UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

A General Framework for Static Resource Analysis and Profiling of (Parallel) Programs and an Application to Runtime Checking

PH.D THESIS

Maximiliano Klemen
M.Sc. Software and Systems
Universidad Politécnica de Madrid

Advisor: **Dr. Pedro López García**
Ph.D in Computer Science
Universidad Politécnica de Madrid

2020

DEPARTAMENTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERIA DE SOFTWARE

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

A General Framework for Static Resource Analysis and Profiling of (Parallel) Programs and an Application to Runtime Checking

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF:
Doctor of Philosophy in Software, Systems and Computing

Author: **Maximiliano Klemen**
M.Sc. Software and Systems
Universidad Politécnica de Madrid

Advisor: **Dr. Pedro López García**
Ph.D in Computer Science
Universidad Politécnica de Madrid

November 2020

Abstract of the Dissertation

The goal of static cost analysis is to automatically estimate the resources used by program executions without running the programs with concrete data, as functions of input data sizes and possibly other (environmental) parameters. In this thesis we improve and extend state-of-the-art static cost analysis techniques by developing a novel, general and flexible framework for resource usage analysis that can be easily instantiated to infer a wide range of resources, notions of costs, and approximations, which can deal with different programming languages, platforms and execution models.

For some applications, standard resource analyses, which estimate the *total* resource usage of a program, do not provide the information required. For example, helping developers make resource-related design decisions requires knowing how such total resource usage is *distributed* over selected parts of a program. The novel, general, and flexible framework developed in this thesis solves this problem, by allowing setting up cost relations that can be instantiated for performing a wide range of resource usage analyses, including both *static profiling* and the standard notion of cost. We show how to instantiate such framework to perform *static profiling of accumulated cost* (also parameterized by input data sizes). Such information identifies the parts of the program that have the greatest impact on the total program cost.

Moreover, parallel computing has become the dominant paradigm in computer architecture, and predicting resource usage on such platforms poses a difficult challenge. We address it by extending and instantiating our general framework for performing resource usage analysis of parallel (logic) programs. Besides cost functions, the analysis also infers other useful information to better exploit and assess the potential and actual parallelism of a system.

We also develop a novel application of our cost analysis framework: inferring static performance guarantees for programs with run-time checks. Instrumenting programs for performing run-time checking of properties, such as regular shapes, is

a common and useful technique that helps programmers detect incorrect program behaviors. However, such run-time checks inevitably introduce run-time overhead (in execution time, memory, energy, etc.). We propose a method that uses static analysis to estimate such overhead. This approach can provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input, which is a parameter of the estimated cost functions, grows. Our method also extends an existing assertion verification framework to express "admissible" overheads, and statically and automatically checks whether the instrumented program conforms with such specifications.

The accuracy and applicability of our framework strongly depend on the capabilities of the component in charge of solving (or safely approximating) the cost and size recurrence relations generated during the analysis. In this thesis we propose techniques for solving recurrence relations that extend state-of-the-art solvers, addressing some of their limitations. In particular, we develop a novel approach for solving arbitrary, constrained recurrence relations. It is a *guess and check* approach that uses well-known machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a Computer Algebra System for the *check* stage. Additionally, we develop a method for solving cost relations involving a maximization operator, which appears when representing complex size and cost relations.

Finally, we report on the implementation of the techniques developed in this thesis within the CiaoPP system and their experimental evaluation, obtaining encouraging results.

Resumen de la Tesis Doctoral

El objetivo del análisis estático de coste es estimar automáticamente los recursos utilizados por los programas sin ejecutarlos con datos concretos, en forma de funciones de los tamaños de las entradas.

En esta tesis mejoramos y ampliamos las técnicas de análisis estático de coste actuales, desarrollando un marco novedoso, general y flexible de análisis, que puede instanciarse para inferir una amplia gama de recursos, nociones de costes y aproximaciones, así como tratar con diferentes lenguajes de programación y modelos de ejecución.

En algunas aplicaciones, los análisis de coste estándar no proporcionan la información requerida. Por ejemplo, para ayudar a los desarrolladores a tomar decisiones de diseño, es necesario saber cómo se distribuye el uso total de recursos entre determinadas partes de un programa. El marco novedoso, general y flexible desarrollado en esta tesis resuelve este problema al permitir establecer relaciones de coste que pueden instanciarse para realizar una amplia gama de análisis, incluyendo tanto el perfilado estático como el coste estándar. Mostramos cómo instanciar dicho marco para realizar un perfilado estático del coste acumulado (también paramétrico respecto a tamaños), que identifica las partes del programa que tienen mayor impacto en el coste total.

Por otra parte, la computación paralela se ha convertido en el paradigma de arquitectura dominante, y predecir el uso de recursos en dichas plataformas plantea un difícil reto. Para abordarlo, extendemos e instanciamos nuestro marco general para estimar el coste de programas (lógicos) paralelos. Adicionalmente, el análisis infiere información útil para explotar y evaluar mejor el paralelismo potencial y real de un sistema.

También desarrollamos una novedosa aplicación de nuestro análisis de coste: la inferencia estática de garantías de rendimiento para programas que usan la técnica de comprobación de propiedades en tiempo de ejecución. Dicha técnica se usa comúnmente para detectar comportamientos incorrectos en los programas.

Sin embargo, tales comprobaciones introducen sobrecargas en tiempo de ejecución (en términos de tiempo, memoria, energía, etc.). En esta tesis utilizamos nuestro análisis estático para estimar dichas sobrecargas, proporcionando garantías para todas las posibles ejecuciones, además de evaluar cómo crece la sobrecarga en función del tamaño de la entrada. Asímismo, extendemos un marco de verificación de aserciones existente para permitir expresar una sobrecarga *admisible*, y comprobar estáticamente si el programa instrumentado se ajusta a ella.

La aplicabilidad de nuestras técnicas de análisis depende fuertemente de las capacidades del componente a cargo de resolver o aproximar relaciones de recurrencia, el cual presenta algunas limitaciones. En esta tesis abordamos dicho reto, desarrollando un enfoque novedoso para resolver relaciones arbitrarias de recurrencia con restricciones, extendiendo los resolutores tradicionales. El nuestro es un enfoque basado *adivinar y comprobar*, usando técnicas de aprendizaje automático para la fase de *adivinar*, y una combinación de un resolutor SMT y un sistema de álgebra computacional para la etapa de comprobación. Adicionalmente, desarrollamos un método para resolver recurrencias que contienen un operador de maximización, que surgen al representar relaciones complejas de tamaño y coste.

Por último, describimos la implementación de las técnicas desarrolladas en esta tesis, integradas en el sistema CiaoPP, así como su evaluación experimental, obteniendo resultados prometedores.

A mis padres, Estela y Juan,
y a Yami

Acknowledgments

First and foremost, I would like to thank my advisor Pedro López García for giving me the greatest opportunity of my life by allowing me to work with him on such exciting research topics. His advice, all the fruitful discussions, his continuous support and guidance were absolutely fundamental for reaching this point. This thesis would not have been possible without his great value as a researcher and as a human being.

I would also like to give special thanks to Manuel Hermenegildo for his humility, selfless support and constant advice, and for transmitting his passion and enthusiasm in every talk and meeting.

My sincere thanks also to John Gallagher, for welcoming me so generously and kindly as a visiting student in Roskilde, Denmark. It was an unforgettable experience to work with him in such beautiful land. I have acquired very valuable knowledge, useful both for my thesis and my life (for example, the importance of *hygge*).

I would also like to thank Miguel Ángel Perpiñán, for generously sharing all his knowledge and expertise, which have been fundamental for the completion of this dissertation.

I would like to express my gratitude to the IMDEA Software Institute, its former Director Manuel Hermenegildo and its current Director Manuel Carro, for funding my Ph.D. and providing me with such inspiring workplace. Thanks to the support staff for their permanent help and attention in every detail. Also, thanks to all the researchers, it has been the most amazing experience to share a work environment with some of the world top researchers in their respective fields.

Many thanks to all my office mates and members of the CLIP group, both past and present, for encouraging me, listening to me, sharing their ideas and enduring my anxiety when deadlines were close: Umer Liaqat, Raúl Nestor Neri Alborodo, Joaquín Arias-Herrero, Jesús Domínguez, Isabel García Contreras, Nataliia Stulova, Bishoksan Kafle, José Francisco Morales, Luthfi Darmawan,

Rémy Haemmerlé. Ignacio de Casso, and Daniel Loscos.

On a more personal note, I would like to thank my friends Marisa and David for sharing with me, between drinks, food and laughter, their own experiences in the pursuit of a doctorate in order to encourage me.

I would like to express my deepest gratitude to my family and friends from Argentina, especially to my parents Estela and Juan. They have made a great sacrifice so that I could get here, always trusting me and encouraging me to strive for and achieve any goal I set for myself in life.

And last but not least, thanks to Yamila for pushing me forward in those moments when I wanted to go backward. This whole adventure around the world, that started in the remote Patagonia Argentina, wouldn't have been possible nor enjoyable without her company, support, bravery, and love.

Funding Acknowledgments

This research was partially supported by the EU FP7 agreement no 318337, ENTRA, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES* projects, Spanish MICINN PID2019-108528RB-C21 *ProCode* project and the Madrid M141047003 *N-GREENS* and P2018/TCS-4339 *BLOQUES-CM* programs.

Contents

List of Figures

List of Tables

1 Introduction

The execution of software consumes resources, such as time, energy, memory, or storage space, to name a few. Usually, there are constraints limiting the amount of such resources used in the environment where the software is deployed, which need to be considered during development. Every design decision is guided primarily by the expected functionality of the software, but also by these *non-functional* requirements, which are fundamental. If a system does not meet these non-functional requirements, it will not be considered correct at all. This becomes crucial, for example, on embedded systems where meeting real-time constraints is critical. The goal of automatic static resource analysis is to estimate such non-functional properties without running the program with concrete data, as functions of input data sizes and possibly other (environmental) parameters. Typical size metrics are the actual value of a number, the length of a list, the size or depth of a data structure, etc. [84, 102]. The information inferred by the analysis is a sound over-approximation of the actual behaviour of the program, considering all possible execution traces. Using this information, we can statically verify programs, obtaining guarantees about the absence of resource-related bugs, unlike testing, which only detects the presence of bugs.

In this work we assume a broad concept of resources as numerical properties of the execution of a program, such as the ones mentioned before, the classical number of *execution steps*, and user-definable resources such as the number of *calls* to a procedure, the number of *network accesses*, the number of *transactions* in a database, etc. Estimating in advance the resource usage of computations is useful for a number of applications; examples include granularity control in parallel/distributed systems, automatic program optimization, verification of resource-related specifications and detection of performance bugs, helping developers make resource-related design decisions, as well as security applications (e.g., detection of side channels attacks), and more recently, blockchain platforms (e.g., smart-contract gas analysis and verification).

Supporting different programming languages In order to deal with different programming languages and paradigms in an uniform way, we use a Horn-clause based intermediate common representation, i.e., logic programs, which any language can be translated into (preserving the resource usage semantics), and making this intermediate representation analyzable. There is a current trend favoring the use of Horn-clause programs (i.e., a set of connected code blocks) as intermediate representations in analysis and verification tools [14, 37, 52]. The main reason for choosing Horn clauses as the intermediate representation is that it offers a good number of features that make it very convenient for the analysis [72]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms.

Different notions of cost The information inferred by the analyzers is guided by its final use, i.e., each application, as the ones already mentioned, requires different types of information. For example, analyzers can infer safe approximations, namely upper and lower bounds, on the resources used by the program or parts of it, which are needed for verification. They can also infer probabilistic information, which is useful for optimisation. Static profiling (and accumulated cost) determines the distribution of resource usage over the parts of the code. This can be very useful to the developer, showing which parts of the program are the most resource-critical, and helping design decision making.

Static profiling and hot-spot identification When the target application is helping developers make resource-related design decisions, the analysis has to show which parts of the program are the most resource-consuming, i.e., which components would bring the highest overall improvement if they were optimized, so that programming efforts can be focused more productively. The standard cost information only partially meets these objectives. For example, often procedures with the highest (standard) cost are not the ones whose optimization is most profitable, since procedures which have lower costs but which are called more often may be responsible for a larger part of the overall resource usage. The input data sizes to such calls are also relevant. Thus, rather than the global costs provided by standard cost analyses, what is really needed in many such applications is the results of a *static profiling* of the program that helps identify the parts responsible for highest fractions of the cost, or, more generally, how the total resource usage of the execution of a program is *distributed* over selected parts of it. By *static profiling* we mean the static inference of the kinds of information that are usually obtained at run-time by profilers. The traditional profiling techniques are dynamic (i.e., require executing the program on some particular input) and are based either on code instrumentation, i.e., introducing additional pieces of code in the sections to be measured, or on running a process that performs the profiling together with the measured program. In both cases, the dynamic profiler introduces an overhead in the resource measured that needs to be properly discriminated, which is non trivial. For example, it may be the case that an instruction in the original

program has a very different energy consumption in the presence of code added by the profiler just before it. In contrast, the static profiling approach we propose obtains safe upper and lower bounds on resource consumption, because it is based on the semantics of the program rather than particular executions of it. I.e., the results are valid for all possible program inputs. For this reason, in this thesis we are interested in the concept of *accumulated cost*. Intuitively, accumulated cost represents how the total resource usage of the execution of a program is *distributed* over selected parts of it, called *cost centers*. Our starting point in this thesis is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes [6, 25, 26, 27, 84, 98, 102, 117], which are then solved to obtain (exact or safely approximated) closed forms of such functions (i.e., functions that provide upper or lower bounds on resource usage). Then, we build on this and develop a novel, general, and flexible framework that allows setting up cost equations/relations which can be instantiated for performing a wide range of resource usage analyses, including both *static profiling* and the inference of the standard notion of cost.

Resource usage analysis of parallel programs Regarding execution models, parallel programming is currently the mainstream technique to improve system performance. Many chip manufacturers are turning to multi-core processor designs as a way of increasing performance. However, while resource usage analysis for sequential programming languages has received considerable attention, in parallel programming there are comparatively much less results, due in part to the complexity that this paradigm adds to the problem. Parallel programming is inherently present in data centers applications, distributed systems and cloud services. The amount of energy consumed by these kind of architectures, not only for computation and communication but also for cooling, is impressively high. Thus, developing effective tools for inferring resource usage of this kind of systems is an important challenge that can contribute to reduce, for example, the footprint of energy usage worldwide, while having an important impact on the industry in terms of economic savings. In this thesis we also further extend and generalize the resource analysis framework already mentioned to deal, in a uniform and integrated way, with parallel Horn clause programs, which could be the result of a translation from a parallel imperative program or be themselves the source program. The resulting analysis estimates both lower and upper bounds on the resource usage of a parallel program as functions on input data sizes.

Recurrence relation solving The applicability of our techniques strongly depends on the capabilities of the component in charge of solving (or safely approximating) the cost and size recurrence relations generated during the analysis, which has some limitations. In this thesis we address such a challenge, proposing a novel approach for solving arbitrary, constrained recurrence relations, which extends traditional recurrence solvers. It is a *guess and check* approach that uses

well-known machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a Computer Algebra System for the *check* stage. Additionally, we develop a method for solving cost relations involving a maximization operator, which arises when setting up size and cost relations for alternative execution paths and parallel programs.

Run-Time checking overhead estimation and verification Instrumenting programs for performing run-time checking of properties, such as regular shapes, is a common and useful technique that helps programmers detect incorrect program behaviors. This is specially true in dynamic languages such as Prolog. However, such run-time checks inevitably introduce run-time overhead (in execution time, memory, energy, etc.). Several approaches have been proposed for reducing this overhead, such as eliminating the checks that can statically be proved to always succeed, and/or optimizing the way in which the (remaining) checks are performed. However, there are cases in which it is not possible to remove all checks statically (e.g., open libraries which must check their interfaces, complex properties, unknown code, etc.) and in which, even after optimizations, these remaining checks may still introduce an unacceptable level of overhead. It is thus important for programmers to be able to determine the additional cost due to the run-time checks and compare it to some notion of admissible cost. The common practice used for estimating run-time checking overhead is profiling, which is not exhaustive by nature. Instead, we propose a method that uses static analysis to estimate such overhead, with the advantage that the estimations are functions parameterized by input data sizes. Unlike profiling, this approach can provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input grows. Our method also extends an existing assertion verification framework to specify "admissible" overheads, and statically and automatically checks whether the instrumented program for run-time checking conforms with such specifications.

1.1 State of the Art

The approach to cost analysis based on setting up and solving recurrence equations was proposed in [117] and has been developed significantly in subsequent work. For example, in [98] an automatic upper-bound analysis was presented based on an abstract interpretation of a step-counting version of a functional program, in order to infer both execution time and execution steps. However, size measures could not automatically be inferred and the experimental section showed few details about the practicality of the analysis. The cost analysis in [113] deals with recursive, polymorphic and higher-order functional programs. In the context of Logic Programming, a semi-automatic analysis was presented in [25, 26] that inferred upper-bounds on the number of execution steps, given as functions on the

input data sizes. It also proposed techniques to address the additional challenges posed by the Logic Programming paradigm, as for example, dealing with the generation of multiple solutions via backtracking. However, a shortcoming of the approach was 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. This approach was later fully automated (by integrating it into the CiaoPP system and automatically providing *modes and size measures*) and extended to inferring both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [27, 48]. In addition, [27] introduced the setting up of non-deterministic recurrence relations for the class of divide-and-conquer programs mentioned above, and proposed a technique for computing approximated closed form bound functions for some of them. Such a technique was based on bounding the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [5, 6] (named Cost Relations). The approach in [25, 26, 27] was generalized in [84] to infer *user-defined resources* (by using an extension of the Ciao assertion language [46]), and was further improved in [102] by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [82, 95] of CiaoPP. The resource usage analysis in [102] is based on sized types [101]. Sized types are types that incorporate information about lower and upper bounds on the size of the terms they represents, as well as its subterms at any position and depth. Other approaches to static analysis, based on the transformation of the analyzed code into an intermediate representation, have been proposed for analyzing low-level languages [44] and Java (by means of a transformation into Java bytecode) [7]. In [7], a cost relation system is obtained directly for these bytecode programs and solve it using a specialized solver, finding upper bounds for such cost relations. In [83] the bytecode is first transformed into Horn clauses.

The general resource analyzer in [84] was also instantiated in [73] for the estimation of execution time of logic programs running on a bytecode-based abstract machine.

In [33] the authors present an amortized complexity analysis based on recurrence relations, using a novel cost representation called *cost structure*, which allows to reduce the inference of complex polynomial expressions to a set of linear problems that can be solved efficiently.

The size-change abstraction (SCA) is a program abstraction for termination analysis, successfully applied in tools for functional and logic programs. In [120], the authors establish that SCA represents also an effective abstract domain for the bound analysis of imperative programs, showing that SCA captures many of the essential ideas of previous termination and bound analysis and goes beyond in a conceptually simpler framework.

In [105] the authors present a scalable bound analysis able to perform amortized

complexity analysis. This analysis is not based on general purpose reasoners such as abstract interpreters, software model checkers or computer algebra tools. Rather, the approach is based on lossy vector addition systems (VASS). First, the analysis computes a lexicographic ranking function that proves the termination of a VASS, and then derives a bound from this ranking function.

Tools for the inference of numeric invariants provide a way to obtain information about the resource usage of programs, considering the resource to be analyzed as an extra numeric output. Compositional recurrence analysis (CRA) is a static-analysis method based on a combination of symbolic analysis and abstract interpretation. In [57] the authors address the problem of creating a context-sensitive inter-procedural version of CRA able to handle recursive procedures.

In [19] the authors present a modular automatic complexity analysis, based on an alternation between finding symbolic time bounds for program parts and using these to infer size bounds on program variables.

A number of static analyses are also aimed at worst case execution time (WCET), usually for imperative languages in different application domains (see e.g., [118] and its references). The worst-case analysis presented in [55], which is not based on recurrence equation solving, distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. However, these worst case analysis methods do not obtain functions on input data sizes as result, but rather absolute maximum execution times, in general requiring annotations from the user indicating upper bounds for the numbers of iterations of each loop.

Abstract interpretation is a formal method introduced by Patrick Cousot and Rhadia Cousot in the late 70s [23]. It is used for analyzing directly and with very few human intervention the source code of a program on some level of abstraction. Due to decidability and efficiency issues, these abstractions need to over-approximate the semantics of programs, resulting in a loss of precision. Nevertheless, we can obtain a sound and efficient analyzer by applying this technique properly. This theory was applied successfully for inferring useful information about programs, such as in [48], and also for run-time error detection in sequential and concurrent embedded avionic systems [77]. It has also been proved useful for the implementation of resource consumption analysis on sequential logic programs [102]. In [77], abstract interpretation is combined with Rely-guarantee proof methods to implement a thread-modular static analyzer for run-time error detection on concurrent embedded systems. However, there is very few work done on static analysis of the resource usage of concurrent programs using abstract interpretation.

1.1.1 Cost Analysis of Parallel Programs

With respect to the cost analysis of parallel programs, the most closely-related work to the approach presented in Chapter 4, is [51], which describes an automatic

analysis for deriving bounds on the worst-case evaluation cost of first order functional programs. The analysis derives bounds under an abstract *dual* cost model based on two measures: *work* and *depth*, which over-approximate the sequential and parallel evaluation cost of programs, respectively, considering an unlimited number of processors. Such an abstract cost model was introduced by [15] to formally analyze parallel programs. The work is based on type judgments annotated with a cost metric, which generate a set of inequalities which are then solved by linear programming techniques. The same approach is followed in [49] for the analysis of a simple imperative language with explicit parallel loops.

There are other approaches to cost analysis of parallel and distributed systems, based on different models of computation than the independent and-parallel model used in this thesis. In [9] the authors present a static analysis which is able to infer upper bounds on the maximum number of *active* (i.e., not finished nor suspended) processes running in parallel, and the total number of processes created for imperative *async-finish* parallel programs. The approach described in [4] uses recurrence (cost) relations to derive upper bounds on the cost of concurrent object-oriented programs, with shared-memory communication and future variables. In [10] the authors address the cost of parallel execution of object-oriented distributed programs.

1.1.2 Static Profiling

Regarding static profiling, the analysis presented in [41] constitutes the starting point of the work presented in Chapter 3. In [41], the analysis computes a *static profiling of accumulated cost*, where the results are also parameterized by input data sizes. However, the approach is based on a global program transformation.

Static profiling have also been considered in the context of Worst Case Execution Time (WCET) Analysis of real-time programs. In [18] the authors propose an approach for estimating the worst-case timing information for all code parts of a program using a concept called *criticality*. Similarly, in [16] the authors present static profiling techniques to estimate the execution *likelihood* and *frequency* of program points in order to assess whether the cost of certain compile-time optimizations would pay off.

1.1.3 Analysis of Run-Time Checking Overheads

Dealing with excessive run-time checking overhead is a challenging problem. Proposed approaches addressing this problem include discharging as many checks as possible via static analysis [20, 31, 42, 47, 93, 94, 108], optimizing the dynamic checks themselves [60, 88, 97, 107], or limiting run-time checking points [74].

Prior work on using profiling in the context of optimizing the performance of programs with run-time checks [34, 75, 106] clearly demonstrates the benefits of this approach. Still, profiling infers information that is valid only for some

particular input data values (and their execution traces). I.e, the profiling results thus obtained may not be valid for other input data values. Since the technique is by nature not exhaustive, detecting the worst cases can take a long time, and is impossible in general.

