau_1 \equiv X = [\,]$, $\tau_2 \equiv X = [H|L], H > Y$, and $\tau_3 = X = [H|L], H \leq Y$. τ_1 can be written as $\tau_1^H \wedge \tau_1^A$, where $\tau_1^H \equiv X = [\,]$ and $\tau_1^A \equiv \mathit{true}$. Similarly, $\tau_2^H \equiv X = [H|L]$ and $\tau_2^A \equiv H > Y$, and $\tau_3^H \equiv X = [H|L]$ and $\tau_3^A \equiv H \leq Y$. $D = D_1 \oplus D_2 \oplus D_3$, where $D_1 \equiv ([\,], Y)$, $D_2 \equiv ([H|L], Y)$ and $D_3 \equiv ([H|L], Y)$. $\Box$

To test whether τ covers ρ, we first test that D covers M, ignoring the arithmetic tests. If D does not cover M, then obviously, the (whole) input test of p, τ, does not cover M, and we report failure. Otherwise, we create (zero or more) covering subproblems, each of them containing only arithmetic tests, as follows:

1. Let A be the set of all the tuples of terms and negations of cobasic sets appearing in D (note that the negation of a cobasic set is a tuple of terms, thus A is a set of tuples of terms), and let $A' = \{b \in A \mid M \otimes b \not\simeq \Lambda\}$.

2. For each tuple of terms b in A':

(a) Let $I = M \otimes b$ and $\theta = mgu(\bar{t}_M, \bar{t}_I)$;

(b) Let $\tau_b = \bigvee_{j=1}^{m} r_j$, where $\{r_1, \ldots, r_m\} = \{t_i \mid b \sqsubseteq D_i$ for some $1 \leq i \leq n$ and t_i is the result of applying θ to τ_i^A (this is done to take into account possible variable aliasing)$\}$. Note that there is an algorithm to test whether $b \sqsubseteq D_i$ in [Kun87].

(c) Test whether τ_b covers ρ_I (recall that ρ_I refers to the type assignment of I):

i. Assume that $\tau_b = s_1 \vee \cdots \vee s_n$ and each s_i is a conjunction of primitive arithmetic tests. If $\tau_b \equiv$ **true** then report success;

ii. otherwise, if for some variable x appearing in all s_i, $1 \leq i \leq n$, it holds that $type(x, \rho_I)$ is not a numeric type, then report failure;

iii. otherwise, use the algorithm described in section 4.3.2 to test whether τ_b covers ρ_I.

Note: another way to create the subproblems is: $\tau_b = \bigvee_{j=1}^{m} r_j$, where $\{r_1, \ldots, r_m\} = \{t_i \mid I \sqsubseteq D_i$ for some $1 \leq i \leq n$ and t_i is the result of applying θ to $\tau_i^A\}$. This algorithm is more precise than the former, but is more complex because I is a type-annotated term and thus we have to use the covering algorithm described in Section 4.3.1 to test that $I \sqsubseteq D_i$.

Theorem 4.3.8 *If D covers M and for each $b \in A'$, τ_b covers I, then the input test of p, τ, covers M.*

Proof It is clear that if D covers M, then the disjunction of all the tuples of terms in A' also covers M. Thus, for any tuple of terms $\bar{x}$ which is an instance of M, there is at least a $b \in A'$, such that $\bar{x}$ is an instance of b, and all the tests τ_i^H such that $b \sqsubseteq D_i$, will succeed for $\bar{x}$. If τ_b covers I, then at least one of the tests t_i in τ_b will succeed for $\bar{x}$. Thus, by the construction of τ_b, at least one τ_i will succeed for $\bar{x}$, and we conclude that τ covers M. ∎

Example 4.3.5 Consider Example 4.3.4. It is clear that D covers M, thus we proceed as follows:

1. $A = \{([\,], Y), ([H|L], Y)\}$, and $A' = A$.

2. Let $b1 = ([], Y)$ and $b2 = ([H|L])$. Then $\tau_{b1} \equiv$ **true** and $\tau_{b2} \equiv H > Y \vee H \leq Y$.

3. We have that **true** covers $(([], Y), (Y : \mathit{integer}))$, and also that $H > Y \vee H \leq Y$ covers $(L : \mathit{intlist}, H : \mathit{integer}, Y : \mathit{integer})$, thus τ covers M. □

Note that the former approach can be also used to partition a problem into a Herbrand covering subproblem (unification/disunification tests) and zero or more subproblems of any type. In this case, we would use the appropriate algorithm to solve each of the resulting covering subproblems.

4.4 Non-Failure Analysis

4.4.1 The Analysis Algorithm

Once we have determined which predicates cover their types, determining non-failure is straightforward: from Theorem 4.3.1, analysis of non-failure reduces to the determination of reachability in the call graph of the program. In other words, a predicate p is non-failing if there is no path in the call graph of the program from p to any predicate q that does not cover its type. It is straightforward to propagate this reachability information in a single traversal of the call graph in reverse topological order. The idea can be illustrated by the following example.