1.2 Thesis Objectives

The general objective of this research is the design and implementation of a novel, general, and flexible framework for resource usage analysis which can be easily instantiated to infer a wide range of resources, notions of costs (e.g., standard cost or accumulated cost), and approximations (e.g., lower or upper bounds), and can deal with different programming languages and execution models (e.g., sequential, parallel or distributed execution). In order to contribute to this general objective, we focus on the following particular objectives:

- Development of general techniques and practical, parametric tools for the inference of resource consumption information of programs. These tools will also be able to use such information for detecting early in the software lifecycle parts of the program with the greatest impact on the total program cost, and which therefore should be optimized first.
- Extend the current techniques to be able to automatically infer useful information about the resource usage of parallel programs. This information can be used to understand the impact of parallelization in terms of a set of resources, and to apply optimizations either at compile-time or at run-time.
- Extend and generalize the traditional approach for setting up and solving recurrence equations representing the computational cost of procedures as well as data sizes.
- Apply the techniques developed for the estimation and verification of the overhead introduced by run-time checking of properties in a software system.
- The techniques and concepts to be developed will be language and architecture agnostic by means of a transformation into a common Horn clause representation (i.e., logic programs), so that they could be applied to the analysis of a wide range of programming languages (and associated lower level program representations) and architectures.

1.3 Contributions

Most of the results obtained in this thesis have been published and presented in international forums that include first class conferences and JCR indexed journals.

Such publications are co-authored with other researchers, and in all of them, the contribution of the candidate has been relevant. The main contributions of the thesis are enumerated here along with the international forum these have been presented and published to, where correspond. Figure 1.1 depicts how the different contributions are related:

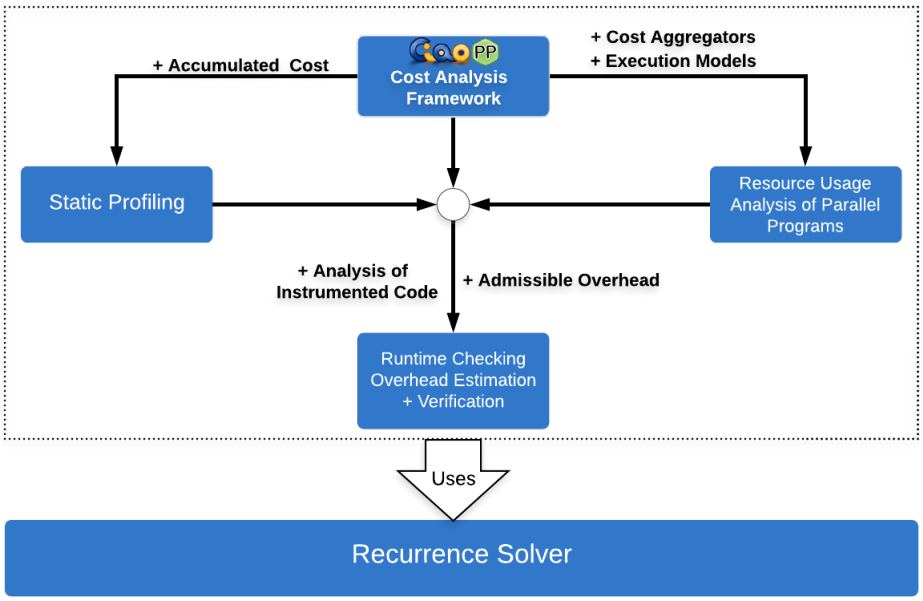


Figure 1.1: Contributions of the thesis.

- We have developed a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of resource usage analyses, including both *static profiling (of accumulated cost)* and the standard notion of cost. It is more general than [41], which is limited to accumulated cost analysis. Our new approach can deal with non-deterministic/multiple-solution predicates, unlike [41]. This is obviously a requirement for analyzing logic programs and is also useful for dealing with certain aspects of imperative programs, such as multiple dispatch. While our previous approach could conceivably be extended to deal with such programs, it would certainly result in a more complicated and indirect solution. The results of this activity have been published in [71].
- We have further generalized and extended our cost analysis framework for dealing with parallel programs. In particular, we have proposed a novel, general, and flexible framework for setting up cost equations/relations that can be instantiated for performing resource usage analysis of both sequential and parallel (logic) programs for a wide range of resources, platforms and execution models. These results have been published in [58].

- We have developed a method for solving recurrence relations involving a maximization operator, and provided correctness proofs. This class of recurrences frequently arises in size analysis, expressing sizes of inner terms of data structures. The maximization operator can also be used for expressing upper bounds for conditionals. Finally, it also arises in the analysis of parallel programs.

- We have developed a novel method, as an alternative to traditional recurrence solvers, which follows a *guess and check* approach, using well-known machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a Computer Algebra System for the *check* step. We illustrate with a set of examples how this approach is useful for improving the scalability and applicability of our static cost analysis approach based on setting up and solving recurrences.

- Finally, we have used our parametric cost analysis framework in a novel application: static estimation and verification of the overhead introduced by run-time checking. It includes the extension of an assertion verification framework to express *admissible* overheads. The common practice used for estimating the run-time checking overhead is profiling, which is not exhaustive by nature. Instead, we propose a method that uses static analysis to estimate such overhead, with the advantage that the estimations are functions parameterized by input data sizes. Unlike profiling, this approach can provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input grows.

  The results of this work have been published in [59]

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2 we describe the standard parametric cost relations framework present in the CiaoPP system, which constitutes our starting point for most of our contributions. In Chapter 3 we propose a generalization of the aforementioned framework to deal with different notions of cost, including both *static profiling* and the inference of standard cost. In Chapter 4 we describe further extensions of the framework to analyze the resource usage of parallel (logic) programs. In Chapter 5 we describe our novel approach to recurrence solving, and extensions to the existing solver. In Chapter 6 we show a novel application of our cost analysis framework to the estimation and verification of the impact in performance of run-time checking. Finally, in Chapter 7, we draw some conclusions and discuss some directions for future work.

2 The Standard Parametric Cost Relations Framework

In this chapter we describe the starting point of our work, which is the standard general framework described in [25], and extended in [27, 84], for setting up parametric relations representing the resource usage (and size relations) of logic programs.[1] The analysis infers size relations for each predicate in a program: arithmetic functions that express the size of output arguments of a predicate as a function of its input data sizes. It also infers size relations for each clause, which give the input data sizes of the body literals as functions of the input data sizes of the clause head. Such size relations are instrumental for setting up cost relations. The framework is doubly parametric: first, the costs inferred are functions of input data sizes, and second, the framework itself is parametric with respect to the type of approximation made (upper or lower bounds), and to the resource analyzed. The framework is implemented in CiaoPP [48], the preprocessor of the Ciao programming environment [46]. CiaoPP provides a rich set of static analyses whose information is also used for size and resource analysis. The rest of the chapter is organized as follows: in Section 2.1 we describe the resource usage semantics. In Section 2.2 we briefly expose the Ciao Assertion Language. In Section 2.3 we explain how a resource can be defined in the framework using user-provided assertions. In Section 2.4 we show how the size relation analysis is performed. In Section 2.5 we show in general terms how the parametric cost relations are obtained. Finally, in Section 2.6 we describe an implementation of the framework as an abstract interpretation domain, which uses regular sized types as a general size metric.

[1] We give equivalent but simpler descriptions than in [84], which are allowed by assuming that programs are the result of a normalization process that makes all unifications explicit in the clause body, so that the arguments of the clause head and the body literals are all unique variables. We also change some notation for readability and illustrative purposes.

2.1 Resource Usage Semantics

Consider a program $\mathcal{P}$, a predicate $\mathtt{p} \in \mathcal{P}$ of arity k, and the function $\mathcal{C}_\mathtt{p} : \Pi \to \mathcal{R}_\infty$, where Π is the set of k-tuples of calling data to $\mathtt{p}$. We extend such a function to the powerset of Π, i.e., $\hat{\mathcal{C}}_\mathtt{p} : 2^\Pi \to 2^{\mathcal{R}_\infty}$, where $\hat{\mathcal{C}}_\mathtt{p}(E) = \{\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) \mid \bar{\mathtt{e}} \in E\}$. Our goal is to abstract (safely approximate, as accurately as possible) $\hat{\mathcal{C}}_\mathtt{p}$ of $\mathtt{p}$ (note that $\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) = \hat{\mathcal{C}}_\mathtt{p}(\{\bar{\mathtt{e}}\})$). The goal of the analysis is to infer two functions $\hat{\mathcal{C}}^{\downarrow}_\mathtt{p}$ and $\hat{\mathcal{C}}^{\uparrow}_\mathtt{p} : \mathcal{N}^m_\top \to \mathcal{R}_\infty$ that give lower and upper bounds respectively on the cost function $\hat{\mathcal{C}}_\mathtt{p}$, where $\mathcal{N}^m_\top$ is the set of m-tuples whose elements are natural numbers or the special symbol $\top$, meaning that the size of a given term under a given size metric is *undefined*. different arguments and even (i.e., in general $m \neq k$). Such bounds are given as a function of tuples of data sizes (representing the concrete tuples of data of the concrete function $\hat{\mathcal{C}}_\mathtt{p}$). Typical size metrics are the actual value of a number, the length of a list, the size (number of constant and function symbols) of a term, etc.

2.2 The Ciao Assertion Language

In this section we introduce the subset of the Ciao assertion language that we will use in the rest of the thesis, which allows expressing global "computational" properties and, in particular, resource usage. These assertions are part of the Ciao assertion language. For brevity, we only introduce here the class of "`pred`" assertions, since they suffice for our purposes. We refer the reader to [46, 48, 93] and their references for a full description of the Ciao assertion language.

The assertions of class "`pred`" follow the schema:

```
:- [Status] pred Pred [: Precond] [=> Postcond] [+ Comp-Props].
```

where *Pred* is a predicate symbol applied to distinct free variables.[2] *Status* indicates the status of the assertion, as explained in subsection 2.2.1. *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by the set of variable/value bindings associated with a given execution step. The *Comp-Props* field (appearing after the "`+`" operator) is a logic formulae used to describe properties of the whole computation for calls to predicate *Pred* that meet *Precond*. A computation is a sequence of execution states. In order to give a general intuition about the meaning of the assertion, it can be interpreted as follows:

> In any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* should hold in the success state, and *Comp-Props* should hold for the computation performed.

[2] We do not consider assertion syntactic sugar such as *modes* for simplicity.

Also, the set of *Precond*s for all the *pred* assertions for a given *Pred* describes all the possible call states, i.e., for any call state for a predicate, there must be at least one *pred* assertion for that predicate whose *Precond* holds in that state.

2.2.1 Assertion Status

Each assertion has an associated *Status*, marked with one of the following prefixes, placed just before the `pred` keyword: `check` (indicating that the assertion is to be checked), `checked` (the assertion has been checked and proved correct by the system), `false` (it has been checked and proved incorrect by the system; a compile-time error is reported in this case), `trust` (the assertion provides information coming from the programmer in order to guide the analyzer, and it will be trusted), or `true` (the assertion is a result of static analysis and thus correct, i.e., it is a safe approximation of the concrete semantics). The default status, i.e., if no status appears before `pred`, is `check`.

2.2.2 Resource-related Properties

In this subsection we describe the most important properties related to resource usage analysis.

costb/3 This property follows the schema:

costb(*Res_Name***,** *Low_Arith_Expr, Upp_Arith_Expr)*

where *Res_Name* is a user-provided identifier for the resource the assertion refers to, *Low_Arith_Expr* and *Upp_Arith_Expr* are arithmetic functions that map input data sizes to resource usages, representing respectively lower and upper bounds on the resource consumption.

cost/3 Similarly to `costb/3`, the `cost/3` property allows expressing only one resource usage function on input data sizes. It follows the schema:

cost(*Bound_Type***,** *Res_Name***,** *Arith_Expr***)**

where *Res_Name* is the same as in `costb/3`, *Arith_Expr* is similar to *Low_Arith_Expr* and *Upp_Arith_Expr* in `costb/3`, but it can be either upper or lower bound depending on the value of *Bound_Type* which are `lb` for lower bounds and `ub` for upper bounds.

size/3 This property is used to describe the size of arguments, in terms of some metric. It follows the schema

size(*Bound_Type***,***Var***,***Arith_Expr***)**

where *Var* is the argument, and *Arith_Expr* is an arithmetic expression in terms of sizes of (possibly others) arguments. Metrics are represented as function applications in *Arith_Expr*, e.g. *length(Var) + 1*. As before, *Bound_Type* indicates the type of approximation of *Arith_Expr*.

rsize/2 Similarly to `size/3`, this property is used to describe the size of arguments, but specialized for *sized types*. It follows the schema

rsize(*Var*,*Sized_Type***)**

where *Var* is the argument, and *Sized_Type* is the sized type associated with the argument. We refer the reader to Section 2.6 for details about sized types.

2.3 Resource Definition

Each concrete resource r to be tracked is defined by two sets of (user-provided) functions, which can be constants, or general expressions of input data sizes:

1. *Head cost* $\varphi_{[ap,r]}(H)$: a function that returns an approximation of type *ap* of the amount of resource r used by the unification of the calling literal (subgoal) $\mathtt{p}$ and the head H of a clause matching $\mathtt{p}$, plus any preparation for entering a clause (i.e., call and parameter passing cost).
2. *Predicate cost* $\Psi_{[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$: it is also possible to define the *full cost* for a particular predicate $\mathtt{p}$ for resource r and approximation *ap*, i.e., the function $\Psi_{[ap,r]}(\mathtt{p}) : \mathcal{N}_{\top}^{m} \rightarrow \mathcal{R}_{\infty}$ (with the sizes of $\mathtt{p}$'s input data as parameters, $\bar{\mathtt{x}}$) that returns the usage of resource r made by a call to this predicate. This is especially useful for built-in or external predicates, i.e., predicates for which the source code is not available and thus cannot be analyzed, or for providing a more accurate function than analysis can infer. In the implementation, this information can be provided by the user to the analyzer through *trust assertions*.

2.3.1 Assertions for Resource Definitions

The parameters of the resource definition are provided in the framework through assertions. The head cost $\varphi_{[ap,r]}(H)$ is defined using the assertion

`:- head_cost`(*ResourceId*, *Bound_Type*, *Head_Cost_Pred*)

where *ResourceId* is an identifier for the resource r, *Bound_Type* is the type of approximation defined by the assertion (representing *ap*), and *Head_Cost_Pred* is a predicate that takes the head of a clause, the sizes of the arguments of the literal that generated the call, and returns an arithmetic function expressing the amount

of resource consumed. As sugar syntax, if the amount of resource consumed is constant, then the number can be provided directly in the declaration instead of *Head_Cost_Pred*.

As mentioned before, the function $\Psi_{[ap,r]}(\mathtt{p})$ can be provided for the predicate p using a `pred` assertion with the status `trust`.

2.4 Size Analysis

The first step for cost analysis is a data dependency-based method for inferring bounds on the sizes of output arguments in the head of a predicate as a function of the sizes of input arguments to the predicate. Besides this, as a result of the size analysis, we have bounds on the size of each input argument to body literals in a clause as a function of the size of the input arguments to the head of that clause. The size of the input arguments to body literals will be used later to infer functions which give bounds on the resource usage of body literals in terms of the sizes of the input arguments to the head. For the sake of brevity, here we give a brief description of the method for obtaining upper bounds, and refer the reader to [27] for details on how to obtain lower bounds.

The first element of the method is an operation $size(m, t)$, which returns the size of the term t under the metric m. This operation is defined depending on the metric, and can be provided by the user. If the size of the term is undefined for the metric, this operation returns $\bot$. Then, the operation $diff(m, t_1, t_2)$ is used to obtain the size difference between the terms t_1 and t_2, under the metric m.

A directed acyclic graph, called argument dependency graph, is used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). Each node in the graph denotes an argument position. There is an edge from a node $n1$ to a node $n2$ if the variable bindings generated by $n1$ are used to construct the term occurring at $n2$. The node $n1$ is said to be a predecessor of the node $n2$, and $n2$ a successor of $n1$. This graph can be explicitly built before the analysis, as in [25], or can be represented implicitly trough size relations as we explain in Section 2.6.

Using the size and diff functions and the argument dependency graph, the analysis set up size relations for expressing the size of each argument position in terms of the sizes of its predecessors. Let `sz(a)` denote the size of the term occurring at an argument position `a`, and `@a` the term occurring at an argument position a. We omit the metric in the size and diff functions in the rest of the text, for readability purposes.

Output Arguments Let $l_1, \ldots, l_n$ denote the input argument positions of the literal L, and let $\psi_p^b : \mathcal{N}_{\bot,\infty}^n \to \mathcal{R}_\infty$ be a function that represents the size of the b-th output argument of p of literal L in terms of the size of its input arguments, where $\mathcal{N}_{\bot,\infty}$ is the set of natural numbers augmented with $\bot$ and ∞, representing

undefined and infinite sizes, respectively. For each output argument position a, the following relation is set up:

$$sz(a) \leq \psi_p^b(sz(l_1), \ldots, sz(l_n))$$

If L is a non-recursive call, then ψ_p^b can be computed independently, and we replace $\psi_p^b(sz(l_1), \ldots, sz(l_n))$ by the closed-form expression obtained after the analysis of the predicate that L references. In case L corresponds to a recursive call (either direct or indirect), then $\psi_p^b(sz(l_1), \ldots, sz(l_n))$ is kept as a symbolic expression, which will be used afterwards for setting up a recurrence relation.

Input Arguments Let $predecessors(i)$ be the set of predecessors of i in the argument dependency graph. For each input argument position i, we have the following possibilities:

- if $size(@i) \neq \bot$, then $sz(i) \leq size(@i)$.
- Otherwise, if $\exists r \in predecessors(i)$ such that r and i have the same metric, and $diff(@r, @i) \neq \bot$, then $sz(i) \leq size(@r) + diff(@r, @i)$.
- Otherwise, if $size(@i)$ can be expanded one step using its definition, computing recursively the $size(t_j)$ subexpressions appearing in the expansion, where t_j is a subterm of $@i$. If $\forall t_j \cdot size(t_j) \neq \bot$, then we use them to compute $size(@i)$. Otherwise, $sz(i) = \bot$.

Finally, size relations are propagated to transform a size relation corresponding to an input argument in a body literal or an output argument in the clause head into a function in terms of the sizes of the input arguments of the head. For recursive clauses, we obtain recurrence relations that need to be solve, or safely approximate.

2.5 Inferring Resource Usage Functions

In order to infer the resource usage functions, all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider a predicate p defined by clauses $C_1, \ldots, C_m$. Assume $\bar{\mathtt{x}}$ are the sizes of p's input parameters. Then, the resource usage (expressed in units of resource r with approximation ap) of a call to p, for an input of size $\bar{\mathtt{x}}$, denoted as $C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$, can be expressed as:

$$C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}}) = \bigodot_{1 \leq i \leq m} (C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})) \tag{2.1}$$

where $\bigodot = ClauseAggregator(ap, r)$ is a function that takes an approximation identifier ap and returns a function that applies over the cost of all the clauses,

$C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})$, for $1 \leq i \leq m$, in order to obtain the cost of a call to the predicate p. For example, if ap is the identifier for approximation "upper bound" (ub), then a possible conservative definition for $ClauseAggregator(ub, r)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). However, it is straightforward to take mutual exclusion into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses, as done in [102].

Let us see now how to compute the resource usage of a clause. Consider a clause C of predicate p of the form H :- $L_1, \ldots, L_k$ where $L_j, 1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and H is the clause head. Assume that $\psi_j(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to literal L_j, given as functions of the sizes of the input arguments to the clause head. Note that these $\psi_j(\bar{\mathtt{x}})$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses. Then, the cost relation for clause C and a single call to p (obtaining all solutions), is:

$$C_{cl[ap,r]}(C, \bar{\mathtt{x}}) = \varphi_{[ap,r]}(H) + \sum_{j=1}^{lim(ap,C)} sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}})) \tag{2.2}$$

where $lim(ap, C)$ gives the index of the last body literal that is called in the execution of clause C, and $sols_j$ represents the product of the number of solutions produced by the predecessor literals of L_j in the clause body:

$$sols_j(\bar{\mathtt{x}}) = \prod_{i=1}^{j-1} s_{pred}(L_i, \psi_i(\bar{\mathtt{x}})) \tag{2.3}$$

where $s_{pred}(L_i, \psi_i(\bar{\mathtt{x}}))$ gives the number of solutions produced by L_i, with arguments of size $\psi_i(\bar{\mathtt{x}})$. The number of solutions and size relations are both inferred automatically by the framework (we refer the reader to [25, 26, 27, 102] for a description).

Finally, $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by one of the following expressions, depending on L_j:

- If L_j is a call to a predicate q which is in the same strongly connected component as p (the predicate under analysis), then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the symbolic call $C_{pred[ap,r]}(\mathtt{q}, \psi_{\mathtt{j}}(\bar{\mathtt{x}}))$, giving rise to a recurrence relation that needs to be bounded with a closed-form expression by the solver afterwards.

- If L_j is a call to a predicate q which is in a different strongly connected component than p, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{x}))$ is replaced by the closed-form expression that bounds $C_{pred[ap,r]}(q, \psi_j(\bar{x}))$. The analysis guarantees that this expression has been inferred beforehand, due to the fact that the analysis is performed for each strongly connected component, in a reverse topological order.

- If L_j is a call to a predicate q, whose cost is specified (with a trust assertion) as $\Psi_{[ap,r]}(q, \bar{y})$, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{x}))$ is replaced by the expression $\Psi_{[ap,r]}(q, \psi_j(\bar{x}))$.

2.6 Resource Analysis as an Abstract Domain

The general framework presented so far, based on setting up recurrences representing size and resource usage information, is also implemented in CiaoPP using abstract interpretation, based on the sized types abstract domain [101, 102]. Sized types are regular types extended with structural (shape) information that express both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. They also allow relating the sizes of terms and subterms occurring at different argument positions in logic predicates. With this information, the resource analysis can infer both lower and upper bounds on the resources used by all the procedures in a program as functions on the sizes of input terms (and subterms). The abstract domain operations are based on the setting up and solving of recurrence equations for inferring both size and resource usage functions.

We give now an overview of the approach for resource usage analysis by abstract interpretation present in CiaoPP, showing the main ideas by using the classical `append/3` predicate as a running example:

```
1 append([],     S, S).
2 append([E|R], S, [E|T]) :- append(R, S, T).
```

The process starts by performing the regular type analysis present in the CiaoPP system [114]. In our example, the system infers that for any call to the predicate `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, if the call succeeds, then `Z` also gets bound to a list of numbers. The set of "list of numbers" is represented by the regular type *listnum*, defined as follows:

```
listnum := [] | [num | listnum]
```

From this regular type definition, sized type schemes are derived. The sized type schema *listnum-s* is derived from *listnum*. This schema corresponds to a list whose length is between α and β, containing numbers between γ and δ.

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)})$$

From now on, in the examples we will use ln and n instead of $listnum$ and num for the sake of conciseness. The next phase involves relating the sized types of the different arguments to the `append/3` predicate using recurrence (in)equations. Let $size_X$ denote the sized type schema for argument `X` in a call `append(X, Y, Z)` (from the regular type inferred by a previous analysis). We have that $size_X$ denotes $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})$. Similarly, the sized type schema for the output argument `Z` is $ln^{(\alpha_Z,\beta_Z)}(n^{(\gamma_Z,\delta_Z)})$, denoted by $size_Z$. We are interested in expressing bounds on the length of the output list `Z` and the values of its elements as a function of size bounds for the input lists `X` and `Y` (and their elements). For this, we set up a system of inequations. For instance, the inequations that are set up to express a lower bound on the length of the output argument `Z`, denoted α_Z, as a function on the size bounds of the input arguments `X` and `Y`, and their subarguments ($\alpha_X,\ \beta_X,\ \gamma_X,\ \delta_X,\ \alpha_Y,\ \beta_Y,\ \gamma_Y$, and δ_Y) are:

$$\alpha_Z\begin{pmatrix}\alpha_X,\beta_X,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}\alpha_X-1,\beta_X-1,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

Note that in the recurrence inequation set up for the second clause of `append/3`, the expression $\alpha_X - 1$ (respectively $\beta_X - 1$) represents the size relationship that a lower (respectively upper) bound on the length of the list in the first argument of the recursive call to `append/3` is one unit less than the length of the first argument in the clause head.

As the number of size variables grows, the set of inequations becomes too large. Thus, in [102] the authors propose a compact representation, which allows us to grasp all the relations in one view. The first change in this representation is to write the parameters to size functions directly as sized types. Now, the parameters to the α_Z function are the sized type schemas corresponding to the arguments `X` and `Y` of the `append/3` predicate:

$$\alpha_Z\begin{pmatrix}ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

In a second step, all the relations of a single sized type are grouped together. Throughout this chapter we use a representation using $\lessgtr$ for the symbols $\geq$ and $\leq$ that are always paired, as the authors of [102] proposed. In the implementation, constraints for each variable are kept apart and solved separately.

After setting up the corresponding system of inequations for the output argument `Z` of `append/3`, and solving it, we obtain the following expression:

$$size_Z\,(size_X, size_Y) \lessgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))})$$

that represents, among others, the relation $\alpha_z \geq \alpha_X + \alpha_Y$ (resp. $\beta_z \leq \beta_X + \beta_Y$), expressing that a lower (resp. upper) bound on the length of the output list `Z`, denoted α_z (resp. β_z), is the addition of the lower (resp. upper) bounds on the lengths of `X` and `Y`. It also represents the relation $\gamma_Z \geq \min(\gamma_X, \gamma_Y)$ (resp. $\delta_Z \leq \max(\delta_X, \delta_Y)$), which expresses that a lower (resp. upper) bound on the size of the elements of the list `Z`, denoted γ_z (resp. δ_z), is the minimum (resp. maximum) of the lower (resp. upper) bounds on the sizes of the elements of the input lists `X` and `Y`.

Resource analysis builds upon the sized type analysis and adds recurrence equations for each resource we want to analyze. Apart from that, when considering logic programs, we have to take into account that they can fail or have multiple solutions when executed, so we need an auxiliary *cardinality analysis* to get correct results.

Let s_L and s_U denote lower and upper bounds on the number of solutions for `append/3`. Following the program structure we can infer:

$$\begin{array}{rcl} s_L\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & 1 \\ s_L\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & s_L\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ s_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 \\ s_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{array}$$

Since $s_L \leq s_U$, the solution to these inequations must be $(s_L, s_U) = (1, 1)$. Thus, we have inferred that `append/3` has at least (and at most) one solution: it behaves like a function. When setting up the equations, the analysis uses the result of the non-failure analysis to see that `append/3` cannot fail when given lists as arguments. If not, the lower bound is 0.

Now we move forward to the resource usage approximation. We are considering the number of resolution steps performed by a call to `append/3` (we will only focus on upper bounds, r_U, for brevity). For the first clause, it is clear that only one resolution step is needed, so:

$$r_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq 1$$

The second clause performs one resolution step plus all the resolution steps performed by all possible backtrackings over the call in the body of the clause. This number can be bounded as a function of the number of solutions. Thus, the equation reads:

$$\begin{array}{rcl} r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 + s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ & & \times\ r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ & = & 1 + r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{array}$$

Solving these equations the analysis infers that an upper bound on the number of resolution steps is the (upper bound on) the length of the input list `X` plus one. This is expressed as:

$$r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq \beta_X + 1$$

2.7 Example

Consider the following program that checks whether a number n is prime based on Wilson's theorem: *any integer* $n > 1$ *is prime iff* $(n-1)! \equiv -1 \pmod{n}$. Equivalently, n is prime iff $(n-1)! + 1$ is a multiple of n.

```
1  prime(X):-
2    X > 1,
3    X1 is X - 1,
4    fact(X1,F1),
5    F is F1 + 1,
6    multiple(F,X).
7
8  fact(X,1):- X = 1.
9  fact(X,Y):-
10   X > 1,
11   X1 is X - 1,
12   fact(X1,Y1),
13   Y is Y1*X.
```

Assume that `multiple` is a naively implemented library predicate, so that its resource usage, in number of resolution steps, is linear on the size of the input: $\mathtt{C}_{\mathtt{multiple}}(n,m) = n+1$ if $n > 1$. Assuming we don't have access to the implementation of `multiple`, its resource usage function is provided to the analysis by the following trust assertion:

```
1  :- trust pred multiple(N,M) : (num(N),num(M))
2            => (num(N), rsize(N,num(NLB, NUB)))
3            + (costb(steps, NLB, NUB + 1)).
```

Assume that we want to infer the standard cost of this predicate in resolution steps, i.e., we define $\varphi_{[ub,steps]}(H) = 1$ for all predicates $\mathtt{p} \in \mathcal{P}$. For brevity, we also assume that we are only interested in inferring upper bounds on resource usages, so that the expression $C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$ appearing in Equation 2.1 is understood to represent an upper bound, and, assuming no definite failure information, then $lim(C, \bar{\mathtt{x}})$ is the index of the last body literal of clause C. Finally, we also assume that size relations have been inferred for the different arguments in a clause, and that the size metric used is the actual value of an argument, since all arguments

are numeric. Such relations are obvious in this example, so that we focus only on cost relations. However, as already stated, CiaoPP is able to infer and deal with a rich set of size metrics, and also infer such size relations. The size of the kth output argument of predicate $pred$, given as a function of the input data sizes $\bar{n}$ to that predicate is represented as $Sz^k_{pred}(\bar{n})$. It is important also to mention the modes of these predicates (again, inferred automatically by CiaoPP): `prime` has one input argument and no output; `multiple` has two input arguments and no output; and `fact` has one input and one output, whose size we have assumed is already inferred in terms of the size of the input by the size analysis. This size is represented by $Sz^2_{\texttt{fact}}(n)$, and is obtained from the setting up of the following size relation:

$$\begin{array}{rcll} Sz^2_{\texttt{fact}}(n) = & 1 & & \text{if } n = 1 \\ Sz^2_{\texttt{fact}}(n) = & n \times Sz^2_{\texttt{fact}}(n-1) & & \text{if } n > 1 \end{array}$$

By solving this recurrence, the size analysis obtains the closed-form $Sz^2_{\texttt{fact}}(n) = n!$. Regarding the number of solutions, in this example all the predicates generate at most one solution, thus $\forall i : sols_i = 1$ in Equation 2.2. Now we have all the necessary elements to set up the cost relations for `prime`, `fact`, and `multiple`:

$$\begin{array}{rcll} \texttt{C}_{\texttt{fact}}(n) = & 1 & & \text{if } n = 1 \\ \texttt{C}_{\texttt{fact}}(n) = & 1 + \texttt{C}_{\texttt{fact}}(n-1) & & \text{if } n > 1 \\ \texttt{C}_{\texttt{multiple}}(n,m) = & n + 1 & & \text{if } n > 1 \end{array}$$

$$\texttt{C}_{\texttt{prime}}(n) \;=\; 2 + \texttt{C}_{\texttt{fact}}(n-1) + \texttt{C}_{\texttt{multiple}}(Sz^2_{\texttt{fact}}(n-1)+1, n) \text{ if } n > 1$$

Note that in this program, the size of the input of the call to `multiple` is given by the size of the output of `fact`, represented by $Sz^2_{\texttt{fact}}(n)$. After solving these equations and composing the closed forms, we obtain the following closed form functions:

$$\begin{array}{rcll} \texttt{C}_{\texttt{fact}}(n) = & n & & \text{if } n > 1 \\ \texttt{C}_{\texttt{multiple}}(n,m) = & n + 1 & & \text{if } n > 1 \\ \texttt{C}_{\texttt{prime}}(n) = & (n-1)! + n + 3 & & \text{if } n > 1 \end{array}$$

3 Static Profiling

3.1 Introduction

In this chapter, we take as starting point the general framework for resource usage analysis present in the CiaoPP system, described in Chapter 2, which is parametric with respect to resources, approximations (e.g., lower or upper bounds), and programming languages, and further generalize and extend it to be also parametric with respect to notions of costs (e.g., standard cost or accumulated cost).

Standard cost, and, in general, resource usage information, is very useful for a number of applications, such as automatic program optimization, verification of resource-related specifications, detection of performance bugs, or helping developers make resource-related design decisions. In the latter case, the analysis has to show which parts of the program are the most resource-consuming, i.e., which predicates would bring the highest overall improvement if they were optimized, so that programming efforts can be focused more productively. The standard cost information only partially meets these objectives. For example, often predicates with the highest (standard) cost are not the ones whose optimization is most profitable, since predicates which have lower costs but which are called more often may be responsible for a larger part of the overall resource usage. The input data sizes to such calls are also relevant. Thus, rather than the global costs provided by standard cost analyses, what is really needed in many such applications is the results of a *static profiling* of the program that helps identify the parts of a program responsible for highest fractions of the cost, or, more generally, how the total resource usage of the execution of a program is *distributed* over selected parts of it. By *static profiling* we mean the static inference of the kinds of information that are usually obtained at run-time by profilers.

For this reason, herein we are more interested in what we refer to as *accumulated cost*. To give an intuition of this concept, we first explain our notion of *cost centers*, which is similar to the one we use in [41], and was inspired from [79, 99]: they are

user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program. Data about computational events is accumulated by the cost center each time the corresponding program point is reached by the program execution control flow. Assume for example that predicate $\mathtt{p}$ calls another predicate $\mathtt{q}$ (either directly or indirectly), and that we declare that both predicates are cost centers. In this case, the cost of a (single) call $\mathtt{p}(\bar{\mathtt{e}})$ *accumulated in* cost center $\mathtt{q}$, denoted $\mathcal{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{e}})$, expresses how much of the standard cost of $\mathtt{p}(\bar{\mathtt{e}})$ is attributed to $\mathtt{q}$, and is the sum of the costs of the computations performed "under the scope" of all the calls to $\mathtt{q}$ generated during the complete execution of $\mathtt{p}(\bar{\mathtt{e}})$. We say that a computation is "under the scope" of a call to cost center $\mathtt{q}$ if the closest ancestor of such computation in the call stack that is a cost center is $\mathtt{q}$. The *accumulated cost* is formalized as a function $\mathcal{C}^{\mathtt{q}}_{\mathtt{p}} : \Pi \rightarrow \mathcal{R}_\infty$. We refer the reader to [41] for a formal definition of accumulated cost.[1]

The goal of static analysis is to infer approximations (i.e., abstractions) of the concrete functions $\mathcal{C}^{\mathtt{q}}_{\mathtt{p}}$ and $\mathcal{C}_{\mathtt{p}}$ (or, more precisely, of the extensions of such functions to the powerset of Π) that represent the *accumulated* and *standard* cost respectively.

In this chapter we propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of static resource usage analyses, including both *accumulated cost* and standard cost. Our starting point is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes [6, 25, 26, 27, 84, 98, 102, 117], which are then solved to obtain (exact or safely approximated) closed-forms of such functions (i.e., functions that provide upper or lower bounds on resource usage in general).[2] Our proposal extends and generalizes these standard resource analysis techniques by introducing into the derived relations extra Boolean control variables whose value is 0 or 1. A particular resource profile can be analyzed by assigning values to the control variables, effectively switching on or off different terms in the relations. The standard resource analysis is obtained by assigning 1 to all variables. We also define a concrete Boolean variable assignment that instantiates our framework so that it performs *static profiling of accumulated cost*, similarly to [41], where the results are also parameterized by input data sizes. However, the approach we present in this thesis is quite different from the one in [41], which was based on a program transformation. Our main contributions, and the differences and advantages over that work can be summarized as follows:

- We propose a novel, general, and flexible framework for setting up cost relations which can be instantiated for performing a wide range of resource

[1] In [41] we use the notation $\mathtt{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{\mathtt{e}})$ instead of $\mathcal{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{e}})$.

[2] In addition, recently many other approaches have been proposed for resource analysis [6, 11, 36, 38, 39, 50, 54, 87, 113]. While based on different techniques, all these analyses are aimed at inferring the *standard* notion of cost. Please see [41] for a further discussion of related work.

usage analyses. Is more general than [41], which is limited to accumulated cost analysis.

- Our new approach can deal with non-deterministic/multiple-solution predicates, unlike [41]. This is obviously a requirement for analyzing logic programs and is also useful for dealing with certain aspects of imperative programs, such as multiple dispatch; see [72]. While our previous approach could conceivably be extended to deal with such programs, it would certainly result in a more complicated and indirect solution.

- Our new approach and its implementation are based on a direct application of abstract interpretation and integration into the Ciao preprocessor, CiaoPP [48], rather than on a program transformation. As a result, many useful CiaoPP features are inherited for free, such as *multivariance* (being able to infer separate cost functions for different abstract call patterns for the same predicate), communication with the other required analyses, integrated treatment of special control features (such as, e.g., the cut), assertion-based verification and user interaction, efficient fixpoint, etc. Also, for this integration we define a novel abstract domain for resource analysis that keeps track of the *environment*.

- Furthermore, this direct implementation avoids the disadvantages of the transformation-based approach, such as making it more difficult to relate the results (and warnings/errors) to the original program, and complicating the task of the auxiliary analyses needed for cost analysis (types, modes, determinism, non-failure, etc.). This is because if the analyses are performed on the original program, then the results need to be transferred to the transformed program; and if the analyses are performed on the transformed program, then there is always the risk of loss of precision. Also, the transformation required by our previous approach is global, which is problematic for modular compilation. In general, this new approach allows much better and easier integration in a real-world compilation infrastructure.

- The integration also inherits the capability of CiaoPP's analyzers of *analyzing for several resources* at the same time. While it might be possible to define a new transformation capable of keeping track of several resources, this would further complicate the transformed program, and in any case requires additional work.

- Finally, as our experimental results show, our new approach is more efficient than the transformation-based approach. This is not only due to its implementation as a direct abstract interpretation, but also to the inclusion and use of reachability information, performed automatically by the abstract interpretation framework.

In the rest of the chapter, Sect. 3.2 describes the proposed generalized approach in which the cost relations extracted from the program incorporate extra Boolean variables controlling the profile to be analyzed. Sect. 3.3 presents an instantiation of such extended approach for static profiling of accumulated cost. Sect. 3.4 describes the implementation of the technique using the CiaoPP program analysis framework, and provides experimental results. Sect. 3.6 discusses related work, and finally, Sect. 3.7 presents our conclusions.

3.2 Generalizing the Standard Cost Relations Approach

Our proposal extends and generalizes the approach described in Chapter 2. We introduce a new concept of cost, $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}})$, representing the (part of the) cost of the complete execution of a single call $\mathtt{p}(\bar{\mathtt{x}})$ (i.e., $\mathtt{C}_{\mathtt{p}}(\bar{\mathtt{x}})$ in Section 2.1 of Chapter 2), performed in an *environment* e, that is attributed/assigned to *cost center* $\mathtt{c}$ of the program. The parameter e is used to capture a broad notion of *environment*. For example, it can be just the name of a predicate that is an ancestor of $\mathtt{p}$ in the call stack. In a more complex setting, for example when inferring hardware-dependent resources, such as energy [64, 65, 83], e can also include information about the state of the hardware (or the whole system, including the running software environment), e.g., the last instruction executed (useful for modeling the *switching cost* of instructions), temperature, voltage, cache state, and pipeline state. There is of course a trade-off between the amount of information in e and analysis efficiency and accuracy.

As already said, and similarly to [41], herein we assume that a *cost center* is a predicate in the program. Conceptually, we can say that we extend the notion of resource so that it is now a pair $(\mathtt{c}, r)$, where r is a resource identifier as before (e.g., resolution steps, execution time, energy, etc.), and $\mathtt{c}$ is the cost center (predicate) that the resource usage is attributed/assigned to.

We introduce *Boolean functions* $B_{\varphi}(\mathtt{p}, \mathtt{c}, e)$ and $B(\mathtt{p}, \mathtt{c}, e, \mathtt{q})$ to control which terms of the cost relation should be considered. Using these boolean functions, we generalize Equation 2.2 as:

$$\begin{aligned} \mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}}) = {} & B_{\varphi}(\mathtt{p}, \mathtt{c}, e) \times \varphi(H) \\ & + \sum_{j=1}^{lim(ap,C)} sols_j(\bar{\mathtt{x}}) \times B(\mathtt{p}, \mathtt{c}, e, \mathtt{q}_j) \times \mathtt{C}^{\mathtt{c}}_{\mathtt{q}_i,e'}(\psi_j(\bar{\mathtt{x}})) \end{aligned} \tag{3.1}$$

where $e' = \mathcal{E}(\mathtt{p}, \mathtt{c}, e, \mathtt{q}_j(\psi_j(\bar{\mathtt{x}})))$, and $\mathcal{E}$ is the *environment change* function, which obtains the new environment for $\mathtt{q}_j$. If the cost of $\mathtt{p}$ is given (by using a trust

assertion) as a function $\Psi(\mathtt{p})(\bar{\mathtt{x}})$, then:

$$\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}}) = B_{\varphi}(\mathtt{p},\mathtt{c},e) \times \Psi(\mathtt{p})(\bar{\mathtt{x}}) \tag{3.2}$$

Again, this equational framework can be instantiated to obtain the standard cost by defining $B_{\varphi}(\mathtt{p},\mathtt{c},e) = B(\mathtt{p},\mathtt{c},e,\mathtt{q}) \equiv 1$, and defining $\mathcal{E}$ so that it does not change the environment and always returns the input environment, i.e., $\mathcal{E}(\mathtt{p},\mathtt{c},e,\mathtt{q}_i(\bar{\mathtt{x}}_i)) = e$. The *standard* cost $\mathtt{C}_{\mathtt{p}}(\bar{\mathtt{x}})$ is then given by $\mathtt{C}^{\mathtt{p}}_{\mathtt{p},\perp}(\bar{\mathtt{x}})$, where $\perp$ is the *null* environment, in which no information about the environment is tracked, and the only cost center that the cost of a single call to $\mathtt{p}$ is attributed to is the predicate $\mathtt{p}$ itself.

Notation For the sake of readability, in this chapter we simplify the general notation used in Equation 2.2. Concretely, we omit global parameters *ap* and r, indicating type of approximation and the resource under analysis respectively, whenever they can be deduced from the context. Also, as a slight abuse of notation, we omit subscripts $cl, pred$ and lit from the cost function symbol C, considering they can also be deduced from the context.

3.3 Instantiation for Parametric Accumulated-cost Static Profiling

We now instantiate the general approach described in Sect. 3.2 for the static inference of accumulated cost. As mentioned before, in [41] we proposed a technique for this purpose, but following a quite different approach that required a global program transformation, which implies the limitations listed in Sect. 3.1. The approach we propose herein is more general and flexible, deals with essential aspects such as non-determinism and multiple solutions, and does not require any program transformation.

Assume we are given a set of (user-defined) cost centers $\Diamond$, which, as mentioned before, in our approach program predicates. Assuming that $\mathtt{p}$ is a cost center, the standard cost of a single call $\mathtt{p}(\bar{\mathtt{x}})$ (as defined in Sect. 3.1, and whose inference was discussed in Sect. 2.1 of Chapter 2) is the sum of its accumulated costs in all the cost centers in the program, or, equivalently in all the cost centers that are descendants (in the call stack) of $\mathtt{p}$. This is formally expressed in [41] Theorem 1, and, intuitively, the proof is based on the fact that, according to the definition of accumulated cost, the cost of any computation performed during the complete execution of $\mathtt{p}(\bar{\mathtt{x}})$ is uniquely attributed to a cost center (predicate): the closest ancestor of such computation in the call stack that is a cost center.

Given a predicate $\mathtt{p}$, we refer to the computations performed by a call $\mathtt{p}(\bar{\mathtt{x}})$ that are not under the scope of any cost center that is a descendant (in the call stack) of $\mathtt{p}$, as the *residual computations* of $\mathtt{p}$. We refer to the cost of such computations

as the *residual cost* of $\mathtt{p}$. Note that such computations include the computations performed by calls to non-cost-center predicates that are descendants of $\mathtt{p}$ and that are not under the scope of any cost center that is a descendant of $\mathtt{p}$. Assume that the analysis is inferring accumulated costs on a given cost center $\mathtt{c}$. When analyzing a call to a non-cost-center predicate $\mathtt{p}$, its residual cost must be attributed to $\mathtt{c}$ only if the call $\mathtt{p}(\bar{\mathtt{x}})$ is under the scope of $\mathtt{c}$ (i.e., is a descendant of $\mathtt{c}$). When analyzing a call to a cost-center predicate $\mathtt{p}$, its residual cost must be attributed to $\mathtt{c}$ only if $\mathtt{p} = \mathtt{c}$. Thus, in the expression $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}})$ (where necessarily $\mathtt{c} \in \Diamond$) the environment e is just a Boolean value representing whether the (single) call to $\mathtt{p}$ is in the scope of cost center $\mathtt{c}$ ($e = 1$) or not ($e = 0$). To this end, we define the *environment change* function as follows: $\mathcal{E}(\mathtt{p}, \mathtt{c}, e, \_) \equiv (\mathtt{p} = \mathtt{c} \vee (\mathtt{p} \notin \Diamond \wedge e))$.

Knowing that a given predicate cannot be called by another during program execution allows the analysis to ignore some parts not affecting the cost to be inferred. We define a simple *calls* relation between predicates as: $\mathtt{p}$ calls $\mathtt{q}$, denoted $\mathtt{p} \rightsquigarrow_\alpha \mathtt{q}$, if and only if a literal with predicate symbol $\mathtt{q}$ appears in the body of a clause defining $\mathtt{p}$; $\rightsquigarrow^\star_\alpha$ is the reflexive transitive closure of $\rightsquigarrow_\alpha$. This $\rightsquigarrow_\alpha$ relation is an abstraction (over-approximation) of the concrete $\rightsquigarrow$ relation (a more precise abstraction is computed by CiaoPP).

The Boolean assignment functions (appearing in Expression 3.1) are defined as follows:

$$B_\varphi(\mathtt{p}, \mathtt{c}, e) \equiv (\mathtt{p} = \mathtt{c} \vee (\mathtt{p} \notin \Diamond \wedge e)) \tag{3.3}$$

$$B(p, \mathtt{c}, e, q) \equiv B_\varphi(p, \mathtt{c}, e) \vee (q \rightsquigarrow^\star_\alpha c) \tag{3.4}$$

Note that the analysis of the accumulated cost of a given non-cost-center predicate $\mathtt{p}$ in a given cost center $\mathtt{c}$ can create at most two versions of $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},e}(\bar{\mathtt{x}})$ for the same input (calling pattern) $\bar{\mathtt{x}}$ (and hence, there will be at most two versions of the cost relations for $\mathtt{p}$): the version $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},1}(\bar{\mathtt{x}})$ created if there is a (direct or indirect) call to $\mathtt{p}$ in the scope of $\mathtt{c}$, e.g., if such call is in the body of a clause defining $\mathtt{c}$ (in which case the φ *cost* is added to the cost relations for $\mathtt{p}$), and the variant $\mathtt{C}^{\mathtt{c}}_{\mathtt{p},0}(\bar{\mathtt{x}})$ created if there is a call to $\mathtt{p}$ not in the scope of $\mathtt{c}$ (in which case the φ *cost* is not added).