Example 4.4.1 Consider the following predicate taken from a quicksort program:

```
  qs(X1,X2)  :− X1 = []  ‖ X2 = [].
  qs(X1,X2)  :− X1 = [H|L]  ‖ part(H,L,Sm,Lg),
     qs(Sm,Sm1), qs(Lg,Lg1), app(Sm1,[H|Lg1],X2).
```

Suppose that qs/2 has mode (in, out) and type (intlist, -), and suppose we have already shown that part/4 and app/3 cover the types (int, intlist, -, -) and (intlist, intlist, -) induced for their body literals in the recursive clause above. The input test for qs/2 is X1 = [] ∨ X1 = [H|L], and this covers the type intlist, which means that head unification will not fail for qs/2. It follows that a call to qs/2 with the first argument bound to a list of integers will not fail. □

The accuracy of the described algorithm can be improved by detecting situations as illustraded in example 4.4.2.

Example 4.4.2 Consider the following partition program, and suppose that partition/4, less/2 and greatereq/2 have mode partition(in, in, out, out), less(in, in) and greatereq(in, in), and type partition(intlist, integer, -, -), less(integer, integer) and greatereq(integer, integer) respectively.

```
partition([E|R],C,[E|Left1],Right)  :−
       less(E, C),
       partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1])  :−
       greatereq(E, C),
       partition(R,C,Left,Right1).
less(X, Y)  :− X < Y.
greatereq(X, Y)  :− X >= Y.
```

Since user defined predicates `less/2` and `greatereq/2` do not cover their types, they are marked as *possibly failing*. This information is propagated through the call graph of the program, so that predicate `partition/4` is also marked as *possibly failing*. However, `partition/4` does not fail for the modes and types described. This drawback in accuracy is due to the fact that the tests of `less/2` and `greatereq/2` are "hidden" when the covering problem for predicate `partition/4` is set up. In order to avoid this situation, we can, make "visible" the tests of `less/2` and `greatereq/2` when the covering problem is set up.

□

4.4.2 A Prototype Implementation

In order to evaluate the effectiveness and efficiency of our approach to non-failure analysis we have constructed a relatively complete prototype which performs such analysis in an automatic way. This prototype has been integrated in *CiaoPP* [HBPLG99, HBC$^+$99]. The system takes Prolog programs as input, which include a module definition in the standard way. In addition, the types and modes of the arguments of exported predicates are given, as well as the required type definitions. The system uses the PLAI analyzer (also integrated in *CiaoPP*) to derive mode information, using the Sharing+Freeness domain [MH91], and an adaptation of Gallagher's analysis to derive the types of predicates [GdW94], and also to deal with parametric types. The resulting type- and mode-annotated programs are analyzed using the algorithms presented for Herbrand and linear arithmetic tests.

Herbrand covering is checked by a naive direct implementation of the analyses presented. Testing of covering for linear arithmetic tests is implemented directly using the Omega test [Pug92]. This test determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities, referred to as a problem.

We have tested the prototype first on a number of simple standard benchmarks, and then on more complex ones. The latter are taken from those used in the cardinality analysis of Braem *et al.* [BCMH94], which is the closest related previous work that we are aware of. Some relevant results of these tests are presented in Table 4.1. **Program** lists the program names, **N** the number of predicates in the program, **F** the number of predicates detected by the analysis as non-failing, **Cov** the number of predicates detected to cover their type, **C** the number of non-failing predicates detected in [BCMH94], $\mathbf{T}_F$ the time required by the non-failure analysis (SPARCstation 10, 55MHz, 64Mbytes of memory), $\mathbf{T}_M$ the time required to derive the modes and types, and $\mathbf{T}_T$ the total analysis time (all times are given in milliseconds). Averages (per predicate in the case of analysis time) are also provided in the last row of the table.

The results are quite encouraging showing that the developed analysis is fairly accurate. The analysis is significantly more powerful than those previously reported in non-failure detection (the experimental results presented in [BCMH94] suggest that it is more appropriate for detecting determinacy than for non-failure). It is pointed out in [BCMH94] that the sure success information can be improved by using a more sophisticated type domain. However, this is also applicable to our analysis, and the types inferred by our system are similar to those used in [BCMH94]. Much of the power of our algorithm comes from the use of the notion of covering, which allows detecting when at least one of the clauses (not necessarily the same) defining a predicate will not fail for all possible calls. The cardinality analysis detects non-failure only when at least one of the clauses (always the same) defining a predicate will not fail for all the possible calls. The non-failure analysis times are quite good, despite the currently naive implementation of the system (for example, the call to the omega test is done by calling an external process). The overall analysis times are quite acceptable, even when including the type and mode analysis times, which are in any case very useful in other parts of the compilation process.

The Mercury system [HSC96] allows the programmer to declare that a predicate will produce at least one solution, and attempts to verify this with respect to the Herbrand terms with equality tests. As far as we know, the Mercury compiler does not handle disequality constraints on the Herbrand domain. Nor does it handle arithmetic tests, except in the context of the if-then-else construct. As such, it is considerably weaker than the approach described here.