Lemma 1. *$\forall \mathtt{p}, \mathtt{q} \in \Diamond$, $\forall e \in \{0, 1\}$, it holds that $\mathcal{E}(\mathtt{p}, \mathtt{q}, e, \_) \equiv (\mathtt{p} = \mathtt{q})$ and $B_\varphi(\mathtt{p}, \mathtt{q}, e) \equiv (\mathtt{p} = \mathtt{q})$.*

This implies that:

Lemma 2. *$\forall \mathtt{p}, \mathtt{q} \in \Diamond$ it holds that $\mathtt{C}^{\mathtt{q}}_{\mathtt{p},0}(\bar{\mathtt{x}}) = \mathtt{C}^{\mathtt{q}}_{\mathtt{p},1}(\bar{\mathtt{x}})$.*

Thus, if $\mathtt{p} \in \Diamond$ we omit the environment e and write $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{x}})$. Note that necessarily $\mathtt{q} \in \Diamond$.

Lemma 3. *$\forall \mathtt{p}, \mathtt{q} \in \Diamond$, if $\mathtt{p} \not\rightsquigarrow^\star_\alpha \mathtt{q}$ then $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\bar{\mathtt{x}}) = 0$*

Lemma 4. $\forall \mathtt{p} \notin \Diamond, \forall \mathtt{q} \in \Diamond$, *if* $\mathtt{p} \not\rightsquigarrow^{\star}_{\alpha} \mathtt{q}$ *then* $\mathtt{C}^{\mathtt{q}}_{\mathtt{p},0}(\bar{\mathtt{x}}) = 0$

Note also that in the standard cost relation-based static analysis, cost relations are set up for each predicate in the program. In the approach we propose here for accumulated cost, cost relations are set up for each cost center and for each predicate in the program.

Example 1. *In Example 2.7, predicate* `prime` *was found too expensive in terms of resolution steps to be practical, since* $\mathtt{C}_{\mathtt{prime}}(n) \in \mathcal{O}(n!)$. *However, the standard cost inferred for all the predicates called from* `prime` *is linear, and it is not easy to detect at first glance where the resource is really consumed. To locate the culprit, traditionally this would be attempted using a dynamic profiling tool, executing the program with several test cases –commonly known as* hot spot detection. *However, as with the standard cost analysis, we want to detect such hot spots* statically, *in order to have sound information for* any *possible input. For this purpose, we perform the accumulated cost analysis declaring that all predicates are cost centers (i.e,* $\Diamond = \{\mathtt{prime}, \mathtt{fact}, \mathtt{multiple}\}$*). Based on the equational framework instantiation in Sect. 3.3 and Lemma 2, consider the cost of a single call to* `prime` *accumulated in* `fact`, $\mathtt{C}^{\mathtt{fact}}_{\mathtt{prime}}(n)$, *for an input size* n. *As already stated, the number of solutions of all these predicates is 1, and the output sizes have already been inferred. For the sake of conciseness, from now on we refer to* `prime`, `fact` *and* `multiple` *as* p, f *and* m *respectively.*

$$\mathtt{C}^f_p(n) = \; B_\varphi(p, f, \_) \times \varphi(p(n)) + \mathtt{C}^f_f(n-1) + \mathtt{C}^f_m(Sz^2_f(n-1))$$

$$\begin{aligned} \mathtt{C}^f_f(n) &= \; B_\varphi(f, f, \_) \times \varphi(f(n)) && \text{if } n = 1 \\ \mathtt{C}^f_f(n) &= \; B_\varphi(f, f, \_) \times \varphi(f(n)) + \mathtt{C}^f_f(n-1) && \text{if } n > 1 \end{aligned}$$

$$\mathtt{C}^f_m(n) = \; B_\varphi(m, f, \_) \times \Psi(m)(n) = B_\varphi(m, f, \_) \times (n+1) \quad \text{if } n > 1$$

Following the definitions in Sect. 3.3, we know that $B_\varphi(p, f, \_) = B_\varphi(m, f, \_) = 0$, $B_\varphi(f, f, \_) = 1$ *and* $\varphi(\_) = 1$. *Using these values, the cost relations defining* $\mathtt{C}^f_p(n)$ *are:*

$$\mathtt{C}^f_p(n) = \; \mathtt{C}^f_f(n-1)$$

$$\begin{aligned} \mathtt{C}^f_f(n) &= \; 1 && \text{if } n = 1 \\ \mathtt{C}^f_f(n) &= \; 1 + \mathtt{C}^f_f(n-1) && \text{if } n > 1 \end{aligned}$$

Solving this system of equations, we finally obtain: $\mathtt{C}^f_p(n) = n$
Analogously, we obtain the closed-form functions for $\mathtt{C}^m_p(n)$ *and* $\mathtt{C}^p_p(n)$*:*

$$\begin{aligned} \mathtt{C}^p_p(n) &= \; 1 && \text{if } n > 1 \\ \mathtt{C}^m_p(n) &= \; (n-1)! + 2 && \text{if } n > 1 \end{aligned}$$

Now, it is clear that the most expensive part of this program is the call to `multiple`*. Even though the (standard) cost of this implementation of* `multiple` *is linear, its input size is the output size of the call to* `fact`*, which is the factorial of the input to* `prime` *minus 1. In this case the problem can really only be fixed by using a better implementation of* `multiple` *($\mathcal{O}(1)$) or of* `prime`*, to achieve the expected polynomial resource usage.*

This example illustrates how the accumulated cost is more useful than the standard cost. Neither the standard cost of `multiple` *($n+1$) nor the number of calls to this predicate from* `prime` *(since it is called just once) gives a direct hint that this predicate is responsible for most of the resource consumption of* `prime`*.*

Example 2. *Consider the following program $\mathcal{P}$:*

```
1  p(X,Y):-
2    h(X),
3    q(X,Y),
4    w(Y),
5    s(X).
6
7  q(0,_).
8  q(X,Y):-
9    X > 0,
10   X1 is X - 1,
11   m(Y),
12   q(X1,Y),
13   s(X).
```

```
14 m(0).
15 m(X):-
16   X > 0,
17   w(X),
18   X1 is X - 1,
19   m(X1).
20
21 s(0).
22 s(X):-
23   X > 0,
24   X1 is X - 1,
25   w(X),
26   s(X1).
27
28 h(2).
29 h(3).
```

Assume as in the previous example that we want to infer upper bounds of the standard costs of all the predicates in resolution steps, i.e., $\varphi(\mathtt{p}(\bar{\mathtt{x}})) = 1$ for all predicates $\mathtt{p} \in \mathcal{P}$. Assume also that `w` *is a library predicate and that its (standard) cost is given as a predicate cost function (by using a trust assertion):*

$$\Psi(\mathtt{w})(x) = 2x + 1 \tag{3.5}$$

We assume again that the size metric used is the actual value of the arguments, since they are all numeric, and that size relations, again obvious, have been inferred for all clause arguments, which are all inputs, and we focus only on cost relations. The cost relation for the recursive clause of predicate `s`*, according to Expression 3.1 is (for simplicity, $sols_i = 1$ for all predicates in this example):*

$$\mathtt{C_s}(x) = \quad 1 + \mathtt{C_w}(x) + \mathtt{C_s}(x-1) \quad \textit{if } x > 0$$

Since $\mathtt{C_w}(x)$ is given by a trust assertion as $\Psi(\mathtt{w})(x) = 2\ x + 1$, this cost relation, together with the one for the non-recursive clause, form the system:

$$\begin{array}{lll} \mathtt{C_s}(x) = & 1 & \textit{if } x = 0 \\ \mathtt{C_s}(x) = & 1 + 2x + 1 + \mathtt{C_s}(x-1) & \textit{if } x > 0 \end{array}$$

and its closed-form solution is $\mathtt{C_s}(x) = x^2 + 3x + 1$ *for* $x \geq 0$*. The same cost relations correspond to predicate* `m`*, therefore its closed form is* $\mathtt{C_m}(x) = x^2 + 3x + 1$ *for* $x \geq 0$*. For predicate* `h`*, the following non-recursive system of cost relations is set up:*

$$\mathtt{C_h}(x) = 1, \ \textit{if } x = 2 \qquad \textit{and} \qquad \mathtt{C_h}(x) = 1, \ \textit{if } x = 3$$

obtaining $\mathtt{C_h}(x) = 1$*, since the clauses of* `h` *are mutually exclusive. Now, the cost relations for* `q` *are:*

$$\begin{array}{lll} \mathtt{C_q}(x,y) = & 1 & \textit{if } x = 0 \\ \mathtt{C_q}(x,y) = & 1 + \mathtt{C_m}(y) + \mathtt{C_q}(x-1,y) + \mathtt{C_s}(x) & \textit{if } x > 0 \end{array}$$

Replacing $\mathtt{C_m}(y)$ *and* $\mathtt{C_s}(y)$ *with their corresponding closed-form functions obtained before, and solving the recurrence, we obtain* $\mathtt{C_q}(x,y) = \frac{1}{3}x^3 + xy^2 + 2x^2 + 3xy + \frac{14}{3}x + 1$*. Finally, the cost relations for the main predicate* `p` *result in:*

$$\mathtt{C_p}(x,y) = \ 1 + \mathtt{C_h}(x) + \mathtt{C_q}(x,y) + \mathtt{C_w}(y) + \mathtt{C_s}(x)$$

and its closed form is: $\mathtt{C_p}(x,y) = \frac{1}{3}x^3 + xy^2 + 3x^2 + 3xy + \frac{23}{3}x + 2y + 4$.

Now assume that we declare that predicates `p`*,* `q`*,* `m` *and* `h` *are cost centers, i.e.,* $\Diamond = \{\mathtt{p}, \mathtt{q}, \mathtt{m}, \mathtt{h}\}$*, and* `s` *and* `w` *are not. For space reasons, we will only illustrate the inference of upper bounds on accumulated costs in all cost centers.*

The accumulated costs in cost center `q` *are inferred as follows. Consider the clause defining predicate* `p`*. Since* $\mathtt{p} \in \Diamond$*, by Lemma 1 the current environment* e *is irrelevant for the computation of the new environment* e' *(i.e.,* $e' = \mathcal{E}(\mathtt{p}, \mathtt{q}, 0, \_) = \mathcal{E}(\mathtt{p}, \mathtt{q}, 1, \_) \equiv (\mathtt{p} = \mathtt{q}) \equiv 0$*), and for the computation of the head cost, i.e.,* $B_\varphi(\mathtt{p}, \mathtt{q}, 0) = B_\varphi(\mathtt{p}, \mathtt{q}, 1) \equiv (\mathtt{p} = \mathtt{q}) \equiv 0$*. Thus, the cost relation for* `p` *according to Equation 3.1 is* $\mathtt{C}^{\mathtt{q}}_{\mathtt{p}}(\mathtt{x}, \mathtt{y}) = \mathtt{C}^{\mathtt{q}}_{\mathtt{h}}(\mathtt{x}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{w},0}(\mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},0}(\mathtt{x})$*. Consider predicate* `q` *now. Since* $B_\varphi(\mathtt{q}, \mathtt{q}, e) \equiv (\mathtt{q} = \mathtt{q}) \equiv 1$ *and* $B_\varphi(\mathtt{q}, \mathtt{q}, 0) = B_\varphi(\mathtt{q}, \mathtt{q}, 1) \equiv (\mathtt{p} = \mathtt{q}) \equiv 1$ *for* $e \in \{0, 1\}$*, the cost relations for the base case and recursive clause of* `q` *respectively are:*

$$\begin{array}{ll} \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) = B_\varphi(\mathtt{q}, \mathtt{q}, \_) \times 1 = 1 \times 1 = 1 & \textit{if } \mathtt{x} = 0 \\ \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}, \mathtt{y}) = 1 + \mathtt{C}^{\mathtt{q}}_{\mathtt{m}}(\mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{q}}(\mathtt{x}-1, \mathtt{y}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) & \textit{if } \mathtt{x} > 0 \end{array}$$

For expression $\mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x})$ *appearing in the recursive cost relation for* `q` *above (i.e., the version of the cost of* `s` *when called in the scope of cost center* `q`*), the cost relations are:*[3]

$$\begin{array}{lll} \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) = & 1 & \textit{if } x = 0 \\ \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}) = & 1 + \mathtt{C}^{\mathtt{q}}_{\mathtt{w},1}(\mathtt{x}) + \mathtt{C}^{\mathtt{q}}_{\mathtt{s},1}(\mathtt{x}-1) & \textit{if } \mathtt{x} > 0 \end{array}$$

[3] Since $\mathtt{s} \notin \Diamond$, the environment is needed in this case.

We now need to infer the function represented by expression $C^{q}_{w,1}(x)$ *appearing in the recursive cost relation for* s *above. Since the cost function for* w *is given by a trust assertion (see Expression 3.5) and* $B_{\varphi}(w, q, 1) = 1$*, we have that* $C^{q}_{w,1}(x) = B_{\varphi}(w, q, 1) \times (2\,x + 1) = 2\,x + 1$*. Using this function, the closed-form solution for* $C^{q}_{s,1}(x)$ *is* $x^2 + 3x + 1$ *for* $x \geq 0$*. For expression* $C^{q}_{w,0}(y)$ *appearing in the equation for* p *above, we have that* $C^{q}_{w,0}(y) = B_{\varphi}(w, q, 0) \times (2y+1) = 0 \times (2y+1) = 0$*. Now, for expression* $C^{q}_{s,0}(x)$ *appearing in the cost relation for* p *above (i.e., the version of the cost of* s *when it is not called in the scope of cost center* q*), we have that* $C^{q}_{s,0}(x) = 0$ *(Lemma 4). For expression* $C^{q}_{m}(y)$ *appearing in the second cost relation for* q *above, we have that* $C^{q}_{m}(y) = 0$ *(Lemma 3), and no cost relation is set up for predicate* m*. Now, the accumulated costs in cost center* h *are inferred as follows. The accumulated cost in* h *for a call to* p *is given by:*

$$C^{h}_{p}(x, y) = C^{h}_{h}(x) + C^{h}_{q}(x, y) + C^{h}_{w,0}(y) + C^{h}_{s,0}(x)$$

We have that:

$$C^{h}_{q}(x, y) = C^{h}_{m}(y) = 0 \text{ (by Lemma 3)}$$

and:

$$C^{h}_{s,0}(x) = \text{ (by Lemma 4)}$$

and $C^{h}_{w,0}(y) = B_{\varphi}(w, h, \_) \times 1 = 0 \times 1 = 0$*. Then, the cost relations for the accumulated cost in* h *for a call to* h *are:*

$$\begin{aligned} C^{h}_{h}(x) &= B_{\varphi}(h, h, \_) \times 1 = 1 \\ C^{h}_{h}(x) &= B_{\varphi}(h, h, \_) \times 1 = 1 \end{aligned}$$

Therefore, $C^{h}_{h}(x) = 1$ *and* $C^{h}_{p}(x, y) = 1$*. For cost center* m *we have:*

$$\begin{aligned} C^{m}_{p}(x, y) &= xy^2 + 3xy + 2y + \tfrac{1}{3}x^3 + \tfrac{5}{2}x^2 + \tfrac{25}{6}x + 1 \\ C^{m}_{q}(x, y) &= xy^2 + 3xy + \tfrac{1}{3}x^3 + \tfrac{3}{2}x^2 + \tfrac{13}{6}x \\ C^{m}_{m}(x) &= x^2 + 3x + 1 \end{aligned}$$

Finally, for cost center p *we have:*

$$C^{p}_{p}(x, y) = 1 \quad C^{p}_{q}(x, y) = 0 \quad C^{p}_{m}(x) = 0 \quad C^{p}_{h}(x) = 0$$

Note that the large complexity of $C^{m}_{p}(x, y)$ *makes us realize that if we move the call* m(y) *from the recursive clause of* q *to the clause of* p*:*

```
1 p(X, Y):- h(X), m(Y), q(X, Y), w(Y), s(X).
2
3 q(0, _).
4 q(X, Y):- X > 0, X1 is X - 1, q(X1, Y), s(X).
```

then, the standard cost of p *will be reduced. In particular, it is reduced from* $C_{p}(x, y) = \frac{1}{3}x^3 + xy^2 + 3x^2 + 3xy + \frac{23}{3}x + 2y + 4$ *to* $C_{p}(x, y) = y^2 + 5y + \frac{1}{3}x^3 + 3x^2 + \frac{20}{3}x + 8$*.*

3.4 Implementation and Experimental Results

We have implemented the proposed approach within the CiaoPP system, by extending the implementation of [102]. The latter improved on [84] by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [82, 95] of CiaoPP, inheriting features such as multivariance, efficient fixpoints, and assertion-based verification and user interaction. A significant additional improvement brought about by [102] is it use of a *sized types* abstract domain, which allows the inference of non-trivial cost bounds when these depend on the sizes of parts of input terms at any position and depth. The resulting abstract interpretation-based implementation builds the cost equations described in Sect. 3.2. Separate equations are built for each procedure *version* thanks to the built-in multivariance in PLAI. Other optimizations include not building equations for unreachable program parts.

Table 3.1 shows the results of the comparison between the proposed approach and our previous, program transformation-based approach [41] –**New** and **Prev** respectively from now on. Column **Bench** shows, for each program, the entry predicate (marked with a *star*, e.g., *sublist**) and the predicates that are declared as cost centers (which always include the entry predicate). **Acc. Cost** shows the parametric accumulated cost functions inferred for each cost center, which depend on the input data sizes of the entry predicate. For conciseness we only show upper bound functions, although in the experiments both upper and lower bounds were inferred. The resource inferred in these tests is the number of resolution steps (i.e., each clause body is assumed to have unitary cost). The symbols in Column **C** compare **New** and **Prev**: $\times$ means that it is a non-deterministic program that produces multiple solutions and **New** is able to obtain non-trivial bounds while **Prev** fails to obtain a correct bound (as mentioned before, **Prev** is not applicable for these programs). $=$ indicates that **New** obtains the same bounds as **Prev**. Only these two symbols are required because all the results coincide except for the non-deterministic programs. **AvD** is the average deviation of the accumulated costs obtained by evaluating the functions in Column **Acc. Cost**, with respect to the costs measured with a dynamic profiler [75]. The input data for dynamic profiling was selected to exhibit worst case execution,[4] in order to compare with upper bound functions. **Time (s)** lists the analysis times of **New** in seconds (Ciao/CiaoPP version 1.15-4048-g6bd1569, MacBook Pro, 2.4GHz Intel Core i7 CPU, 8 GB 1333 MHz DDR3 memory, MAC OS X Lion 10.7.5) and, between brackets, how efficient **New** is with respect to **Prev** ($\frac{\textbf{New}-\textbf{Prev}}{\textbf{Prev}} \times 100$). **New** is more efficient than **Prev** in all programs, with one exception (hanoi). Times are quite encouraging in any case, specially considering the currently inefficient implementation of the Mathematica interface, one of the solvers used for the recurrence equations.

[4] Except for *queens*: the *queens* program was simply run for 8 queens. The selection of input data that can make a program exhibit worst case execution is non-trivial.

Table 3.1: Experimental results (static profiling of accumulated cost).

Bench	Acc. Cost	C	AvD	Time (s)	Std. Cost	#Calls	Acc.BigO
*sublist**	n_2+3	×	5%	4.7	$n_1n_2+3n_2+2$	2	$\mathcal{O}(n_2)$
append	$n_1n_2+2n_2-1$		40%	(NA)	$2n-1$	$n_1n_2+2n_2-1$	$\mathcal{O}(n_1n_2)$
*is_prime**	1	=	0%		$(n-1)!+n+3$	1	$\mathcal{O}(1)$
fact	n		0%	1.6	n	n	$\mathcal{O}(n)$
mult	$(n-1)!+2$		0%	(-24%)	$n+1$	$(n-1)!+2$	$\mathcal{O}(n!)$
*queens**	$n+2$	×	7%		$\mathcal{O}(n^n)^\dagger$	1	$\mathcal{O}(n)$
consistent	$\frac{((n-1)n-1)n^{n+1}+n}{(n-1)^2}$		10^4%	4.7	$2n+1$	$\frac{((n-1)n-1)n^{n+1}+n}{(n-1)^2}$	$\mathcal{O}(n^n)$
choose	$\frac{(2n-1)(n^n-1)}{(n-1)}$		10^4%	(NA)	$2n-1$	$\frac{(2n-1)(n^n-1)}{n-1}$	$\mathcal{O}(n^n)$
noattack	$\frac{(n-2)n^{(n+2)}+n^2}{(n-1)^2}$		10^4%		1	$\frac{(n-2)n^{n+2}+n^2}{(n-1)^2}$	$\mathcal{O}(n^n)$
*search**	1	×	0%	1.4	$2n+2$	1	$\mathcal{O}(1)$
member	$2n+1$		0.1%	(NA)	$2n+1$	$2n+1$	$\mathcal{O}(n)$
*appAll2**	b_1	=	0%	5.3	$\mathcal{O}(b_1b_2b_3)^\dagger$	1	$\mathcal{O}(b_1)$
appAll	b_1b_2		0%	(-16%)	b_1b_2	b_1	$\mathcal{O}(b_1b_2)$
append	$2b_1b_2b_3$		0%		n	$b_1b_2+b_1$	$\mathcal{O}(b_1b_2b_3)$
*hanoi**	2^n-1	=	0%	1.6	$2^{n+1}-2$	1	$\mathcal{O}(2^n)$
move	2^n-1		0%	(-19%)	1	2^n-1	$\mathcal{O}(2^n)$
*coupled**	1	=	0%	2.4	$n+2$	1	$\mathcal{O}(1)$
p	$\frac{n}{2}+\frac{(-1)^n}{4}+\frac{3}{4}$		1.2%	(-14%)	$n+1$	$\frac{n}{2}-\frac{(-1)^n}{4}+\frac{1}{4}$	$\mathcal{O}(n)$
q	$\frac{n}{2}-\frac{(-1)^n}{4}+\frac{1}{4}$		0%		$n+1$	$\frac{n}{2}+\frac{(-1)^n}{4}-\frac{1}{4}$	$\mathcal{O}(n)$
*isort**	$n+1$	=	0%	3	n^2+n+1	$n+1$	$\mathcal{O}(n)$
insert	n^2		71%	(-19%)	$2n+1$	n^2	$\mathcal{O}(n^2)$
*minsort**	$n+1$	=	0%	3.5	$\frac{(n+1)^2}{2}+\frac{n+1}{2}$	1	$\mathcal{O}(n)$
findmin	$\frac{(n+1)^2}{2}+\frac{n-1}{2}$		7%	(-27%)	n	$n+1$	$\mathcal{O}(n^2)$
*dyade**	n_1	=	0%	3.2	$n_1(n_2+1)$	1	$\mathcal{O}(n_1)$
mult	n_1n_2		0%	(-20%)	n	n_1	$\mathcal{O}(n_1n_2)$
*variance**	1	=	0%	3.6	$2n^2$	1	$\mathcal{O}(1)$
sq_diff	$n-1$		0%	(-39%)	$2n_2n_1-2n_2$	$n-1$	$\mathcal{O}(n)$
mean	$2n^2-n$		0%		$2n+1$	n	$\mathcal{O}(n^2)$
*variance2**	1	=	0%	3.1	$5n+3$	1	$\mathcal{O}(1)$
sq_diff	n		0%	(-40%)	n	n	$\mathcal{O}(n)$
mean	$4n+2$		0%		$2n+1$	2	$\mathcal{O}(n)$
*listfact**	b_1	=	0%	1.9	$b_1(b_2+2)$	1	$\mathcal{O}(b1)$
fact	$b_1b_2+b_1$		0%	(-23%)	n	b_1	$\mathcal{O}(b_1b_2)$

† For space limitations only the complexity order is shown.