4.5 Applications

There are several applications of this analysis. The first application is implementing granularity control in parallelizing compilers, which is the main motivation for our non-failure analysis. We refer the reader to Chapter 2 where this kind of application is described in detail, and also to Chapter 3, where an analysis for estimating lower bounds to the cost of goals and procedures is described.

The second application has to do again with (and-)parallelism, in particular with the avoidance of speculative computation. Consider a number of goals in a resolvent which are determined to be independent. As shown in [HR95], and ignoring parallelization overheads (which can be dealt with as illustrated above), the time involved in their parallel execution can be guaranteed to be smaller or equal to that of the corresponding sequential execution. However, it is impossible to guarantee that no more work will be performed. This is due to the possibility of failure of one of the goals. Consider two goals p and q so that q is executed after p in the sequential execution. Assume also that p fails (both in the sequential and, correspondingly, in the parallel execution). If p and q are scheduled for execution in parallel, a part of q may be executed until the point in which p fails

Program	**N**	**F (%)**	**Cov (%)**	**C**	$\mathbf{T}_F$	$\mathbf{T}_M$	$\mathbf{T}_T$
Hanoi	2	2 (100)	2 (100)	N/A	60	860	920
Deriv	1	1 (100)	1 (100)	N/A	80	940	1,020
Fib	1	1 (100)	1 (100)	N/A	20	90	110
Mmatrix	3	3 (100)	3 (100)	N/A	90	350	440
Tak	1	1 (100)	1 (100)	N/A	10	110	120
Subs	1	1 (100)	1 (100)	N/A	50	90	140
Reverse	2	2 (100)	2 (100)	N/A	10	100	110
Qsort	3	3 (100)	3 (100)	0 (0)	80	440	520
Qsort2	5	3 (60)	3 (60)	0 (0)	100	390	490
Queens	5	2 (40)	2 (40)	0 (0)	120	360	480
Gabriel	20	3 (15)	10 (50)	0 (0)	420	1,860	2,280
Read	38	8 (21)	19 (50)	8 (21)	540	12,240	12,780
Kalah	44	18 (40)	29 (65)	6 (13)	1,500	14,570	16,070
Plan	16	4 (25)	11 (68)	0 (0)	810	7,000	7,810
Credit	25	10 (40)	18 (72)	0 (0)	4,720	1,470	6,190
Pg	10	2 (20)	6 (60)	0 (0)	540	1,600	2,140
Mean	–	36%	63%	3%	51 (/p)	239 (/p)	291 (/p)

Table 4.1: Accuracy and efficiency of the non-failure analysis (times in mS).

(the execution of q will normally be killed at this point). Although not producing a slow-down, this constitutes unnecessary computation which steals computing resources from any useful work that may exist in the system (and therefore does reduce speedup). Determining that goals in a conjunction will not fail (at least all but the rightmost one – note that failure of q in the example above does not have these ill-effects) thus allows guaranteeing avoidance of speculative computation.

A third application is in the general area of program transformation, where information about non-failure can be used in determining the order of execution of literals in a clause. Consider a clause

$$H :- B_1, p(X), B_2, q(X), B_3$$

where B_1, B_2, B_3 are sequences of literals, $p(X)$ produces bindings for X, and $q(X)$ is the left-most body goal that has X as an input argument. If p is known to be non-failing, it may be possible to transform this clause to

$$H :- B_1, B_2, p(X), q(X), B_3.$$

The resulting code may be more efficient than the original if a goal in B_2 can fail.

Finally, among the most important applications of non-failure we envision is in speeding up program development by assisting programmers by reporting predicates that are not guaranteed to not fail. This can help in detecting programming errors at compile time, in much the same way as type checking does in statically typed languages, since in logic programs the usual expectation is that a predicate will succeed and produce one or more solutions. In most logic programming systems, however, little compile-time checking is performed. The system is currently integrated in the *CiaoPP* system and used for these purposes (as well as for optimization).

4.6 Chapter Conclusions

We have provided a method whereby, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed not to fail (i.e., to produce at least one solution or not terminate). The technique is based on an intuitively very simple notion, that of a (set of) tests “covering” the type of a set of variables. We have given sound and complete algorithms for determining covering that are precise and efficient in practice. We have commented on applications of non-failure analysis, such as for example, estimating lower bounds on the computational costs of goals (used for granularity control); avoiding speculative parallelism; programming error detection, and program transformation. We have implemented (and integrated in the *CiaoPP* system) our non-failure analysis and shown that better results are obtained than with previously proposed approaches.

$\boxed{\textit{empty1}(C, R, AL) :}$

Input: a type-annotated term R, a set of cobasic sets C, and, a set AL of triples of the form (B, AV, CS) where:

- B is a type-annotated term,
- CS is a cobasic set,
- $vars(B) \cap vars(CS) = \emptyset$,
- For all $x \in vars(B)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_B, \bar{t}_{CS})$, and,
- $v \in AV$ iff $v \in vars(B)$ and exists $v' \in vars(B)$, $v \neq v'$, such that $v\theta = v'\theta$ (i.e., AV is the set of variables in $vars(B)$ which are aliased with some other variable in $vars(B)$ by θ).

Output: the boolean value **true** if $R/C_1 \simeq \Lambda$, where $C_1 = C \cup \{Cob \mid (B, A, Cob) \in AL,\ for\ some\ B\ and\ A\}$. Otherwise, returns **false**.

Process:

1. Let $C'' = \{Cob \in C \mid \ \textit{intersection}(R, \bar{t}_{Cob}) \not\simeq \Lambda\}$;
2. If $C'' = \emptyset$ then return($\textit{empty2}(AL', R)$), where $AL' = \{(S, AVars, Cob) \mid (S, AVars, Cob) \in AL, \textit{intersection}(R, \bar{t}_{Cob}) \not\simeq \Lambda, \theta = mgu(\bar{t}_S, \bar{t}_R)$, and for all x, y such that $x \in AVars$ and $y \in vars(x\theta)$, $\textit{type}(y, R)$ is finite (there are straightforward algorithms to test whether a type expression denotes an infinite or finite set of terms) $\}$.
3. Otherwise, if $\textit{included}(R, \bar{t}_{Co})$ for some cobasic set Co in C'' then return(**true**);
4. Otherwise, take a cobasic set Cob of C'', and let $C' = C'' - \{Cob\}$ and $(R', Rest) = \textit{expansion}(R, Cob)$;
5. If $\textit{included}(R', \bar{t}_{Cob})$ then return($\bigwedge_{X \in Rest} \textit{empty1}(C', X, AL)$);
6. Otherwise, let $AVars = \textit{aliased}(R', \bar{t}_{Cob})$. If for some $x \in vars(R')$, it holds that $\textit{type}(x, R')$ is an infinite function symbol type, and $x \in AVars$ or $x\theta'$ is not a variable, where $\theta' = mgu(\bar{t}_{R'}, \bar{t}_{Cob})$, then return($\textit{empty1}(C', R, AL)$);
7. Otherwise, let $AL' = AL \cup \{(R', AVars, Cob)\}$;
8. return($\textit{empty1}(C', R', AL') \wedge (\bigwedge_{X \in Rest} \textit{empty1}(C', X, AL))$);

Figure 4.2: Definition of the function *empty1*.

empty2(AL, R):

Input:

- a type-annotated term R,
- a set AL of triples of the form (B, AV, CS) where:
  - B is a type-annotated term,
  - CS is a cobasic set,
  - $vars(B) \cap vars(CS) = \emptyset$,
  - for all $x \in vars(B)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_B, \bar{t}_{CS})$ ($type(x, B)$ can be any type, including an infinite function symbol type),
  - $v \in AV$ iff $v \in vars(B)$ and exists $v' \in vars(B)$, $v \neq v'$, such that $v\theta = v'\theta$ (i.e., AV is the set of variables in $vars(B)$ which are aliased with some other variable in $vars(B)$ by θ), and
  - for all $v \in AV$, $type(v, \rho_B)$ is finite.

Output: a boolean value, **true** if $R/C \simeq \Lambda$, where $C = \{CS \mid (B, AV, CS) \in AL\ for\ some\ B\ and\ AV\}$ (i.e., C is the set of cobasic sets in AL), **false** otherwise.

1. If $AL = \emptyset$ then return(**false**); otherwise, take an item $A \in AL$. Assume that $A \equiv (B, AV, Cob)$, and let $AL' = AL - \{A\}$ and $\theta = mgu(\bar{t}_B, \bar{t}_R)$;
2. if $included(R, \bar{t}_{Cob})$ then return(**true**);
3. otherwise, for all variables $y \in AV$, expand all variables x such that $x \in vars(y\theta)$ (necessarily $x \in vars(R)$ and $type(x, R)$ is finite). Let RS be the set of type-annotated terms resulting from these expansions.
4. Let $RS' = \{r \in RS \mid intersection(r, \bar{t}_{Cob}) \simeq \Lambda\}$ (necessarily for all $s \in RS$ and $s \notin RS'$, it holds that $s \sqsubseteq \bar{t}_{Cob}$);
5. if $RS' = \emptyset$ then return(**true**);
6. otherwise return($\bigwedge_{X \in RS'}$ *empty2*(AL', X)).

Figure 4.3: Definition of the function *empty2*.

Chapter 5

Efficient Term Size Computation

As mentioned in Chapter 2, 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. By term size we refer to measures such as list length, term depth, number of nodes in a term, etc.

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.

Our objective 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. In this Chapter 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 [HLG94]. We also discuss the advantages and applications of our technique (specifically in the task of granularity control) and present some performance results.