- $n_1, n_2, \ldots, n_k$ represent the sizes of k input arguments. For a single input argument, the subscript is dropped.
- $b_1, b_2, \ldots, b_k$ represent the sizes of the nested structures of an input argument, where b_1 represents the size of the outer most structure and b_k the inner most. In cases, where the cost only depends on the outer most structure, the previous representation is used.

Std. Cost shows the cost functions inferred using the standard notion of cost (in particular, the cost functions inferred by [102]) for comparison with the accumulated cost functions (**Acc. Cost**). The latter clearly signal hot spots that are not visible from the standard cost functions. Note also that in all cases the sum of the functions for all the cost centers is the standard cost of the entry predicate. Due to space limitations we do not include analysis times for obtaining the standard costs in Column **Std. Cost**, but while the analysis times of **New** are higher, as expected, it is only by 20% on average. **#Calls** shows the number of times each predicate is called, as a function of input data sizes of the entry predicate. These functions are inferred using the standard analysis by defining explicitly a **#Calls** *resource* for each cost center predicate. A large complexity order in the number of calls to a predicate (in relation to that of a single call) suggests that it could be profitable to optimize the program to reduce the number of calls to this predicate, to effectively reduce its impact on the overall cost of the program. More interestingly, since both resources **Acc. Cost** and **#Calls** of a predicate `p` are expressed as functions of input data sizes of the entry predicate, their quotient (**Acc. Cost**/ **#Calls**) is meaningful and will give an approximation of the cost of a single call to `p` as a function of the input data sizes *of the entry predicate*. Note that the standard analysis (Column **Std. Cost**) also provides an upper-bound approximation of this cost but as a function of the input data sizes of `q`. Finally, Column **Acc.BigO** shows the *actual* asymptotic resource usage of the accumulated cost in different cost centers.

3.5 Hot Spots Detection using Static Profiling

The main objective of the approach presented in this chapter is to help identifying the most resource-consuming parts of a program and the causes for them. In this section, we illustrate with an example how we can achieve such goal by combining the accumulated and the standard cost, the size analysis, together with a user-defined resource to count the number of calls to a particular predicate.

Let $k_1, k_2, \ldots, k_r$ be nonnegative integers, and let $n = k_1 + k_2 + \cdots + k_r$. The *multinomial coefficient* is defined as

$$\binom{n}{k_1, k_2, \ldots, k_r} = \frac{n!}{k_1! k_2! \cdots k_r!}$$

and represents the number of ways n objects can be arranged into r sets, where the i-th set contains exactly k_i objects. Consider the following program that

computes the multinomial coefficient, following such definition exactly:

```
1  multi_coef(N,KL,Res):-
2    fact(N,N1),
3    lfact(KL,KLF),
4    multiply(KLF,Div),
5    Res is N1/Div.
6
7  multiply([],1).
8  multiply([X|Y],Z):-
9    multiply(Y,Z0),
10   Z is Z0*X.
```

```
11 lfact([],[]).
12 lfact([N|R],[F|Fr]):-
13   fact(N,F),
14   lfact(R,Fr).
15
16 fact(0,1).
17 fact(N,Fact):- N>0,
18   N1 is N-1,
19   fact(N1,Fact1),
20   Fact is N*Fact1.
```

We first use our cost analysis tool to infer the standard cost of the main predicate, `multi_coef/3` in our example, given in terms of resolution steps. For brevity, we only show upper bounds in this example, although the process is analogous for the case of lower bounds. We assume that size relations have been inferred for the different arguments in a clause, and that the size metrics used are the number of type rule applications for lists, and the actual value for numbers. After running the analyzer, we obtain the following closed form functions representing the standard cost of each predicate in the program:

$$\mathtt{C}_{\mathtt{fact}}(n) = n+1 \qquad \mathtt{C}_{\mathtt{multiply}}(l) = l$$
$$\mathtt{C}_{\mathtt{lfact}}(l,e) = el-e+2l-1 \qquad \mathtt{C}_{\mathtt{multi\_coef}}(n,l,e) = el-e+3l+n+1$$

where l represents an upper bound on the size of the input list, e represents an upper bound on the size of elements in the input list (which are all numbers in this example), and n is an upper bound on the number of objects to arrange. Note that the standard cost of our interest, $\mathtt{C}_{\mathtt{multi\_coef}}(n,l,e)$, is in $O(el+n)$. We know from this what is the overall cost of our program. However, this information does not show details about how the cost is distributed among the different predicates.

3.5.1 Hot Spots Identification

In general, in order to optimize the performance of an existing piece of code, a straightforward procedure consists of a) identify the most costly parts of it, and b) try to optimize those parts. To achieve this, we run our static profiling tool declaring all the predicates in the program as cost centers. In the example, the resulting accumulated costs are the following:

$$\mathtt{C}^{\mathtt{fact}}_{\mathtt{multi\_coef}}(n,l,e) = el-e+l+n \qquad \mathtt{C}^{\mathtt{multiply}}_{\mathtt{multi\_coef}}(n,l,e) = l$$
$$\mathtt{C}^{\mathtt{lfact}}_{\mathtt{multi\_coef}}(n,l,e) = l \qquad \mathtt{C}^{\mathtt{multi\_coef}}_{\mathtt{multi\_coef}}(n,l,e) = 1$$

Then, we rank the cost centers based on the complexity orders, identifying `fact/2` as the predicate where most of the resource consumption is occurring, and focus our optimizations efforts on aspects related to `fact/2`.

3.5.2 Calls and Size Analysis

After identifying the cost center(s) where most of the resource is consumed, we need to fetch more details about the causes of that behavior, in order to take appropriate actions. Intuitively, the factors causing a high accumulated cost in a given predicate possibly are:

- a high standard cost of the predicate (i.e. the predicate is *intrinsically expensive* in terms of cost),
- a high number of calls to the predicate,
- a *big* input data in some of the calls to the predicate, or
- a combination of all the factors above.

At this stage, the analysis has already obtained bounds on the size of the outputs for all the predicates related to the call of our interest. In our example, these bounds are the following:

$$\begin{array}{rlrl} Sz^2_{\texttt{fact}}(n) = & n! & Sz^2_{\texttt{lfact}}(l) = & l \\ Sz^2_{multiply}(l,n) = & n^{l-1} & Sz^{2/(./1)}_{\texttt{lfact}}(n) = & n! \end{array}$$

The next information we need to infer is an upper bound on the number of calls made to each hot spot detected before. We need to distinguish *external* from *internal* (directly recursive) calls. While internal calls are caused by the control flow of the predicate, depending on its initial input, calls originated externally do not dependent on the callee. This difference is important in order to know where we can potentially apply optimizations. To this end, we can define a resource, $calls[w]$, that represents the number of times that predicate w is called. In order to identify the different sources of the calls, we use the accumulated cost in terms of this resource. If we want the number of calls to w, originated from w itself (i.e., the *internal* calls of w), in the context of a single call to the predicate p, this information is exactly given by C^w_p. The external calls to w, on the other hand, can be given by C^p_p, declaring only p and w as cost centers for the resource $calls[w]$. In our example, the analyzer obtains the number of external and internal calls to `fact/2`, which is the one with the biggest accumulated cost:

$$\begin{array}{rll} Calls^{external}_{\texttt{fact}}(l) & = \mathtt{C}^{\texttt{multi\_coef}}_{\texttt{multi\_coef}}(l) & = l \\ Calls^{internal}_{\texttt{fact}}(l,e) & = \mathtt{C}^{\texttt{fact}}_{\texttt{multi\_coef}}(l,e) & = 2le - 2e \end{array}$$

Note that most of the calls to `fact/2` are originated internally (recursively).

3.5.3 Interpreting the Results

So far, the analysis pointed out which are the hot-spots with the highest cost, together with the distribution of calls to these parts, and the size complexity of each predicate. Combining this information we can draw some *conclusions* about where and how it is worth to focus most of the optimization efforts. In our running example, based on the information obtained, we conclude that `fact/2` is a good target for optimizations, and that a possible action to take is to reduce the number of internal calls to *fact/2*, which means, reducing its standard cost. Intuitively, the analysis *suggests* this after observing that:

1. $Calls^{internal}_{\texttt{fact}}(l, e) \in O(el)$ is *higher* than $Calls^{external}_{\texttt{fact}}(l) \in O(l)$
2. The input received by `fact/2` has the same size than the input of *multi_coef/2*, without any increment through the data path.

3.5.4 Hot Spots Optimizations

Finally, we can apply optimizations to the hot-spots detected, taking into account the causes spotlighted by the analysis. The optimizations can be automatic, releasing the developer of this task. For example, we can declare the hot spots as tabled predicates, or automatically parallelize them. In other cases, we can modify the code using some common performance improving techniques, such as, dynamic programming, or simply replacing the code with a better, closer to optimal algorithm.

Continuing with our example, lets pay closer attention to how we use this predicate in particular. First, each time we compute $n!$, we also compute $k!$ for $1 \leq k < n$. By saving these intermediate results in an auxiliary data structure, we can avoid unnecessary recomputations, and calls to *fact/2* consequently. Second, we actually know from the input the biggest number for which we need to compute factorial, which is $n = k_1 + \cdots + k_r$. These two facts allow us to precompute all the necessary factorial numbers in linear time, storing them in an efficient data structure accessible from the rest of the program when necessary. The optimized

program is the following (we assume that the cost of the builtin `arg/3` is 0):

```
multi_coef_opt(N,KL,R):-
  inc_fact(N,Memo),
  fact_cons(N,Memo,N1),
  lfact(Memo,KL,KLF),
  multiply(KLF,Div),
  R is N1/Div.

multiply([],1).
multiply([X|Y],Z):-
  multiply(Y,Z0),
  Z is Z0*X.

inc_fact(N,Fact):-
  inc_fact_(0,N,1,Fact0),
  Fact =.. [p|Fact0].
```

```
inc_fact_(N,N,_F,[]).
inc_fact_(I,N,F0,[F1|F]):-
  I<N,
  I1 is I + 1,
  F1 is I1*F0,
  inc_fact_(I1,N,F1,F).

fact_cons(N,T,E):-
  arg(N,T,E).

lfact(_,[],[]).
lfact(Memo,[N|R],[F|Fr]):-
  fact_cons(N,Memo,F),
  lfact(Memo,R,Fr).
```

Then, we run the analysis again, this time obtaining the following accumulated costs (with the new set of cost centers):

$$\mathtt{C}^{\mathtt{fact\_cons}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = l \qquad \mathtt{C}^{\mathtt{multiply}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = l$$
$$\mathtt{C}^{\mathtt{inc\_fact}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = 1 \qquad \mathtt{C}^{\mathtt{lfact}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = l$$
$$\mathtt{C}^{\mathtt{inc\_fact\_}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = n+1 \qquad \mathtt{C}^{\mathtt{multi\_coef\_opt}}_{\mathtt{multi\_coef\_opt}}(n,l,e) = 1$$

Now the total cost of a call to *multi_coef/3* is bounded by $\mathtt{C}_{\mathtt{multi\_coef}}(n,l,e) = 3l+n+3$. Thus, after our modification we have reduced the number of resolution steps significatively, improving the asymptotic behaviour of the program from $O(el+n)$ to $O(l+n)$. Figure 3.1 compares both bounds graphically, for a list of 100 subsets ($l = 100$). We can continue this procedure iteratively, taking in each iteration the next cost center identified as hot spot by the analysis. This example has shown how our general framework helps with the process of improving the overall resource usage of a program.

3.6 Related Work

Wegbreit [117] proposed a general approach to cost analysis, based on setting up and solving recurrence equations, for simple lisp programs. This seminal work has been developed significantly in subsequent work, addressing limitations, increasing automation, applying to several language paradigms, and extending the notions of cost. For example, also within functional programming, [98] proposed an automatic upper-bound analysis based on an abstract interpretation of a step-counting version of the functional program, in order to infer both execution time and execution steps. However, elements like size measures had to be provided by

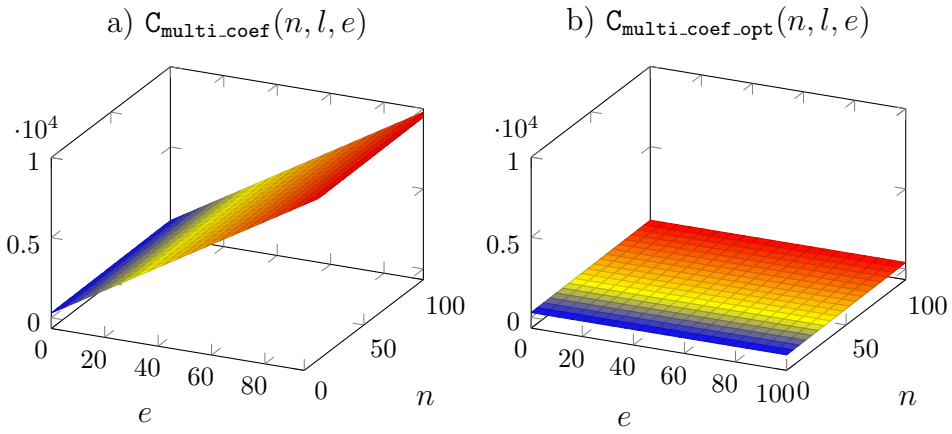


Figure 3.1: Comparison of the costs (in resolution steps) between `multi_coeff/3` and its optimized version `multi_coeff_opt/3` (for $l = 100$).

hand and few details were provided about the practicality of the analysis.

In the context of logic programming, [25, 26] presented an analysis that inferred upper-bounds on the number of execution steps, given as functions on the input data sizes. This work also proposed techniques to address the additional challenges posed by the logic programming paradigm, such as, for example, dealing with non-deterministic programs that generate multiple solutions and provided experimental results, including those from an application in granularity control for automatic program parallelization. This work also required annotations for modes and size measures. It was however fully automated in [27, 48, 70] (by integrating it into CiaoPP and using its analyzers to automatically provide modes and size measures) and extended to infer both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) [27].

In addition, [27] introduced the idea of setting up *non-deterministic* recurrence relations and applied it to the class of divide-and-conquer programs (for which previous approaches incurred in loss of precision), proposing as well a technique for computing approximated closed form bound functions for some of them. The proposed technique was based on bounding the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [6] (referred to there as cost *relations*).

The approach in [25, 26, 27] was generalized in [84] to infer *user-defined resources* (by using an extension of the Ciao assertion language [46]), and was further improved in [102] by defining the cost relations and functions as an *abstract domain* and the whole resource analysis as a multivariant abstract interpretation. This is, as mentioned before, the implementation that we have used as starting point for our experiments.

Recently, many other approaches have been proposed for resource analysis [6, 11, 36, 38, 50, 54, 87, 113]. While based on different techniques, all the analyses discussed share the goal of inferring the *standard* notion of cost –that we have called herein the *standard cost*– which provides for a given program $\mathcal{P}$, and for all predicates $\mathtt{p} \in \mathcal{P}$, an approximation of the total cost of a single call.

The exception is [41], which computes a *static profiling of accumulated cost*, where the results are also parameterized by input data sizes. However, that approach is based on a global program transformation and, as mentioned before, has significant shortcomings with respect to our work: it cannot deal with non-deterministic programs that produce multiple solutions, the transformation is complex (and, because it is global, problematic in the context of modular compilation), it complicates the inference of more than one resource by requiring multiple transformations, it is less efficient and more complicated to integrate with the compiler, and it is less flexible, since it is specific to profiling.

Static profiling in the context of Worst Case Execution Time (WCET) Analysis of real-time programs is presented in [18]. It proposes an approach to computing worst-case timing information for all code parts of a program using a complementary metric, called *criticality*. Every statement of a real-time program is assigned with a criticality value, expressing how critical the respective code is for the global WCET. Our approach is not limited to WCET, since it is able to obtain results for a general class of *user-defined* resources. Furthermore, our inferred metrics are parametric on the input data sizes of the main program, in contrast to the *criticality* metric, which is a numeric value in the range $[0, 1]$. In addition, our approach is modular and compositional, able to compute accumulated costs w.r.t. calls originating from different procedures of the program, and not only the main program entry point. In [16] the authors present static profiling techniques to estimate the execution *likelihood* and *frequency* of program points in order to assess whether the cost of certain compile-time optimizations would pay off. To this end, they explore the use of some static analysis techniques for predicting the result of conditional branches, such as assuming uniform distribution over all branches, making heuristic based predictions, and performing value range propagation. In this context, our approach can be used to infer bounds on the number of times a certain program point will be called from a given entry point, as functions on input data sizes, in contrast with a single value representing the execution likelihood or frequency. Besides, since our techniques are supported mainly by the theory of abstract interpretation, the approximations inferred are *correct* by design.

3.7 Conclusions

We presented a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of resource

usage analyses, including both *accumulated cost* and standard cost. We have also reported on an implementation of this general framework within the CiaoPP system, and its instantiation for accumulated cost, and provided some experimental results. The results show that the resulting accumulated cost analysis, in addition to providing results for non-deterministic programs, is also more efficient than our previous approach based on program transformation, and has a good number of additional advantages. We argue that our approach is quite general, and it can be easily applied to other paradigms, including imperative programs, functional programs, CHR, etc., using the strategy based on compilation to Horn Clauses, as in our previous work with Java or XC.

4 Analysis of Parallel Programs

4.1 Introduction

In this chapter, we further generalize and extend the framework for resource usage analysis present in the CiaoPP system, described in Chapter 2, to be also parametric with respect to execution models (e.g., sequential or parallel execution).

Due to the heat generation barrier in traditional sequential architectures, parallel computing with (heterogeneous) architectures has become the dominant paradigm in current computer architecture. In the past decades, electronic components used to build computers have decreased in size and cost, doubling the number of transistors on a single chip every two years approximately (Moore's law [78]).

With more transistors on a single chip, data paths became shorter, allowing greater clock speeds and in this way increasing the rate at which instructions were issued. This was a key source of performance improvement during the 90's, in combination with instruction level parallelism (superscalar architectures). Thus, Software engineers' main concern was to develop more software, while hardware engineers developed better processors where the same software could run faster. However, this approach stumbled upon the power consumption barrier. While the performance of processors was growing linearly, the power consumption and heat emission of these chips was raising much more faster [13]. To overcome this limitation, new architectures appeared, where multiple processing cores with shared-memory are configured on a chip. With lower clock frequencies and less voltage, multi-core processors require less power while maintain and improve the throughput. Multi-core architectures are present everywhere, from data-centers to mobiles and small embedded devices. As a consequence of this change, it becomes vital that software be able to make effective use of the parallelism that is present now in hardware. In order to achieve this, programmers need to expose the concurrency of programs. Thus, languages and tools for parallel programming are key nowadays, and their importance is raising.

Many languages and language extensions have been developed for parallel programming. An extensively used framework is OpenMP [2]. OpenMP is a shared-memory application programming interface (API). OpenMP is suitable for developing on a broad range of shared-memory architectures. Rather than being a new new programming language, OpenMP is mostly used through directives added to a sequential program in Fortran, C, or C++. Much of the work with OpenMP consists in taking a sequential piece of code, detecting the computing intensive hot-spots in it, and then optimize those parts by exploiting their parallelism. In order to help in this process, profilers are used extensively.

MPI (Message-Passing Interface) is a message-passing library interface specification [1]. With tools implementing this specification, such as OpenMPI [35], developers can build parallel programs following the message-passing parallel programming model, in which data is moved across different processors with possibly different memory spaces through cooperative operations. MPI is usually combined with OpenMP, exploiting intra-node parallelism with OpenMP and inter-node parallelism with MPI.

GPGPU, short for general-purpose computation on graphics processing units, is another parallel approach to accelerate the execution of scientific applications, by taking advantage of the GPU's massively-parallel architectures. The CUDA programming model [3], from NVIDIA, was created for developing applications that delegate part of their computations to GPU cards. This kind of applications moves data back and forth in different memory spaces over a bus.

To this end, predicting resource usage on parallel platforms poses important challenges. It is not surprising that most work on static resource analysis has focused on sequential programs, and relatively little progress has been made on the analysis of parallel programs, or on parallel logic programs in particular. The significant body of work on static analysis of sequential logic programs has already been applied to the analysis of other programming paradigms, including imperative programs. This is achieved via a transformation into *Horn clauses* [72]. In this chapter we concentrate on the analysis of parallel Horn clause programs, which could be the result of such a translation from a parallel imperative program or be themselves the source program. Our starting point is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes [6, 25, 26, 27, 84, 98, 102, 117], which are then solved to obtain (exact or safely approximated) closed-forms of such functions (i.e., functions that provide upper or lower bounds on resource usage). We build on this and propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing static resource usage analysis of parallel logic programs for a wide range of resources, platforms, and execution models. Such an analysis estimates both lower and upper bounds on the resource usage of a parallel program as functions on input data sizes. We have instantiated the framework for dealing with Independent And-Parallelism (IAP) [40, 45], which refers to the parallel execution of conjuncts in a goal.

However, the results can be applied to other languages and types of parallelism, by performing suitable transformations into Horn clauses.

The main contributions presented in this chapter can be summarized as follows:

- We have extended a general static analysis framework for the analysis of sequential Horn clause programs [84, 102], to deal with parallel programs.
- Our extensions and further generalizations support a wide range of resources, platforms, and parallel/distributed execution models, and allow the inference of both lower and upper bounds on resource usage. This is the first approach, to our knowledge, to the cost analysis of *parallel logic programs* that can deal with features such as backtracking, multiple solutions (i.e., non-determinism), and failure.
- We have instantiated the developed framework to infer useful information for assessing and exploiting the potential and actual parallelism of a system.
- We have developed a prototype implementation that instantiates the framework for the analysis of logic programs with Independent And-Parallelism within the CiaoPP system [48, 84, 102], and provided some experimental results.

The rest of the chapter is organized as follows: in Section 4.2 we introduce some preliminary concepts and give an overview of our approach through a running example. In Section 4.3 we present our extensions to the general framework, to deal with Independent And-parallel logic programs. In Section 4.4 we describe our implementation of the techniques using the CiaoPP program analysis framework, and discuss about the experiments performed. In Section 4.5 we discuss related work. Finally, in Section 4.6 we present our conclusions.

4.2 Preliminaries

In this section we present preliminary concepts that are used in the rest of the chapter. Independent And-Parallelism refers to the case where the execution of two goals do not affect each other, and thus can be executed in parallel without altering the functional behaviour of the program. A sufficient condition for independence (assuming side-effect free execution) is the absence of variable sharing at run-time among the goals. The binary operator `&/2` is used to indicate the parallel execution of its operands. Concretely, `p & q` triggers the execution of the goals `p` and `q` in parallel, finishing when both executions finish.

4.2.1 Cost Metrics for Parallel Programs

We now enumerate different metrics used to evaluate the performance of parallel logic programs, compared against its corresponding sequential version [103]. Here,

these metrics are parameterized with respect to the resource in which the cost is expressed (e.g., number of *resolution steps*, *execution time*, or *energy consumption*):

- **Sequential cost (*Work*)**: It is the standard cost of executing a program, assuming no parallelism.
- **Parallel cost (*Depth*)**: It is the cost of executing a program in parallel, considering an unbounded number of processors.
- **Maximum number of processes running in parallel**: It is the maximum number of processes that may run simultaneously in a program. This is useful to determine what is the minimum number of processors that are required to guarantee that all the processes run in parallel.