5.1 Overview of the Approach

As mentioned at the beginning of this chapter, 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:[1]

[1] 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

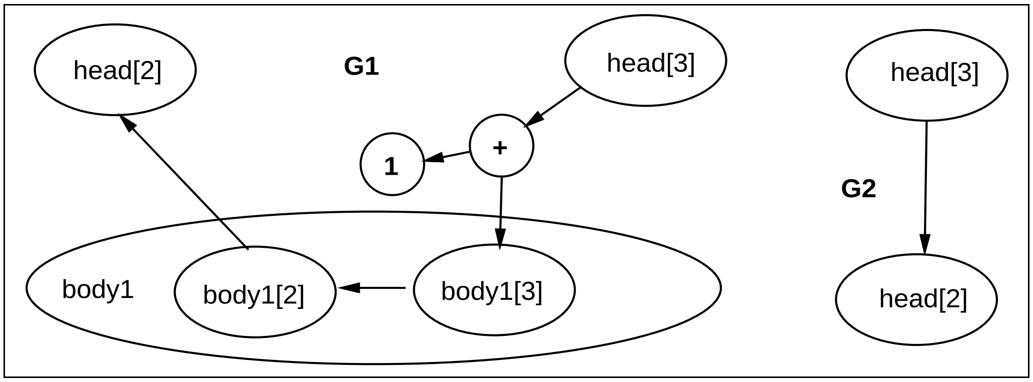

Figure 5.1: Size dependency graphs for predicate `append`/3.

```
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 5.1 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 in a search tree which will have to be explored with the objective

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.

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,[2] 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 [UV88, VS92, DLH90]. In particular, we refer to the size relations described in [DL93]. Consider the function $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ (as defined in [DL93]), 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 [DLH90], 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 [VS92]. 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

[2] 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.

head[3] is equal to the size of the term appearing at position $body_1[3]$ plus one, i.e. $\mid$ `[H|R]` $|_{\mathrm{list\_length}} = \mid$ `R` $|_{\mathrm{list\_length}} + 1$ holds for every substitution that makes `H` and `R` ground.

5.2 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 5.1; **b)** A binary *arithmetic operator* (+, −, etc.); or **c)** A non-negative integer *number*.

We distinguish two classes of edges:

- *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 5.1. 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 5.1. $head[2]$ 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, $(\mathit{append}/3, \{2\}, \{3\})$ 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 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 5.2.1 Figure 5.1 represents the transformation node composed by the size dependency graphs `G1` and `G2`, namely $((\mathit{append}/3, \{2\}, \{3\}), \{\texttt{G1, G2}\})$. □
□

We require that the size dependency graphs meet the following condition: if there is an inter-literal edge from a supplied size node $\mathit{body}_i[k]$ to a computed size node $\mathit{body}_j[n]$ 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 [JL87] which can solve linear equations. However actual equation solving would probably incur in significant overhead. Thus we enforce the condition both for efficiency reasons

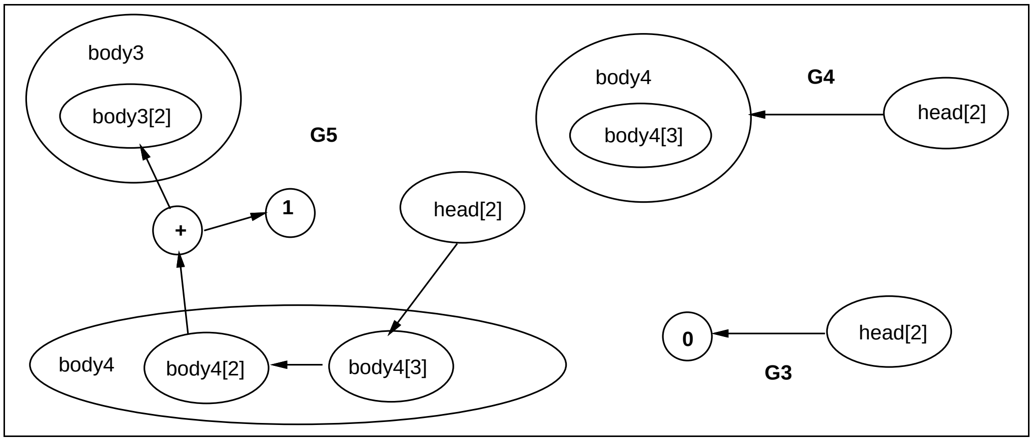

Figure 5.2: Size dependency graphs for predicate `qsort`/2.

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 5.1. 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.

5.3 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 5.3.1 [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. □

■

Definition 5.3.2 [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. □ ■

Definition 5.3.3 [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 $(Pred, Is, Os)$. □ ■

Theorem 5.3.1 *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 $Pred$).*□

5.4 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 5.4.1 [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 5.4.2 [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 5.3.3 without ambiguity). Thus, all edges are omitted.

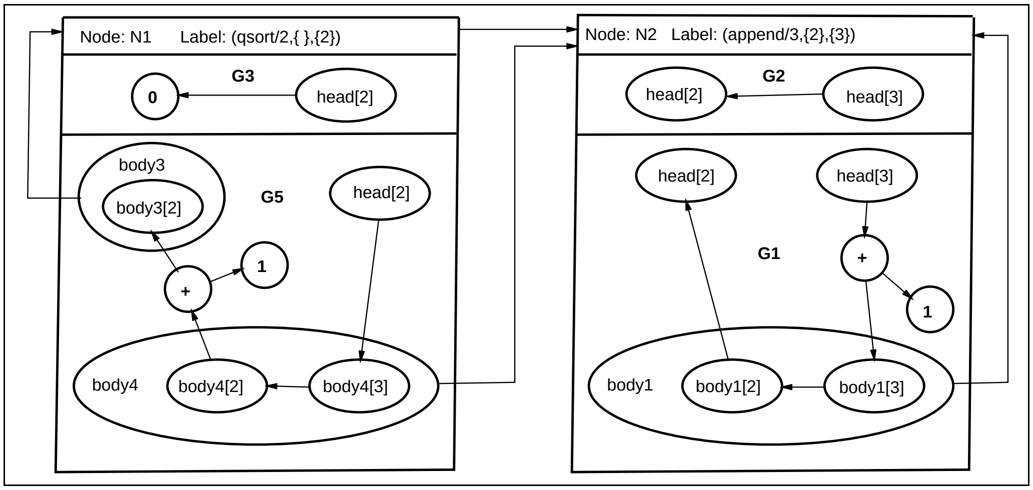

Figure 5.3: An irreducible transformation.

Example 5.4.1 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 5.2 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 5.2.1. 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 5.3. 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

[Deb89, MH92, Bru91] and an argument size analysis [DLH90]. 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 two first arguments), and the intra-literal size relation $size_2 = size_1$ for predicate `qsort`/2. Consider the clause `C2` in Example 5.4.1. Using $size_3 = size_1 + size_2$ for `append`/3 we have that $|\texttt{L2}|_{\text{list\_length}} = |\texttt{Ls2}|_{\text{list\_length}} + |[\texttt{First}|\texttt{Lg2}]|_{\text{list\_length}}$ holds for every substitution that makes all the terms appearing in it ground, and also $|\texttt{L2}|_{\text{list\_length}} = |\texttt{Ls2}|_{\text{list\_length}} + |\texttt{Lg2}|_{\text{list\_length}} + 1$ 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 $|\texttt{Ls2}|_{\text{list\_length}} = |\texttt{Ls}|_{\text{list\_length}}$ and $|\texttt{Lg2}|_{\text{list\_length}} = |\texttt{Lg}|_{\text{list\_length}}$ 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 5.4.1 *If there is a transformation T for a size computation specification X then there is an irreducible transformation T' for X.*□

Theorem 5.4.1 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.

5.5 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 [DLH90] (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 5.4 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 5.5.1 Consider the predicate `qsort`/2 as defined in Example 5.4.1, 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 5.2, 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 5.2, 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 5.1, 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 [PP94] 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 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.