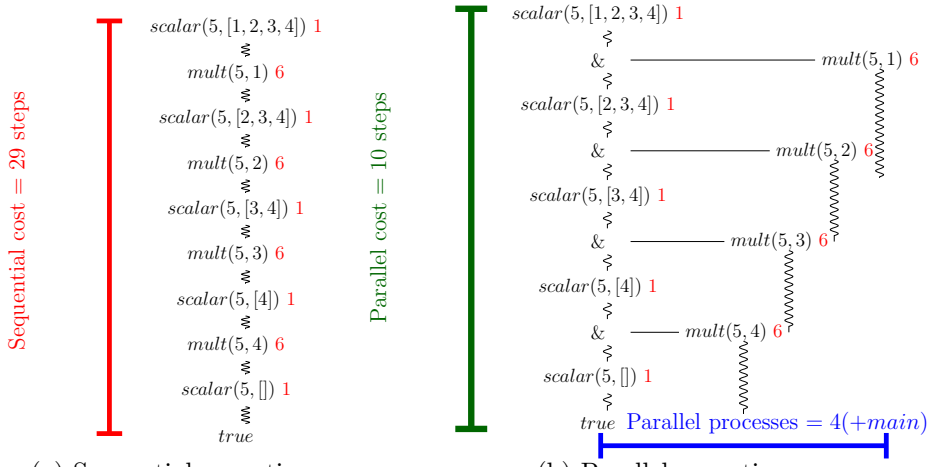
4.2.2 Overview

The following example illustrates our approach.

Example 3. *Consider the predicate* `scalar/3` *below, and a calling mode to it with the first argument bound to an integer* n *and the second one bound to a list of integers* $[x_1, x_2, \ldots, x_k]$*. Upon success, the third argument is bound to the list of products* $[n \cdot x_1, n \cdot x_2, \ldots, n \cdot x_k]$*. Each product is recursively computed by predicate* `mult/3`*. The calling modes are automatically inferred by CiaoPP (see [48] and its references): the first two arguments of both predicates are input, and their last arguments are output.*

```
1  scalar(_,[],[]).
2  scalar(N,[X|Xs],[Y|Ys]):-
3      mult(N,X,Y) & scalar(N,Xs,Ys).
4
5  mult(0,_,0).
6  mult(N,X,Y):-
7      N>1,
8      N1 is N - 1,
9      mult(N1,X,Y0),
10     Y is Y0 + X.
```

The call to the parallel `&/2` *operator in the body of the second clause of* `scalar/3` *causes the calls to* `mult/3` *and* `scalar/3` *to be executed in parallel. We want to infer the cost of such a call to* `scalar/3`*, in terms of the number of resolution steps, as a function of its input data sizes. We use the CiaoPP system to infer size relations for the different arguments in the clauses, as well as dealing with a rich set of size metrics (see [84, 102] for details). Assume that the size metrics used in this example are the* actual value *of* `N` *(denoted* `int(N)`*), for the first argument, and the* list-length *for the second and third arguments (denoted* `length(X)` *and* `length(Y)`*, respectively). Since size relations are obvious in this example, we*



(a) Sequential execution (b) Parallel execution

Figure 4.1: Resolutions steps performed by the call $scalar(5, [1, 2, 3, 4])$, considering different execution models.

focus only on the setting up of cost relations for the sake of brevity. Regarding the number of solutions, in this example all the predicates generate at most one solution. For simplicity we assume that all builtin predicates, such as `is/2` *and the comparison operators have zero cost (in practice they have a "trust"assertion that specifies their cost as if it had been inferred by the system). As the program contains parallel calls, we are interested in inferring both total resolution steps, i.e., considering a sequential execution (represented by the* `seq` *identifier), and the number of parallel steps, considering a parallel execution, with an unbounded number of processors (represented by* `par`*). In the latter case, the definition of this resource establishes that the aggregator of the costs of the parallel calls that are arguments of the* `&/2` *meta-predicate is the* `max/2` *function. Thus, the number of resolution steps performed in parallel for* `p & q` *is the maximum between the parallel steps performed by* `p` *and the ones performed by* `q`*. However, for computing the* total resolution steps*, the aggregation operator we use is the addition, both for parallel and sequential calls. For brevity, in this example we only infer upper bounds on resource usages. In Figure 4.1 we show graphically the actual cost in terms of resolution steps of the goal* $scalar(5, [1, 2, 3, 4])$*, considering a sequential and a parallel execution.*

We now set up the cost relations for `scalar/3` *and* `mult/3`*. Note that the cost functions have two arguments, corresponding to the sizes of the input arguments.*[1] *In the equations, we underline the operation applied as cost aggregator for* `&/2`*.*

For the sequential execution (`seq`*), we obtain the following cost relations:*

[1]For the sake of clarity, we abuse notation in the examples when representing the cost functions that depend on data sizes.

$$\begin{array}{rcll} \mathtt{C}_{\mathtt{scalar}}(n,l) = & 1 & & \textit{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = & 1 + \mathtt{C}_{\mathtt{mult}}(n) \underline{+} \mathtt{C}_{\mathtt{scalar}}(n,l-1) & & \textit{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & 1 & & \textit{if } n = 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & 1 + \mathtt{C}_{\mathtt{mult}}(n-1) & & \textit{if } n > 0 \end{array}$$

After solving these equations and composing the closed-form solutions, we obtain the following closed-form functions:

$$\begin{array}{rcll} \mathtt{C}_{\mathtt{scalar}}(n,l) = & (n+2) \times l + 1 & & \textit{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & n + 1 & & \textit{if } n \geq 0 \end{array}$$

For the parallel execution (`&`), we obtain the following cost relations:

$$\begin{array}{rcll} \mathtt{C}_{\mathtt{scalar}}(n,l) = & 1 & & \textit{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = & 1 + \underline{\underline{max}}(\mathtt{C}_{\mathtt{mult}}(n), \mathtt{C}_{\mathtt{scalar}}(n,l-1)) & & \textit{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & 1 & & \textit{if } n = 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & 1 + \mathtt{C}_{\mathtt{mult}}(n-1) & & \textit{if } n > 0 \end{array}$$

Similarly, we obtain the following closed-form functions:

$$\begin{array}{rcll} \mathtt{C}_{\mathtt{scalar}}(n,l) = & n + l + 1 & & \textit{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = & n + 1 & & \textit{if } n \geq 0 \end{array}$$

By comparing the complexity order (in terms of resolution steps) of the sequential execution of `scalar/3`, $O(n \cdot l)$, with the complexity order of its parallel execution (assuming an ideal parallel model with an unbounded number of processors) $O(n+l)$, we can get a hint about the maximum achievable parallelization of the program.

Another useful piece of information about `scalar/3` that we want to infer is the maximum number of processes that may run in parallel, considering all possible executions. For this purpose, we define a resource named `sthreads`. The operation `count_process/3` aggregates the cost of both arguments of the meta-predicate `&/2` for the `sthreads` resource, by adding the maximum number of processes for each argument plus one additional process, corresponding to the one created by the call to `&/2`. The sequential cost aggregator is now the maximum *operator, in order to keep track of the maximum number of processes created along the different instructions of the program executed sequentially. Note that if the instruction `p` executes at most Pr_p processes in parallel, and the instruction `q` executes at most Pr_q processes, then the program `p, q` will execute at most $max(Pr_p, Pr_q)$ processes in parallel, because all the parallel processes created by `p` will finish before the execution of `q`. Note also that for the sequential execution of both `p` and `q`, the cost in terms of the `sthreads` resource is always zero, because no additional process is created. The analysis sets up the following recurrences for the `sthreads` resource and the predicates `scalar/3` and `mult/3` of our example:*

$$\begin{aligned} \mathtt{C}_{\mathtt{scalar}}(n,l) &= 0 && \text{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) &= \mathtt{C}_{\mathtt{mult}}(n) + \mathtt{C}_{\mathtt{scalar}}(n,l-1) + 1 && \text{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) &= 0 && \text{if } n \geq 0 \end{aligned}$$

For which we obtain the following closed-form functions:

$$\begin{aligned} \mathtt{C}_{\mathtt{scalar}}(n,l) &= l && \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) &= 0 && \text{if } n \geq 0 \end{aligned}$$

As we can see, this predicate will execute, in the worst case, as many processes as there are elements in the input list.

4.3 Our Extended Resource Analysis Framework for Parallel Programs

In this section, we describe how we extend the resource analysis framework detailed in Chapter 2, in order to handle logic programs with Independent And-Parallelism, using the binary parallel `&/2` operator. First, we introduce a new general parameter that indicates the execution model the analysis has to consider. For our current prototype, we have defined two different execution models: standard *sequential* execution, represented by *seq*, and an abstract parallel execution model, represented by $par(n)$, where $n \in \mathcal{N} \cup \{\infty\}$. The abstract execution model $par(\infty)$ is similar to the *work* and *depth* model, presented in [15] and used extensively in previous work such as [51]. Basically, this model is based on considering an unbounded number of available processors to infer bounds on the depth of the computation tree. The *work* measure is the amount of work to be performed considering a sequential execution. These two measures together give an idea on the impact of the parallelization of a particular program. The abstract execution model $par(n)$, where $n \in \mathcal{N}$, assumes a finite number n of processors.

In order to obtain the cost of a predicate, equation (2.1) remains almost identical, the only difference being the addition of the new parameter to indicate the execution model.

Now we address how to set up the cost for clauses. In this case, equation (2.2) is extended with the execution model ex, and also the default sequential cost aggregation, $\sum$, is replaced by a parametric associative operator $\bigoplus$, that depends on the resource being defined, the approximation, and the execution model. For $ex \equiv par(\infty)$ or $ex \equiv seq$, the following equation is set up:

$$C_{cl[ap,r,ex]}(C,\bar{\mathtt{x}}) = \varphi_{[ap,r]}(H) + \bigoplus_{j=1}^{lim(ap,ex,C)} (sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r,ex]}(L_j, \psi_j(\bar{\mathtt{x}}))) \quad (4.1)$$

Note that the cost aggregation operators must depend on the resource r (besides the other parameters). For example, if r is *execution time*, then the cost of executing two tasks in parallel must be aggregated by taking the maximum of the execution times of the two tasks. In contrast, if r is *energy consumption*, then the aggregation is the addition of the energy of the two tasks.

Finally, we extend how the cost of a literal L_i, expressed as $C_{lit[ap,r,ex]}(L_i, \psi_i(\bar{\mathtt{x}}))$, is set up. The previous definition is extended considering the new case where the literal is a call to the *meta-predicate* `&/2`. In this case, we introduce a new parallel aggregation associative operator, denoted by $\bigotimes$. Concretely, if $L_i = B_1 \& B_2$, where B_1 and B_2 are two sequences of goals, then:

$$C_{lit[ap,r,ex]}(B_1 \& B_2, \bar{\mathtt{x}}) = C_{body[ap,r,ex]}(B_1, \bar{\mathtt{x}}) \bigotimes C_{body[ap,r,ex]}(B_2, \bar{\mathtt{x}}) \tag{4.2}$$

$$C_{body[ap,r,ex]}(B, \bar{\mathtt{x}}) = \bigoplus_{j=1}^{lim(ap,ex,B)} (sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r,ex]}(L_j^B, \psi_j(\bar{\mathtt{x}}))) \tag{4.3}$$

where $B = L_1^B, \ldots, L_m^B$.

Consider now the execution model $ex \equiv par(n)$, where $n \in \mathcal{N}$ (i.e., assuming a finite number n of processors), and a recursive parallel predicate `p` that creates a parallel task $\mathtt{q}_i$ in each recursion i. Assume that we are interested in obtaining an upper bound on the cost of a call to `p`, for an input of size $\bar{\mathtt{x}}$. We first infer the number k of parallel tasks created by `p` as a function of $\bar{\mathtt{x}}$. This can be easily done by using our cost analysis framework and providing the suitable assertions for inferring a resource named "*ptasks*." Intuitively, the "counter" associated to such resource must be incremented by the (symbolic) execution of the `&/2` parallel operator. More formally, $k = C_{pred[ub,ptasks]}(\mathtt{p}, \bar{\mathtt{x}})$. To this point, an upper bound m on the number of tasks executed by any of the n processors is given by $m = \lceil \frac{k}{n} \rceil$. Then, an upper bound on the cost (in terms of resolution steps, i.e., $r = steps$) of a call to `p`, for an input of size $\bar{\mathtt{x}}$ can be given by:

$$C_{pred[ub,r,par(n)]}(\mathtt{p}, \bar{\mathtt{x}}) = C^u + Spawn^u \tag{4.4}$$

where C^u can be computed in two possible ways: $C^u = \sum_{i=1}^{m} C_i^u$; or $C^u = m\ C_1^u$, where C_i^u denotes an upper bound on the cost of parallel task $\mathtt{q}_i$, and $C_1^u, \ldots, C_k^u$ are ordered in descending order of cost. Each C_i^u can be considered as the sum of two components: $C_i^u = Sched_i^u + T_i^u$, where $Sched_i^u$ denotes the cost from the point in which the parallel subtask $\mathtt{q}_i$ 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. T_i^u denotes the cost of the execution of $\mathtt{q}_i$ disregarding all the overheads mentioned before, i.e., $T_i^u = C_{pred[ub,r,seq]}(\mathtt{q}, \psi_q(\bar{\mathtt{x}}))$, where $\psi_q(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to predicate `q` in the body of `p`. $Spawn^u$ denotes an upper bound on

Table 4.1: Description of the benchmarks.

`map_add1/2`	Parallel increment by one of each element of a list.
`fib/2`	Parallel computation of the nth Fibonacci number.
`add_mat/3`, `mmatrix/3`	Parallel matrix multiplication and addition.
`blur/2`	Generic parallel image filter.
`intersect/3`, `union/3`, `diff/3`	Set operations.
`dyade/3`, `dyade_map/3`	Dyadic product of two vectors (and on a set of vectors).
`append_all/3`	Appends a prefix to each list of a list of lists.

the cost of creating the k parallel tasks $\mathtt{q}_i$. It will be dependent on the particular system in which `p` 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, or number of tasks) and can be experimentally determined.

4.4 Implementation and Experimental Results

We have implemented a prototype of our approach, leveraging the existing resource usage analysis framework of CiaoPP. The implementation basically consists of the parameterization of the operators used for sequential and parallel cost aggregation, i.e., for the aggregation of the costs corresponding to the arguments of `,/2` and `&/2`, respectively. This allows the user to define resources in a general way, taking into account the underlying execution model.

We selected a set of benchmarks that exhibit different common parallel patterns, briefly described in Table 4.1, together with the definition of a set of resources that help understand the overall behavior of the parallelization. Table 4.2 shows some results of the experiments that we have performed with our prototype implementation. Column **Bench** shows the main predicates analyzed for each benchmark. Set operations (`intersect`, `union` and `diff`), as well as the programs `append_all`, `dyade` and `add_mat`, are Prolog versions of the benchmarks analyzed in [51], which is the closest related work we are aware of. Column **Res** indicates the name of each of the resources inferred for each benchmark: *sequential resolution steps* (**SCost**), *parallel resolution steps* assuming an unbounded number of processors (**PCost**), and *maximum number of processes executing in parallel* (**SThreads**). The latter gives an indication of the maximum parallelism that can potentially be exploited. We are considering a resolution step as the overhead of spawning a new thread. Column **Bound Inferred** shows the upper

Table 4.2: Resource usage inferred for Independent And-Parallel Programs.

Bench	Res	Bound Inferred	Acc.BigO	$\mathbf{T_A}$(ms)
`map_add1(x)`	**SCost**	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	31.17
	PCost	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	
	SThreads	l_x	$\mathcal{O}(l_x)$	
`fib(x)`	**SCost**	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	127.81
	PCost	$2 \cdot i_x + 1$	$\mathcal{O}(i_x)$	
	SThreads	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	
`mmatrix`(m_1, n_1, m_2, n_2)	**SCost**	$i_{n_2} \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_1} + 1$	$\mathcal{O}(i_{n_2} \cdot i_{m_2} \cdot i_{m_1})$	194.45
	PCost	$i_{n_2} + 2 \cdot i_{m_2} + 2 \cdot i_{m_1} + 1$	$\mathcal{O}(i_{n_1} + i_{m_1} + i_{m_2})$	
	SThreads	$i_{m_2} \cdot i_{m_1} + i_{m_1}$	$\mathcal{O}(i_{m_2} \cdot i_{m_1})$	
`blur(m,n)`	**SCost**	$2 \cdot i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	126.63
	PCost	$2 \cdot i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	SThreads	i_n	$\mathcal{O}(i_n)$	
`add_mat(m,n)`	**SCost**	$i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	128.93
	PCost	$i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	SThreads	i_n	$\mathcal{O}(i_n)$	
`intersect`(a, b)	**SCost**	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	233.14
	PCost	$l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
`union`(a, b)	**SCost**	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	218.31
	PCost	$2 \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
`diff`(a, b)	**SCost**	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	232.55
	PCost	$l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
`dyade`(a, b)	**SCost**	$l_a \cdot l_b + 2 \cdot l_a + 1$	$\mathcal{O}(l_a \cdot l_b)$	82.71
	PCost	$l_b + 2 \cdot l_a + 1$	$\mathcal{O}(l_a + l_b)$	
	SThreads	l_a	$\mathcal{O}(l_a)$	
`dyade_map`(l, m)	**SCost**	$i_{max(m)} \cdot l_m \cdot l_l + 2 \cdot l_m \cdot l_l + 2 \cdot l_m + 1$	$\mathcal{O}(i_{max(m)} \cdot l_m \cdot l_l)$	177.91
	PCost	$i_{max(m)} + 2 \cdot l_m + 2 \cdot l_l + 1$	$\mathcal{O}(i_{max(m)} + l_m + l_l)$	
	SThreads	$l_l \cdot l_m + l_l$	$\mathcal{O}(l_m \cdot l_l)$	
`append_all`(l, m)	**SCost**	$l_l \cdot l_m + 2 \cdot l_m + 1$	$\mathcal{O}(l_l \cdot l_m)$	81.97
	PCost	$l_l + 2 \cdot l_m + 1$	$\mathcal{O}(l_l + l_m)$	
	SThreads	l_m	$\mathcal{O}(l_m)$	

$F(n)$, $L(n)$ represent the nth. element of the Fibonacci sequence and the nth. Lucas number, respectively. l_n, i_n represent the size of n in terms of the metrics *length* and *int*, respectively.

bounds obtained for each of the resources indicated in Column **Res**. While in the experiments both upper and lower bounds were inferred, for the sake of brevity, we only show upper-bound functions. Column **Acc.BigO** shows the complexity order, in big O notation, corresponding to each resource. For all the benchmarks in Table 4.2 we obtain the exact complexity orders. We also obtain the same complexity order as in [51] for the Prolog versions of the benchmarks taken from that work. Finally, Column $\mathbf{T_A}$**(ms)** shows the analysis times in milliseconds. The results show that most of the benchmarks have different asymptotic behavior in the sequential and parallel execution models. In particular, for `fib(x)`, the analysis infers an exponential upper bound for sequential execution steps, and a linear upper bound for parallel execution steps. As mentioned before, this is an upper bound for an ideal case, assuming an unbounded number of processors. Nevertheless, such upper-bound information is useful for understanding how the cost behavior evolves in architectures with different levels of parallelism. In

Table 4.3: Resource usage inferred for a bounded number of processors.

Bench	Bound Inferred	Acc.BigO	T_A(ms)
`map_add1(x)`	$2 \cdot \lceil \frac{l_x}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_x}{p} \rceil)$	54.36
`blur(m,n)`	$2 \cdot \lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	205.97
`add_mat(m,n)`	$\lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	185.89
`intersect`(a,b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	330.47
`union`(a,b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + l_b + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	311.3
`diff`(a,b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	339.01
`dyade`(a,b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	120.93
`append_all`(l,m)	$\lceil \frac{l_m}{p} \rceil \cdot l_l + 2 \cdot \lceil \frac{l_m}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_m}{p} \rceil \cdot l_l)$	117.8

p is defined as the minimum between the number of processors and **SThreads**.

addition, this *dual* cost measure can be combined together with a bound on the number of processors in order to obtain a general asymptotic upper bound (see for example Brent's Theorem [43], which is also mentioned in [51]). The program `map_add1(l)` exhibits a different behavior: both sequential and parallel upper bounds are linear. This happens because we are considering *resolution steps*, i.e., we are counting each head unification produced from an initial call `map_add1(l)`. Even under the parallel execution model, we have a chain of head unifications whose length depends linearly on the length of the input list. It follows from the results of this particular case that this simple, non-associative parallelization will not be useful for improving the number of resolution steps performed in parallel.

Another useful information inferred in our experiments is the maximum number of processes that can be executed in parallel, represented by the resource named **SThreads**. We can see that for most of our examples the analysis obtains a linear upper bound for this resource, in terms of the size of some of the inputs. For example, the execution of `intersect(a,b)` (parallel set intersection) will create *at most* l_a processes, where l_a represents the length of the list a. For other examples, the analysis shows a quadratic upper bound (as in `mmatrix`), or even exponential bounds (as in `fib`). The information about upper bounds on the maximum level of parallelism required by a program is useful for understanding its scalability in different parallel architectures, or for optimizing the number of processors that a particular call will use, depending on the size of the input data.

Finally, the results of our experiments considering a bounded number of processors are shown in Table 4.3.

4.5 Related Work

Our approach is an extension of an existing cost analysis framework for sequential logic programs [27, 41, 69], which extends the classical cost analysis techniques based on setting up and solving recurrence relations, pioneered by [117], with solutions for recurrences involving maximization. The framework also handles characteristics such as backtracking, non-determinism, and failure. Furthermore,

it is able to infer both upper and lower bounds as general expressions, not limited to linear or polynomial expressions. These features are inherited by our approach, and are absent from other approaches to parallel cost analysis in the literature.

In the context of functional programming, the work presented in [51] describes an automatic analysis for the inference of bounds on the worst-case evaluation cost of first order functional programs. Similar to what we presented in the thesis, this analysis derives bounds under a cost model based on two measures: *work* and *depth*, over-approximating the sequential and parallel evaluation cost of functional programs, respectively, considering an unlimited number of processors. The analysis is based on type judgments annotated with a cost metric, which generate a set of inequalities which are then solved by linear programming techniques. However, their analysis is only able to infer multivariate resource polynomial bounds, while non-polynomial bounds are left as future work. The cost model based on *work* and *depth* metrics was introduced by [15] as a framework to formally analyze parallel programs.

In [49] the authors propose an automatic analysis also based on *work* and *depth* model, for a simple imperative language with explicit parallel loops.

There are other approaches to cost analysis of parallel and distributed systems, based on different models of computation than the independent and-parallel model in our work. In [9] the authors present a static analysis which is able to infer upper bounds on the maximum number of *active* (i.e., not finished nor suspended) processes running in parallel, and the total number of processes created for imperative *async-finish* parallel programs. The approach described in [4] uses recurrence (cost) relations to derive upper bounds on the cost of concurrent object-oriented programs, with shared-memory communication and future variables. They address concurrent execution for loops with semi-controlled scheduling, i.e., with no arbitrary interleavings. In [10] the authors address the cost of parallel execution of object-oriented distributed programs. The approach is to identify the synchronization points in the program, use serial cost analysis of the blocks between these points, and then, exploiting the techniques mentioned, construct a graph structure to capture the possible parallel execution of the program. The path of maximal cost is then computed. The allocation of tasks to processors (called "locations") is part of the program in these works, and thus, although independent and-parallel programs could be modeled in this computation style, it is not directly comparable to our more abstract model of parallelism.