5.6 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

bench	$\mathbf{T_{wsc}}$	$\mathbf{T_{st}}$	$\mathbf{T_{pt}}$	$\mathbf{T_{st}} - \mathbf{T_{wsc}}$	$\mathbf{T_{pt}} - \mathbf{T_{wsc}}$	**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 5.1: Execution times (ms) for benchmarks.

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 5.1 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. $\mathbf{T_{st}} - \mathbf{T_{wsc}}$ and $\mathbf{T_{pt}} - \mathbf{T_{wsc}}$ 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: $gain = \frac{(\mathbf{T_{st}}-\mathbf{T_{wsc}})-(\mathbf{T_2}-\mathbf{T_{wsc}})}{\mathbf{T_{st}}-\mathbf{T_{wsc}}}100$ For brevity only a brief description of the benchmarks is provided. A more complete description (including the program text) can be found in [HLG94]. 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.

5.7 Chapter Conclusions

We have developed a technique for transforming predicates so that they compute some of their argument data sizes at run-time, in addition to performing their normal computation. We have presented 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 developed and integrated in the `ciaopp` system an algorithm for finding irreducible transformations. We also have discussed the advantages and applications of our technique (specifically in the task of granularity control) and presented some performance results, which we believe are quite encouraging. The observed gain (with respect to to what we will call the "standard approach" to computing term sizes) arise from two factors: avoiding additional term traversal and performing less arithmetic operations. Another advantage of our approach is that it can take profit of previous size computations so that no recomputation is performed. Among the applications for which our technique for dynamic term size computation is useful, we can mention applications related to program optimization which includes recursion elimination, and selection among different algorithms or control rules whose performance may be dependent on such size.

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 5.4: A top-down algorithm for finding irreducible transformations.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have developed (an integrated in the `ciaopp` system [HBPLG99, HBC+99]) a complete automatic granularity control system for logic programs based on a program analysis and transformation schema, where as much work is done at compile time. For this purpose we have developed some program transformation techniques, so that transformed programs perform an efficient granularity control at runtime, and also have developed some program analysis techniques able to infer the information needed for the program transformation phase, such as (lower bounds on) cost of procedures, which calls will not fail (non-failure analysis), etc.

We have discussed the many problems that arise (for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model) and provided solutions to them. We believe that the results are general enough to be of interest to researchers working on other parallel languages.

We know of no other work which describes (and implements) a complete and fully automatic granularity control system for logic programs, discusses the many problems that arise and provides solutions to them in the generality that we present our work.

We have also assessed the developed granularity control techniques for and-parallelism and or-parallelism on the &-Prolog and Muse systems respectively, and have obtained what we believe are quite encouraging results.

It appears from the sensitivity of the results that we have observed in our experiments 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 control can usefully be performed automatically by a compiler.

We can conclude that granularity analysis/control is a particularly promising technique because it has the potential of making feasible to automatically exploit low-cost parallel architectures, such as workstations on a (possibly high speed) local area network.

Finaly, it is worth of mentioning that some of the techniques that we have developed for granularity control, have other important applications. In particular, non-failure information is very useful for the avoidance of speculative parallelism, program error detection, and program transformation. The technique for dynamic term size computation "on the fly" is useful for applications related to program optimization which includes recursion elimination, and selection among different algorithms or control rules whose performance may be dependent on such size. Information about the computational cost of a program is potentially useful for a variety of purposes. Programmers can use such information to choose between different algorithmic solutions to a problem. Program transformation systems can use cost information to choose between alternative transformations. Information about message sizes and relative frequency of communication between different processes can be used to improve task mapping decisions on distributed memory systems. Information about the number of solutions in deductive database systems can be used for query optimization purposes. Apart from these applications of cost information, the problem of cost analysis may be of some independent interest to researchers on static analysis of logic programs because (*i*) it uses a great deal of information from other kinds of analyses, such as mode and type analysis, inference of size norms, etc., so that any improvements in these analyses potentially yield improvements in cost analysis; and (*ii*) because of the rich variety of algorithms for combinatorial analysis that arise, especially when dealing with constraints.