Solving, or safely bounding recurrence relations with `max` and `min` functions has been addressed mainly for recurrences derived from divide-and-conquer algorithms [12, 53, 116]. In [6] the authors present solutions for Cost Relation Systems by obtaining upper bounds for both the number of nodes and the cost added in each node in the derived evaluation tree. These bounds are then combined in order to obtain a closed-form upper-bound expression. This closed form possibly contains maximization operations to express upper bounds for a set of subexpressions. However, each cost relation is defined as a summatory of costs, while in

our approach, in addition to summations, we also consider other operations for aggregating the costs, including `max` operators. The presence of these operators often produces recurrence relations where the recursive calls are under the scope of such a `max` operator, for which we present a method to obtain a closed-form bound. This class of recurrences are not handled by most of the current computer algebra systems, as the authors in [6] mention.

4.6 Conclusions

We have presented a novel, general, and flexible analysis framework that can be instantiated for estimating the resource usage of parallel logic programs, for a wide range of resources, platforms, and execution models. To the best of our knowledge, this is the first approach to the cost analysis of *parallel logic programs*. Such estimations include both lower and upper bounds, given as functions on input data sizes. In addition, our analysis also infers other information which is useful for improving the exploitation and assessing the potential and actual parallelism of a program. Finally, we have developed a prototype implementation of our general framework, instantiated it for the analysis of logic programs with Independent And-Parallelism, and performed an experimental evaluation, obtaining encouraging results w.r.t. accuracy and efficiency.

5 Recurrence Solver Extensions

5.1 Introduction and Motivation

As already said, the traditional approach to static resource usage analysis is based on setting up recurrence relations representing the size and computational cost of procedures, depending on the input data sizes. Solving such relations means to obtain closed-form expressions, which can be exact representations or approximations of the initial recurrences. Automatically finding closed-form upper and lower bounds for recurrence relations is an uncomputable problem. For some special classes of recurrences, exact solutions are known, for example for linear recurrences with one variable. For some other classes, it is possible to apply transformations to fit into a class of recurrences with known solutions. Such transformation can be useful even if they obtain an appropriate approximation rather than an equivalent expression.

A common approach to automatually solving the size and cost relations that arise during program analysis consists on expressing them as mathematical recurrences and using a Computer Algebra System (CAS) or a specialized solver to find a closed-form. However, this approach poses several difficulties. For example, some recurrences may not have the form required by such systems because an input data size variable does not decreases, but increases instead. Nevertheless, the decreasing variable can be implicit in the program, i.e., it is actually a function of the input data sizes (a ranking function), which can be inferred by applying other techniques traditionally used in termination analysis [91]. However, such techniques are usually restricted to linear arithmetics. Moreover, some recurrence relations contain complex expressions or recursive structures that most of the well-known CASs cannot solve, making necessary to develop ad-hoc techniques to handle such cases.

Our first step towards addressing these challenges is the design of a solver that follows a modular architecture (see Figure 5.1), and is able to integrate different recurrence solvers, either general or specialized for certain classes of recurrences,

following different strategies. Such solver, which is a fundamental component of our resource analysis framework, in charge of solving or bounding recurrences, has been implemented and integrated into the CiaoPP system [48]. The main modules of the architecture in Figure 5.1 are:

- *Solver_Strategies*: This module defines the common interface to the different strategies to solve recurrence relations.
- $Strat_i$: These are the different strategies to solve or over-approximate recurrences, using the services of the back-end solvers and the *classifier* in order to identify different characteristics of the recurrences.
- *Rec_Classifier*: It associates a *label* to each input recurrence relation that identifies the class of recurrence.
- *Solver_Utils*: Defines the common interface to the different *back-end solvers*.
- BS_i: These are the modules that implement the interface defined by **Solver_Utils**, connecting directly with the particular back-end solvers, such as Mathematica®, CiaoPP's built-in solver, etc.

Our second step, which we describe in detail in this chapter, consists on extending our modular solver by developing novel methods for solving arbitrary, constrained recurrence relations, and implementing and integrating them into the modular architecture depicted in Figure 5.1 as specialized solvers (BS_i).

The rest of this chapter is organized as follows. In Section 5.2 we present our specialized solver based on machine learning that performs linear regression. In Section 5.3 we propose a method for solving a subclass of recurrences that include a maximization operator. Finally, in Section 5.4 we give our conclusions.

5.2 Solving Recurrence Relations using Linear Regression

As already said, we have designed and implemented a specialized backend solver that follows a *guess and check* approach, by using well-known machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a CAS for the *check* phase (see Figure 5.2). We illustrate with a set of examples how this approach is useful for improving the accuracy and applicability of static cost analysis.

5.2.1 Overview of the Approach

We now give an overview of our approach, illustrating it with an example. We first show how the CiaoPP system sets up recurrence relations representing the

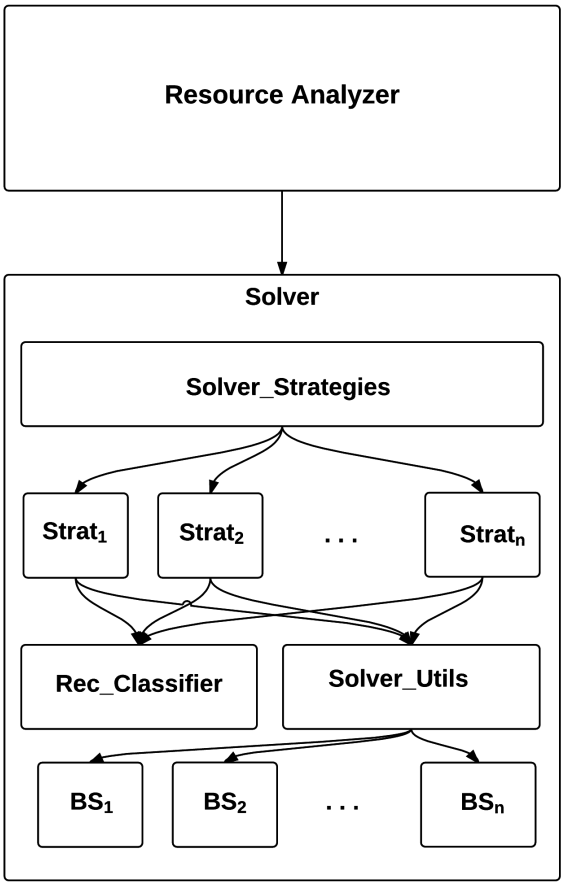


Figure 5.1: Architecture of the modular solver framework.

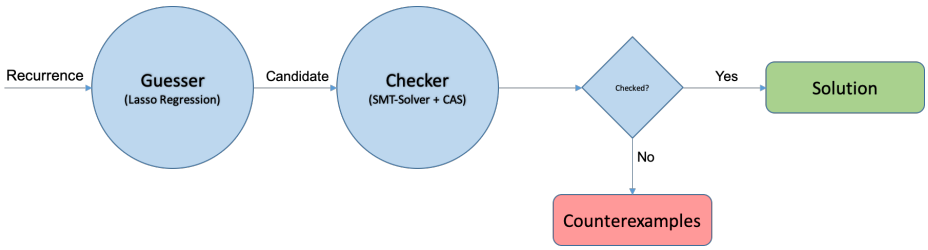


Figure 5.2: Control flow diagram of our novel solver based on machine learning.

sizes of output arguments of procedures, as well as the cost of such procedures. Then, we show how our novel approach is used to solve such recurrence relations.

Example 4. *Consider predicate* `p/2` *in Figure 5.3, for calls where the first argument is bound to a non-negative integer and the second one is a free variable. Upon success of these calls, the second argument is bound to an non-negative integer too, which is the result of the computation. Such calling mode, where the*

first argument is input and the second one is output, is automatically inferred by CiaoPP (see [48] and its references).

```
1 :- entry p/2: nnegint*var.
2 p(X,0):-
3   X==0.
4 p(X,Y):-
5   X>0,
6   X1 is X - 1,
7   p(X1,Y1),
8   p(Y1,Y2),
9   Y is Y2 + 1.
```

Figure 5.3: A program with a nested recursion.

The CiaoPP system first infers size relations for the different arguments of predicates, using a rich set of size metrics (see [84, 102] for details). Assume that the size metric used in this example for a numeric argument `X` is the *actual value* of it (denoted `int(X)`). The system will try to infer a function $\mathtt{S_p}(x)$ that gives the size of the output argument of `p/2` (the second one), as a function of the size (x) of the input argument (the first one). For this purpose, the following size relations for $\mathtt{S_p}(x)$ are automatically set up:

$$\begin{array}{lll} \mathtt{S_p}(x) = & 0 & \text{if } x = 0 \\ \mathtt{S_p}(x) = & \mathtt{S_p}(\mathtt{S_p}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{5.1}$$

which, for this example, represent a nested recurrence. Once the size relations have been inferred, CiaoPP uses them to infer the cost of a call to `p/2`. For simplicity, assume that for this example, such cost is given in terms of the number of *resolution steps*, as a function of the size of the input argument, but recall that CiaoPP's cost analysis can infer a wide range of resources besides resolution steps, since it is parametric with respect to resources, which can be defined by the user by means of a rich assertion language. Also for simplicity, we assume that all builtin predicates, such as `is/2` and the comparison operators have zero cost (in practice there is a "trust"assertion for each builtin that specifies its cost as if it had been inferred by the analysis).

In order to infer the cost function $\mathtt{C_p}(x)$ for `p/2`, CiaoPP sets up the following cost relations:

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & \mathtt{C_p}(x-1) + \mathtt{C_p}(\mathtt{S_p}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{5.2}$$

For some applications, it is enough working with both size and cost functions expressed as recurrences, as they can be evaluated for a given input data size of interest, and the result (a numeric constant) can be used in the appropriate

way. However, for other applications we are interested in finding closed-forms for such functions, i.e., solving such recurrences (or at least finding approximated solutions). Assume that we have a recurrence relation for a function $f(\bar{x})$ (either a size of cost function). Ideally, we want to find a closed-form function $\hat{f}(\bar{x})$ such that $Dom(\hat{f}) = Dom(f)$, and $\forall \bar{x} \in Dom(f)(\hat{f}(\bar{x}) = f(\bar{x}))$. If this is not possible $\hat{f}(\bar{x})$ can be an approximation of $f(\bar{x})$.

In our example, the cost of the second recursive call to `p/2` depends on the size of the output argument of the first recursive call to `p/2`, which is given by another function, $\mathtt{S_p}(x)$. Thus, in order to find an equivalent closed-form for $\mathtt{C_p}(x)$, it is necessary to find a closed-form for $\mathtt{S_p}(x)$ first. However, most of the state-of-the-art recurrence solvers do not handle nested recurrences, as the ones that arise in this example. As a result, CiaoPP fails to find closed-form functions for the recurrence relations above. In order to fill this gap, we propose a novel, automatic approach that uses machine-learning regression techniques to *guess* a candidate closed-form function, and a combination of an SMT-solver and a CAS to *check* if such function is actually a solution of the recurrence relation. We now give an overview of both stages.

The linear regression stage As we essentially use a linear regression mechanism, the model we obtain (which constitutes a candidate solution) is a linear combination of a set of terms, i.e., a function $\hat{f}(\bar{x})$ of the form:

$$\hat{f}(\bar{x}) = a_0 + a_1\ t_1(\bar{x}) + a_2\ t_2(\bar{x}) + \cdots + a_n\ t_n(\bar{x})$$

where the a_i's are the coefficients that are estimated by the regression method, and the t_i's are arbitrary functions on $\bar{x}$ from a set T of candidate terms that we call base functions. Such a set can be user-provided, or can be automatically inferred by using different heuristics. For example, we can start with a set containing all the base functions that are representative of the common complexity orders. Alternativelly, we can perform an automatic analysis of the recurrence we are solving, to extract some features that allow selecting the terms that most likely are part of the solution. For example, if the recurrence has a nested, double recursion, then we can select a quadratic term. Assume that for the regression in this example we use the former approach, providing the following set T of base functions:

$$T = \{\lambda x.x, \lambda x.x^2, \lambda x.x^3, \lambda x.\lceil log_2(x) \rceil, \lambda x.2^x, \lambda x.x \cdot \lceil log_2(x) \rceil\}$$

where each base function is represented as a lambda expression. Then, the linear regression is performed as follows:

1. Generate a training set S. First, a set $X_{train} = \{\bar{x}_1, \ldots, \bar{x}_k\}$ of input values to the recurrence function is randomly generated. Then, starting with an initial $S = \emptyset$, for each input value $\bar{x}_i \in X_{train}$, a training case s_i is generated

and added to S. For any input value $\bar{x} \in X_{train}$ the corresponding training case s is a tuple of the form:

$$s = \langle b, c_1, \ldots, c_n \rangle$$

where $c_i = [\![t_i]\!]_{\bar{x}}$ for $1 \leq i \leq n$, and $[\![t_i]\!]_{\bar{x}}$ represents the result (a constant number) of evaluating the base function $t_i \in T$ for input value $\bar{x}$, where T is a set of n base functions as already explained. The (dependent) value b (also a constant number) is the result of evaluating the recurrence $f(\bar{x})$ that we want to approximate, in our example, $\mathtt{S_p}(x)$, defined in equation 5.1. Following our example, where the tuples of input data in X_{train} have one element only, and assuming that there is a $\bar{x} \in X_{train}$ such that $\bar{x} = \langle 5 \rangle$, its corresponding training case s will be:

$$\begin{aligned} s &= \langle \mathbf{S_p}(\mathbf{5}), [\![x]\!]_5, [\![x^2]\!]_5, [\![x^3]\!]_5, [\![\lceil log_2(x) \rceil]\!]_5, \ldots \rangle \\ &= \langle \mathbf{5}, 5, 25, 125, 3, \ldots \rangle \end{aligned}$$

2. Perform the regression in two steps using the training set S created above. In the first step, we use linear regression with Lasso regularization. Such regression method adds a penalization to those inputs that apparently have a small correlation with the dependent value, assigning 0 to their coefficients. This way, most of the candidate terms in T will be discarded, and only those that best approximate our target function will be kept. The level of penalization is determined by using a real-valued parameter λ. Instead of using a single value, we use cross-validation to find the value of λ that achieves the best fit on S. The result of this step is a (column) vector $\bar{\beta}$ of coefficients[1], and an independent coefficient β_0. Then, we randomly generate a test set X_{test} of input values to the recurrence function (in a similar way as X_{train} was generated in the previous stage, but with different values) to obtain a measure R^2 of the accuracy of the estimation. Additionally, we discard those terms whose corresponding coefficient is less than a given threshold ϵ. The resulting closed-form expression that estimates the target function is

$$\hat{f}(\bar{x}) = rm_\epsilon(\bar{\beta}^T) \cdot E(T, \bar{x}) + \beta_0$$

where $E(T, \bar{x})$ is a vector of the terms in T with the arguments bound to $\bar{x}$, and rm_ϵ takes a vector of coefficients and returns another vector where the coefficients less than ϵ are rounded to zero. Both the Lasso regularization and the pruning function discard many terms from T in the final cost function.

Finally, our method performs again a standard linear regression (without Lasso regularization) using S, but projecting out those inputs corresponding to the

[1]We round these coefficients to two decimals, which causes that many very small coefficients become 0 as well.

terms discarded previously by Lasso and the ϵ-pruning. In our example, with $\epsilon = 0.001$, we obtain:

$$\hat{\mathtt{S}}_\mathtt{p}(x) = 1.0\ x \quad (R^2 = 1)$$

As the measure of the accuracy of the estimation, R^2, is 1 in our example, the estimation obtained predicts exactly the values for the test set, and thus, it is a candidate solution for $\mathtt{S}_\mathtt{p}$. If the final R^2 is less than 1, it means that we have not obtained a candidate solution, but an approximation.

The verification stage Once a function that is a candidate solution for the recurrence has been obtained, the second step of our method tries to verify whether such a candidate is actually a solution. To do so, the recurrence is encoded as a first order logic formula where the references to the target function are replaced by the candidate solution whenever possible. Afterwards, we use an SMT-solver to check whether the negation of such formula is satisfiable, in which case, we can conclude that the candidate is not a solution for the recurrence. Otherwise, if such formula is unsatisfiable, then the candidate function is a correct solution. Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding.

To illustrate this processs, Expression (5.3) below shows the recurrence relation we target to solve, followed by the candidate solution obtained previously using linear regression.

$$\begin{array}{lll} \mathtt{S}_\mathtt{p}(x) = & 0 & \text{if } x = 0 \\ \mathtt{S}_\mathtt{p}(x) = & \mathtt{S}_\mathtt{p}(\mathtt{S}_\mathtt{p}(x-1)) + 1 & \text{if } x > 0 \\ \hat{\mathtt{S}}_\mathtt{p}(x) = & x & \text{if } x \geq 0 \end{array} \tag{5.3}$$

Now, Expression (5.4) below shows the encoding of the recurrence as a first order logic formula.

$$\forall x \cdot \Big((x = 0 \implies \underline{\mathtt{S}_\mathtt{p}(x)} = 0) \wedge (x > 0 \implies \underline{\mathtt{S}_\mathtt{p}(x)} = \underline{\mathtt{S}_\mathtt{p}(\underline{\mathtt{S}_\mathtt{p}(x-1)})} + 1)\Big) \tag{5.4}$$

Finally, Expression (5.5) below shows the negation of such formula, as well as the references to the function name substituted by the definition of the candidate solution. We underline both the subexpressions to be replaced, and the subexpressions resulting from the substitutions.

$$\exists x \cdot \neg(((x = 0 \implies \underline{x} = 0) \wedge (x > 0 \implies \underline{x} = \underline{x - 1} + 1))) \tag{5.5}$$

It is easy to see that Formula (5.5) is unsatisfiable. Therefore, $\hat{\mathtt{S}}_\mathtt{p}(x) = x$ is a exact solution for $\mathtt{S}_\mathtt{p}(x)$. Finally, we can use this result to set up and solve the recurrence for $\mathtt{C}_\mathtt{p}(x)$, obtaining $\hat{\mathtt{C}}_\mathtt{p}(x) = 2^{x+1} - 1$.

For some cases where the candidate solution contains transcendental functions, our implementation of the method uses a Computer Algebra System to perform

simplifications and transformations, in order to obtain a formula supported by the SMT-solver. We find this combination of CAS and SMT-solver particularly useful, since it allows solving more problems than only using one of these systems in isolation.

5.2.2 Preliminaries

We use the last letters from the alphabet to denote variables, and the first letters from the alphabet to denote constants and coefficients. We use f, g to represent functions, and e, t to represent arbitrary expressions. We use φ to represent arbitrary boolean constraints over a set of variables. Sometimes, we also use β to represent coefficients obtained with linear regression. In all cases, the symbols can be subscribed. We use $\bar{x}$ to denote a finite sequence $\langle x_1, x_2, \ldots, x_n \rangle$, for some $n > 0$. Given a sequence S and an element x, $\langle x|S \rangle$ is a new sequence with first element x and tail S.

Given a piecewise function:

$$f(\bar{x}) = \begin{cases} e_1(\bar{x}) & \text{if } \varphi_1(\bar{x}) \\ e_2(\bar{x}) & \text{if } \varphi_2(\bar{x}) \\ \vdots & \vdots \\ e_k(\bar{x}) & \text{if } \varphi_k(\bar{x}) \end{cases} \tag{5.6}$$

where $f \in \mathcal{D} \to \mathcal{R}^+$, with $\mathcal{D} = \{\bar{x} | \bar{x} \in \mathcal{Z}^m \wedge \varphi_{pre}(\bar{x})\}$ for some boolean constraint φ_{pre}, and e_i, φ_i are arbitrary expressions and constraints over $\bar{x}$ respectively. We say that φ_{pre} is the *precondition* of f, and that f is a *constrained recurrence relation* if and only if:

- $\exists i \in [1, k]$ such that e_i contains a call to f.
- $\exists i \in [1, k]$ such that e_i does not contain any call to f (i.e., it is in *closed-form*).
- $\varphi_{pre} \models \bigvee_{1 \leq i \leq k} \varphi_i$.

Given a concrete input $\bar{d} \in \mathcal{D}$, we evaluate $f(\bar{d})$ deterministically, assuming the evaluation of f as a nested *if-then-else* control structure as follows:

```
if φ1(d̄) then
  return e1(d̄)
else
  if φ2(d̄) then
    return e2(d̄)
  else
    ...
  end if
end if
```

More formally, let $def(f)$ denote the definition of a (piecewise) constrained recurrence relation f represented as the sequence $\langle(e_1(\bar{x}), \varphi_1(\bar{x})), \ldots, (e_k(\bar{x}), \varphi_k(\bar{x}))\rangle$, where each element of the sequence is a pair representing a case. The order of such sequence determines the evaluation strategy. Then, the evaluation of f for a concrete value $\bar{d}$, denoted $EvalFun(f(\bar{d}))$, is defined as follows:

$$EvalFun(f(\bar{d})) = EvalBody(def(f), \bar{d})$$

$$EvalBody(\langle(e, \varphi)|Ps\rangle, \bar{d}) = \begin{cases} [\![e]\!]_{\bar{d}} & \text{if } \varphi(\bar{d}) \\ EvalBody(Ps, \bar{d}) & \text{if } \neg\varphi(\bar{d}) \end{cases}$$

Our goal is to find a function $\hat{f} \in \mathcal{D} \to \mathcal{R}^+$ such that for all $\bar{d} \in \mathcal{D}$:

- If $EvalFun(f(\bar{d}))$ terminates, then $EvalFun(f(\bar{d})) = [\![\hat{f}]\!]_{\bar{d}}$, and
- $\hat{f}$ does not contain any recursive call in its definition.

In particular, we look for a definition of the form:

$$\hat{f}(\bar{x}) = a_0 + a_1\ t_1(\bar{x}) + a_2\ t_2(\bar{x}) + \cdots + a_n\ t_n(\bar{x}) \tag{5.7}$$

where $a_i \in \mathcal{R}$, and t_i are expressions over $\bar{y}$, not including recursive references to $\hat{f}$. If the above conditions are met, we say that $\hat{f}$ is a *closed-form* for f.

To illustrate the need of introducing an evaluation strategy for the recurrence that is consistent with the termination of the program, consider the following Prolog program which does not terminate for a call `p(X)` where `X` is bound to an integer:

```
1 p(X) :- X > 0, X1 is X + 1, p(X1).
2 p(X) :- X == 0.
```

The following cost relations can be set up for it:

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & 1 + \mathtt{C_p}(x+1) & \text{if } x > 0 \end{array} \tag{5.8}$$

A CAS will give the closed-form $\mathtt{C_p}(x) = 1 - x$ for such recurrence, however, the cost analysis should give $\mathtt{C_p}(x) = \infty$.

Linear Regression Multiple linear regression (MLR) is a statistical technique used to approximate the linear relationship between a number of independent variables and a dependent (output) variable. Given a vector of independent (input) variables $X^T = (X_1, \ldots, X_p)$, we use MLR to obtain a vector of coefficients $\hat{\beta}^T = (\beta_0, \ldots, \beta_p)$ to predict an output variable Y using the formula

$$\hat{Y} = \hat{\beta}_0 + \sum_{i=1}^{p} \hat{\beta}_i X_i \tag{5.9}$$

A well-known technique for obtaining the coefficient vector is the *least squares* method. Basically, given a set of observations $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$, MLR selects the vector β that minimizes the residual sum of squares

$$\beta = \underset{\beta}{argmin} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_i \right)^2 \right\} \tag{5.10}$$

Sometimes some of the variables are not relevant enough to explain the output, and as a result MLR obtains very small coefficients for them. In such cases, it is preferable to take those variables out of the model, by making their coefficients 0. Lasso is a regression analysis method that uses a penalty to select the most relevant variables. In Lasso, the formula to minimize is

$$\beta = \underset{\beta}{argmin} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_i \right)^2 + \lambda \sum_{i=1}^{p} |\beta_i| \right\} \tag{5.11}$$

where λ is a parameter of the method that determines the level of penalization: the greater the *lambda*, the greater the number of coefficients that are equal to 0.