6.2 Future Work

There are many directions in which the work described in this thesis can be extended. We intend to study task assignment techniques based on the combination of granularity information and scheduling techniques such as those used in data parallelism. We can also combine our techniques with Partial Evaluation Techniques in order to create coarse granularity tasks (e.g. by means of unfolding operations, loop unrolling, etc.). We also have in mind extending the techniques we have developed to concurrent constraint languages.

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

Finally, we also are planning to perform an assesment of our granularity control system with lager and more real programs, and apply the developed tools to the version of parallel ECL^{i}PS^e running on a network of workstations and the distributed version of *Ciao*–Prolog.

Bibliography

[AK90] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[AR94] L. Araujo and J.J. Ruz. PDP: Prolog Distributed Processor for Independent_AND/OR Parallel Execution of Prolog. In *Eleventh International Conference on Logic Programming*, pages 142–156, Cambridge, MA, June 1994. MIT Press.

[AR97] L. Araujo and J.J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, October 1997.

[AS79] B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. In *Proc. 20th ACM Symposium on Foundations of Computer Science*, pages 205–217, October 1979.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BCMH94] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on*

Logic Programming, pages 457–471, Ithaca, NY, November 1994. MIT Press.

[BG96] D. Basin and H. Ganzinger. Complexity Analysis based on Ordered Resolution. In *11th. IEEE Symposium on Logic in Computer Science*, 1996.

[BH89] B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.

[BK96] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *Proc. 6th International Workshop on Logic Program Synthesis and Transformation*, pages 34–153. Stockholm University/Royal Intitute of Technology, 1996.

[BR86] P. Borgwardt and D. Rea. Distributed Semi-Intelligent Backtracking for a Stack-Based AND-Parallel Prolog. In *Symp. on Logic Prog.*, pages 211–222, 1986.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[BSY88] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

[CC94] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[CH83] A. Ciepielewski and S. Haridi. A formal model for or-parallel execution of logic programs. In Mason, editor, *IFIP 83*, 1983.

[CL89] H. Comon and P. Lescanne. Equational Problems and Disunification. *J. Symbolic Computation*, 7(3/4):371–425, 1989.

[Col87] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[Deb89] 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.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DL90] S.K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.

[DL93] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[DLGH95] S.K. Debray, P. López-García, and M. Hermenegildo. Towards Cost Analysis of Divide-and-Conquer Logic Programs. Technical Report CLIP18/95.0, Facultad Informática UPM, Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, March 1995.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLH90] 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.

[DZ92] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[Fag87] B. S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-Case Analysis of Algorithms. *Theor. Comp. Sci.*, (79):37–109, 1991.

[Gal97] M. M. Gallardo. *Análisis de Lenguajes Lógicos Concurrentes Mediante Interpretación Abstracta*. PhD thesis, University of Málaga, November 1997.

[GDL95] R. Giacobazzi, S.K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GH85] B. Goldberg and P. Hudak. Serial Combinators: Optimal Grains of Parallelism. In *Proc. Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 382–399. Springer-Verlag, Aug 1985.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc.*

3rd. Int'l Symposium on Programming Language Implementation and Logic Programming, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[GT94] M. M. Gallardo and J. M. Troya. Granularity Analysis of Concurrent Logic Languages based on Abstract Interpretation. In Alpuente, Barbuti, and Ramos, editors, *GULP-ProDe'94*, volume 2, pages 342–356. Universidad Politécnica de Valencia, September 1994.

[GT95] M. M. Gallardo and J. M. Troya. Studying the Cost of Logic Languages in an Abstract Interpretation Framework for Granularity Analysis. In Logic Program Synthesis and Transformation. *Pre-Proceedings of LOPSTR'95*, Utrecht, Netherlands, September 1995. Extended version as Technical Report CW 216, K.U. Leuven.

[Hau90] B. Hausman. Handling speculative work in or-parallel prolog: Evaluation results. In *North American Conference on Logic Programming*, pages 721–736, Austin, TX, October 1990.

[HBC$^+$99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[Her86] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, U. of Texas at Austin, August 1986.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HLA94] 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.

[HLG94] 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.

[HR95] 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.

[HSC96] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the mercury compiler. In *Proc. Australian Computer Science Conference*, Melbourne, Australia, January 1996.

[Hua85] C.H. Huang. An interpreter of restricted and-parallelism for prolog programs. MCC AI Note, 1985.

[Hue93] L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JM81] Neil D. Jones and Steven S. Muchnick. *Program Flow Analysis: Theory and Applications*, chapter Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra, pages 380–393. Prentice-Hall, 1981.

[Kal87] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[Kap88] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[Kar72] R. M. Karp. Reducibility among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KL88] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[Kow74] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[Kow79] Robert A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., 1979.

[Kow80] R. A. Kowalski. Logic as a computer language. *Proc. Infotec State of the Art Conference, Software Development: Management*, June 1980.

[KSB97] A. King, K. Shen, and F. Benoy. Lower-bound Time-complexity Analysis of Logic Programs. In *1997 International Logic Programming Symposium*, pages 261–275. MIT Press, Cambridge, MA, October 1997.

[Kun87] K. Kunen. Answer Sets and Negation as Failure. In *Proc. of the Fourth International Conference on Logic Programming*, pages 219–228, Melbourne, May 1987. MIT Press.

[LGH93] P. López-García and M. Hermenegildo. Towards Dynamic Term Size Computation via Program Transformation. In *Second Spanish Conference on Declarative Programming*, pages 73–93, Blanes, Spain, September 1993. IIIA/CSIC.

[LGH95] P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[LGHD94] 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.

[LGHD96] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[LH85] G. J. Lipovski and M. Hermenegildo. B-LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs. In *1985 IEEE International Conference on Parallel Processing*, pages 560–568. IEEE Computer Society, August 1985.

[Lin88] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[Lin93] N.-W. Lin. Approximating the Chromatic Polynomial of a Graph. In *Proc. Nineteenth International Workshop on Graph-Theoretic Concepts in Computer Science*, June 1993.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[LM87] J.-L. Lassez and K. Marriott. Explicit Representation of Terms Defined by Counter Examples. *Journal of Automated Reasoning*, 1987.

[LMM88] J.-L. Lassez, M. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–626. Morgan Kaufman, 1988.

[LMM91] J.-L. Lassez, M. Maher, and K. Marriott. Elimination of Negation in Term Algebras. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science*, number 520 in Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1991.

[Lus88] E. Lusk et al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[Mat70] J. V. Matijasevič. Enumerable sets are diophantine. 191:279–282, 1970. English translation in *Soviet Mathematics—Doklady*, 11 (1970), 354-357.

[Met88] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[MG89] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] 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.

[PK88] V. Penner and A. Klinger. AND-Parallel PROLOG on Large Scale Transputer-Systems. Internal report, Institute for Applied Mathematics at the Aachen Technical University, West Germany, 1988.

[PP94] 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.

[Pug92] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[PW93] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. Technical report, Dept. of Computer Science, Univ. of Maryland, December 1993.

[RM90] 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.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Ros89] M. Rosendhal. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London, (1989).

[SCK98] K. Shen, V.S. Costa, and A. King. Distance: a New Metric for Controlling Granularity for Parallel Execution. In Joxan Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 85–99, Cambridge, MA, June 1998. MIT Press, Cambridge, MA.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[UV88] J.D. Ullman and A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal ACM*, 35(2):345–373, 1988.

[vEK76] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, October 1976.

[VS92] 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.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1988.

[War87a] D.H.D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[War87b] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[WR87] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

[WW88] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.

[ZTD$^+$92] 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.

[ZZ89] P. Zimmermann and W. Zimmermann. The Automatic Complexity Analysis of Divide-and-Conquer Programs. Res. Rep. 1149, INRIA, France, Dec. 1989.