5.2.3 Description of the Approach

In this section we describe our approach for generating and checking candidate solutions for recurrences that arise in resource analysis. Algorithms 1 and 2 correspond to the *guesser* and *checker* components, respectively, which are shown in Figure 5.2).

Algorithm 1 receives the recurrence equation for a function F to solve, the set of candidate terms, and a threshold to decide when to discard irrelevant terms. The output is a closed-form expression $\hat{F}$ for F, and a *score* S that reflects the accuracy of the approximation, in the range $[0, 1]$. If $S \sim 1$, the approximation can be considered a candidate solution. Otherwise, $\hat{F}$ is an approximation. In line 1 we start by generating a set of random inputs for F. Each input $\bar{x_i}$ is a m-tuple verifying precondition φ_{pre}, where m is the number of arguments of F. In line 2 we produce the training set. The independent inputs are generated by evaluating the candidate terms in $T = \langle t_1, t_2, \ldots, t_p \rangle$ with each tuple $\bar{x} \in \mathcal{I}$. This is done by using function E, defined as follows:

$$E(\langle t_1, t_2, \ldots, t_p \rangle, \bar{x}) = \langle t_1(\bar{x}), t_2(\bar{x}), \ldots, t_p(\bar{x}) \rangle$$

We also evaluate the recurrence equation for input $\bar{x}$, and add the observed output $F(\bar{x})$ as the first element in the vectors of the training set. In line 3 we generate a first linear model by applying function *CVLassoRegression* to the generated training set. *CVLassoRegression* performs a linear regression with Lasso regularization. As already mentioned, Lasso regularization requires a

Algorithm 1: Candidate solution generation.

Input : $F \in \mathcal{D} \to \mathcal{R}^+$: target recurrence relation.
φ_{pre}: precondition defining $\mathcal{D}$.
$T \subseteq \mathcal{D} \to \mathcal{R}^+$: candidate terms.
Λ: range of values to automatically choose a penalization $\lambda \in \mathcal{R}^+$ for Lasso regularization.
k: indicates performing $k-$fold cross-validation, $k > 2$.
$\epsilon \in \mathcal{R}^+$: threshold for term ($t_i \in T$) selection.

Output : $\hat{F} \in Exp$: a candidate solution (or an approximation) for F.
$S \in [0, 1]$: score, indicates the accuracy of the estimation (R^2).

1 $\mathcal{I} \leftarrow \{\bar{x}_i | \bar{x}_i \in \mathcal{Z}^m \wedge \varphi_{pre}(\bar{x}_i)\}_{i=1}^{N}$; `// N Random inputs for F`
2 $\mathcal{X} \leftarrow \{\langle F(\bar{x}) | E(T, \bar{x})\rangle | \bar{x} \in \mathcal{I}\}$; `// Training set`
3 $(\bar{\beta}', \beta_0') \leftarrow CVLassoRegression(\mathcal{X}, \Lambda, k)$;
4 $(T', \mathcal{X}') \leftarrow RemoveTerms(T, \mathcal{X}, \bar{\beta}', \beta_0', \epsilon)$;
5 $(\bar{\beta}, \beta_0, S) \leftarrow LinearRegression(\mathcal{X}')$;
6 $\hat{F} \leftarrow \lambda\bar{x} \cdot \bar{\beta}^T \times E(T', \bar{x}) + \beta_0$;
7 **return** $(\hat{F}, S)$;

parameter λ that determines the level of penalization for the coefficients. Instead of using a single value for λ, $CVLassoRegression$ uses a range of possible values, applying cross-validation on top of the linear regression to automatically select the best value for that parameter, from the given range. The result of this function is the vector of coefficients $\bar{\beta}'$, together with the intercept β_0'. These coefficients are used in line 4 to decide which candidate terms are discarded before the last regression step. Note that $RemoveTerms$ removes the candidate terms from T together with their corresponding input values from the training set $\mathcal{X}$, returning the new set of candidate terms T' and its corresponding training set X'. In line 5, standard linear regression (without regularization nor cross-validation) is applied, obtaining the final coefficients $\bar{\beta}$ and β_0. Additionally, from this step we also obtain the score of the resulting model. In line 6 we set up the resulting closed-form expression, given as a function on the variables in $\bar{x}$. Note that we use the function E to bind the variables in the candidate terms to the arguments of the closed-form expression. Finally, the closed-form expression and its corresponding score are returned as the result of the algorithm.

Algorithm 2 mainly relies on an SMT solver and a Computer Algebra System. Concretely, given the constrained recurrence relation $F \in \mathcal{D} \to \mathcal{R}^+$ defined as

$$F(\bar{x}) = \begin{cases} e_1(\bar{x}) & \text{if } \varphi_1(\bar{x}) \\ e_2(\bar{x}) & \text{if } \varphi_2(\bar{x}) \\ \vdots & \vdots \\ e_k(\bar{x}) & \text{if } \varphi_k(\bar{x}) \end{cases}$$

our algorithm constructs the logic formula:

$$\left[\!\!\left[\bigwedge_{i=1}^{k} \left(\left(\bigwedge_{j=1}^{i-1} \neg\varphi_j(\bar{x}) \right) \wedge \varphi_i(\bar{x}) \wedge \varphi_{pre}(\bar{x}) \implies Eq_i \right) \right]\!\!\right]_{smt} \tag{5.12}$$

where Eq_i is the result of replacing in $F(\bar{x}) = e_i(\bar{x})$ each occurrence of F, if possible, by the definition of the candidate solution $\hat{F}$ (by using *replaceCalls* in line 4), and performing a simplification by the CAS (by using $simplifyCAS$ in line 6). A goal of such simplification is to obtain (sub)expressions supported by the SMT-solver. The function $replaceCalls(expr, F(\bar{x}'), \hat{F}, \varphi_{pre}, \varphi)$ replaces every subexpression in $expr$ of the form $F(\bar{x}')$ by $\hat{F}(\bar{x}')$, if $\varphi_{pre}(\bar{x}) \wedge \varphi \implies \varphi_{pre}(\bar{x}')$. The operation $[\![e]\!]_{smt}$ is the translation of any expression e to a SMT-LIB expression. Although all variables appearing in 5.12 are declared as integers, we omit these details in Algorithm 2 and in Formula 5.12 for the sake of brevity. Note that this encoding is consistent with the evaluation ($EvalFun$) described in Section 5.2.2. Finally, the algorithm asks the SMT solver for models of the negated formula (line 17). If no model exists, then it returns *true*, concluding that $\hat{F}$ is an exact solution to the recurrence, i.e., $\hat{F}(\bar{x}) = F(\bar{x})$ for any input $\bar{x} \in \mathcal{D}$ such that $EvalFun(F(\bar{x}))$ terminates. Otherwise, it returns $false$. Note that, if it is not possible to replace all occurrences of F by $\hat{F}$, or if after performing the simplification by $simplifyCAS$ there are subexpressions not supported by the SMT, then the algorithm finishes returning $false$.

5.2.4 Implementation and Experimental Evaluation

We have implemented a prototype of this approach as an extension of our current recurrence solver. Concretely, it is a specialized back-end solver that integrates the modular architecture depicted in Figure 5.1. It takes a recurrence and returns the closed-form obtained together with an indication if such closed-form has been verified (i.e., if it has been inferred that the closed-form is an exact solution). It also returns the accuracy of the estimation (*score*). This way, the module could also be integrated in any existing resource analysis tool, besides CiaoPP. Depending on the requirements of the application, only verified solutions can be used, or also approximated solutions together with its corresponding accuracy.

The prototype is implemented in Python 3, using Sympy [76] as Computer Algebra System, and Scikit-Learn [89] for the regression with Lasso regularization. We use Z3 [24] as SMT-Solver, and Z3Py [119] as interface. The solver is pre-configured with a set of global parameters:

- An integer $k > 2$, to perform $k-$fold cross-validation. This means that the training set is split into k parts or *folds*. Then, each fold is taken as the test set, training the model with the remaining $k - 1$ folds. Finally, the performance measure reported is the average of the values computed in the k iterations.

Algorithm 2: Solution Checking

Input : $F \in \mathcal{D} \to \mathcal{R}^+$: Target Recurrence Relation.
φ_{pre}: Precondition defining $\mathcal{D}$.
$\hat{F} \in Exp$: A candidate solution for F.
Output : $true$ if $\hat{F}$ is a solution for F, $false$ otherwise.

```
1  φ_previous ← true ;
2  Formula ← true ;
3  foreach (φ, e) ∈ def(F) do
4      Eq ← replaceCalls("F(x̄) − e = 0", F(x̄), F̂, φ_pre, φ);
5      if ¬containsCalls(Eq, F) then
6          Eq ← simplifyCAS(inlineCalls(Eq, F̂, def(F̂)));
7          if supportedSMT(Eq) then
8              Formula ← "Formula ∧ (φ_pre ∧ φ_previous ∧ φ ⟹ Eq)";
9              φ_previous ← "φ_previous ∧ ¬φ" ;
10         else
11             return false;
12         end
13     else
14         return false;
15     end
16 end
17 return (⊭_SMT [[¬Formula]]_SMT);
```

- A range of real values Λ, to automatically choose a λ for Lasso regularization that maximizes the performance of the model via cross-validation.
- A set of basic symbolic functions T, to form the candidate terms t_i to be used in the expression obtained by the algorithm.
- Optionally, a precondition φ_{pre} on the arguments of the recurrence to solve.

We have implemented a prototype of our approach and performed an experimental evaluation with it, whose results are shown in Table 5.1. Column **Bench** shows the name that we have assigned to each recurrence that we have chosen (which is inspired in the program such recurrence originated from during cost/size analysis), and Column **Recurrence** shows their definitions, where we use the same function symbol, f, for all of them. Such recurrences are challenging for our previous solver, either because they cannot be solved by any of the back-end solvers, or because they are necessarily over-estimated in the solving process. Some recurrences, like **nested**, are problematic even for most of the current state-of-the-art solvers. Column **CF** shows the closed-forms obtained by our previous recurrence solver, and Column **CFNew** shows the closed-forms obtained by our

Bench	Recurrence	CF	CFNew	T (s)
merge-sz	$f(x,y) = \begin{cases} max(f(x-1,y), \\ f(x,y-1))+1 & \text{if } x>0 \wedge y>0 \\ x & \text{if } x>0 \wedge y \leq 0 \\ y & \text{if } x \leq 0 \wedge y>0 \end{cases}$	–	$x+y$	0.92
merge	$f(x,y) = \begin{cases} max(f(x-1,y), \\ f(x,y-1))+1 & \text{if } x>0 \wedge y>0 \\ 0 & \text{otherwise} \end{cases}$	–	$\max(0, x+y-1)$	0.71
nested	$f(x) = \begin{cases} f(f(x-1))+1 & \text{if } x>0 \\ 0 & \text{otherwise} \end{cases}$	–	x	0.13
open-zip	$f(x,y) = \begin{cases} f(x-1,y-1)+1 & \text{if } x>0 \wedge y>0 \\ f(x,y-1)+1 & \text{if } x \leq 0 \wedge y>0 \\ f(x-1,y)+1 & \text{if } y \leq 0 \wedge x>0 \\ 0 & \text{otherwise} \end{cases}$	–	$\max(x,y)$	0.12
div	$f(x,y) = \begin{cases} f(x-y,y)+1 & \text{if } x >= y \\ 0 & \text{otherwise} \end{cases}$	–	$\left\lfloor \frac{x}{y} \right\rfloor$	0.13
div-ceil	$f(x,y) = \begin{cases} f(x-y,y)+1 & \text{if } x >= y \\ 1 & \text{if } x<y \wedge x>0 \\ 0 & \text{otherwise} \end{cases}$	–	$\left\lceil \frac{x}{y} \right\rceil$	0.12
s-max	$f(x,y) = \begin{cases} max(y, f(x-1,y))+1 & \text{if } x>0 \\ y & \text{otherwise} \end{cases}$	$x+y$	$x+y$	0.12
s-max-1	$f(x,y) = \begin{cases} max(y, f(x-1,y+1))+1 & \text{if } x>0 \\ y & \text{otherwise} \end{cases}$	–	$2x+y$	0.14
sum-osc	$f(x,y) = \begin{cases} f(x-1,y)+1 & \text{if } x>0 \wedge y>0 \\ f(x+1,y-1)+y & \text{if } x \leq 0 \wedge y>0 \\ 1 & \text{otherwise} \end{cases}$	–	$x+\frac{y^2}{2}+\frac{3y}{2}$	0.13

Table 5.1: Experimental results: closed-forms obtained with the previous (**CF**) and new solver (**CFNew**).

new extension, i.e., after applying Algorithms 1 and 2. All of them have been validated as exact solutions to the recurrences by Algorithm 2.

Finally, Column **T(s)** shows the total time, in seconds, needed to obtain the closed-forms and validate them by our new approach. For all the experiments, we set $k = 2$, in order to perform $2-$fold cross-validation. We have also set the range for λ to 100 equidistant values taken from the interval $[0.001, 1]$. Regarding the set T of candidate terms, for recurrences with one or two arguments, we provide a predefined set of representative functions of the most common complexity orders, as well as some compositions of them. For recurrences with three or more arguments, we provide an initial set of simple functions, that are combined automatically to generate the basic functions t_i for the set T. Finally, as a default precondition, we assume that the initial values for the variables that are the arguments of the recurrences are all greater than or equal to zero, i.e., $\varphi_{pre} = \bigwedge_{x \in Args} (x \geq 0)$, where $Args$ is the set of arguments of the recurrence.

As we can see, none of the recurrences are solvable by our previous recurrence

solver, except `s-max`, which can be solved by applying Theorem 1 (see Section 5.3), although such theorem is also a contribution of this thesis. In contrast, our new solver is able to infer exact closed-forms functions for all the recurrences in a reasonable time.

5.3 Solving Recurrence Relations Including a Maximization Operator

Recurrences including a maximization operator enclosing a recursive call arise quite commonly in the analysis of independent and-parallel logic programs. For example, if we are analyzing elapsed time of a parallel logic program, a proper parallel aggregation operator is the maximum between the times elapsed for each literal running in parallel. Such recurrences also arise in the analysis of sequential programs, for example, when expressing sizes of inner terms in data structures, or when expressing upper bounds in conditionals. To the best of our knowledge, no general solution exists for recurrences of this type. However, we have identified some common classes of this type of recurrences for which we obtain closed-forms that are proven to be correct. In this section, we present these different classes, together with the corresponding method to obtain a correct bound.

Consider the following function $f : \mathcal{N}^m \to \mathcal{N}$, defined as a general form of a first-order recurrence equation with a *max* operator:

$$f(\bar{\mathtt{x}}) = \begin{cases} max(C, f(\bar{\mathtt{x}}_{|i} - 1)) + D & x_i > a \\ B & x_i \leq a \end{cases} \tag{5.13}$$

where $\bar{\mathtt{x}}$ is the sequence of variables $x_1, x_2, \ldots, x_m$, the sequence $x_1, \ldots, x_i - 1, \ldots, x_m$ for some i, $1 \leq i \leq m$ is denoted by $\bar{\mathtt{x}}_{|i} - 1$, $a \in \mathcal{N}$, and C, D, and B are arbitrary expressions possibly depending on $\bar{\mathtt{x}}$. If C and D do not depend on x_i, then C and D do not change throughout the different recursive instances of f (i.e., "recursive calls to" f). In this case, an equivalent closed-form is given by the following theorem:

Theorem 1. *Given $f : \mathcal{N}^m \to \mathcal{N}$ as defined in (5.13), where C and D are functions of $\bar{\mathtt{x}} \setminus x_i$ (i.e., they do not depend on x_i). Then, for all $\bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) = f'(\bar{\mathtt{x}}) = \begin{cases} max(C, B) + (x_i - a) \cdot D & x_i > a \\ B & x_i \leq a \end{cases}$$

Proof. The proof for the case $x_i \leq a$ is trivial.
In the following, we prove the theorem for $x_i > a$, or equivalently, for $x_i \geq a + 1$. The proof is by induction on this subset.

Base Case. We have to prove that $f(x_1, \ldots, x_{i-1}, a+1, \ldots, x_m) = f'(x_1, \ldots, x_{i-1}, a+1, \ldots, x_m)$. Using the definition of f and f' we have that

$$\begin{aligned} f(x_1, \ldots, x_{i-1}, a+1, \ldots, x_m) &= max(C, f(x_1, \ldots, x_{i-1}, a, \ldots, x_m)) + D \\ &= max(C, B) + D \\ f'(x_1, \ldots, x_{i-1}, a+1, \ldots, x_m) &= max(C, B) + (a+1-a) \cdot D \\ &= max(C, B) + D \end{aligned}$$

General Case. Assuming $f(x_1, \ldots, x_{i-1}, x_i, \ldots, x_m) = f'(x_1, \ldots, x_{i-1}, x_i, \ldots, x_m)$, we need to prove that $f(x_1, \ldots, x_{i-1}, x_i+1, \ldots, x_m) = f'(x_1, \ldots, x_{i-1}, x_i+1, \ldots, x_m)$. By induction hypothesis we have that:

$$\begin{aligned} f(x_1, \ldots, x_{i-1}, x_i+1, \ldots, x_m) &= max(C, f(x_1, \ldots, x_{i-1}, x_i, \ldots, x_m)) + D \\ &= max(C, max(C, B) + (x_i - a) \cdot D) + D \\ &= max(C, B) + (x_i - a) \cdot D + D \\ &= max(C, B) + (x_i - a + 1) \cdot D \\ &= f'(x_1, \ldots, x_{i-1}, x_i+1, \ldots, x_m) \end{aligned}$$

□

For the case where $C = g(\bar{\mathtt{x}})$ and $D = h(\bar{\mathtt{x}})$ are functions non-decreasing on x_i, then the upper bound is given by the following closed-form:

Theorem 2. *Given $f : \mathcal{N}^m \to \mathcal{N}$ as defined in (5.13), where g and h are functions of $\bar{\mathtt{x}}$, non-decreasing on x_i. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) \leq f'(\bar{\mathtt{x}}) = \begin{cases} max(g(\bar{\mathtt{x}}), B) + (x_i - a - 1) \times max(g(\bar{\mathtt{x}}), h(\bar{\mathtt{x}}_{|i} - 1)) + h(\bar{\mathtt{x}}) & x_i > a \\ B & x_i \leq a \end{cases}$$

Proof. In order to prove this theorem, we need to recall some properties and lemmas of the `max/2` operator, whose proofs are trivial.

For all $a, b, c \in \mathcal{N} \cup \{0\}$, the following properties hold:

- Commutative: $max(a, b) = max(b, a)$
- Associative: $max(a, max(b, c)) = max(max(a, b), c)$
- Idempotent: $max(a, a) = a$

Lemma 5. $\forall a, b, c \in \mathcal{N} : max(a, b+c) \leq max(a, b) + max(a, c)$

Lemma 6. $\forall a, b, c, d \in \mathcal{N} : a \leq c \wedge b \leq d \implies max(a, b) \leq max(c, d)$

The proof for the case $x_i \leq a$ is trivial.
In the following, we prove the theorem for $x_i > a$, or equivalently, for $x_i \geq a + 1$. The proof is by induction on this subset. For brevity, we only show the argument corresponding to the position of x_i in $\bar{\mathbf{x}}$. However, the proof is still valid considering all arguments.

Base Case. We have to prove that $f(a+1) \leq f'(a+1)$. Using the definition of f and f' we have that

$$\begin{aligned} f(a+1) &= max(g(a+1), f(a)) + h(a+1) \\ &= max(g(a+1), B) + h(a+1) \\ f'(a+1) &= max(g(a+1), B) + ((a+1) - a - 1) \times max(g(a+1), h(a)) + h(a+1) \\ &= max(g(a+1), B) + h(a+1) \end{aligned}$$

General Case. Assuming $f(x) \leq f'(x)$, we need to prove that $f(x+1) \leq f'(x+1)$. By induction hypothesis and Lemma 6 we have that:

$$\begin{aligned} f(x+1) &= max(g(x+1), f(x)) + h(x+1) \\ &\leq max(g(x+1), max(g(x), B) + (x - a - 1) \times max(g(x), h(x-1)) + h(x)) \\ &+ h(x+1) \end{aligned}$$

By Lemma 5 we have that:

$$\begin{aligned} f(x+1) \leq\ & max(g(x+1), max(g(x), B)) \\ &+ max(g(x+1), (x - a - 1) \times max(g(x), h(x-1))) \\ &+ max(g(x+1), h(x)) \\ &+ h(x+1) \end{aligned} \tag{5.14}$$

Consider now the first term appearing in the sum of the right hand side of the inequality (5.14). Since max is associative, and it holds that $\forall x : g(x+1) \geq g(x)$ (which follows from the hypothesis of the theorem), we obtain:

$$\begin{aligned} max(g(x+1), max(g(x), B)) &= max(max(g(x+1), g(x)), B) \\ &= max(g(x+1), B) \end{aligned} \tag{5.15}$$

We consider now the second term in (5.14). By Lemma 5 we obtain:

$$\begin{aligned} &max(g(x+1), (x - a - 1) \times max(g(x), h(x-1))) \\ &\leq (x - a - 1) \times max(g(x+1), max(g(x), h(x-1))) \end{aligned}$$

As before, by associativity of max, this is equivalent to:

$$(x - a - 1) \times max(g(x + 1), h(x - 1))$$

By Lemma 6, and $h(x - 1) \leq h(x)$ (by hypothesis), we have that:

$$(x - a - 1) \times max(g(x + 1), h(x)) \tag{5.16}$$

Replacing the results of (5.15) and (5.16) in (5.14):

$$\begin{aligned} f(x + 1) &\leq max(g(x + 1), B) \\ &\quad + (x - a - 1) \times max(g(x + 1), h(x)) \\ &\quad + max(g(x + 1), h(x)) + h(x + 1) \\ &\quad = max(g(x + 1), B) \\ &\quad + (x - a) \times max(g(x + 1), h(x)) + h(x + 1) \\ &\quad = f'(x + 1) \end{aligned}$$

$\therefore f(x + 1) \leq f'(x + 1)$ □

If the recurrence is not included in the classes defined by Theorems (1) and (2), we try to eliminate the max operator by simplification. Consider an expression $max(e_1, e_2)$ appearing in a recurrence relation. First, we use the function comparison capabilities of CiaoPP, presented in [67, 69]. If an e_i contains non-closed recurrence function calls, we use an SMT solver [24] representing non-closed functions as uninterpreted functions, assuming that they are positive and non-decreasing. Concretely, for each non-closed function call $f(\bar{\mathtt{x}})$ appearing in e_i, we add the properties $\forall \bar{\mathtt{x}}(f(\bar{\mathtt{x}}) \geq 0)$ and $\forall \bar{\mathtt{x}}, \bar{\mathtt{y}}(\bar{\mathtt{x}} \leq \bar{\mathtt{y}} \iff f(\bar{\mathtt{x}}) \leq f(\bar{\mathtt{y}}))$ to a set M. Then, we check whether $M \models e_1 \leq e_2$ or $M \models e_2 \leq e_1$ hold.[2]

Finally, if no proof is found, we replace the max operator by an addition, which causes a loss in accuracy but gives safe upper bounds.

5.4 Conclusions

In this chapter we have presented two extensions of our traditional recurrence solver for dealing with arbitrary, constrained recurrence relations that arise during resource consumption and size analysis of programs.

The first extension consists on a *guess and check* approach that uses linear regression with Lasso regularization and cross-validation for the *guess* stage,

[2] As the algorithm used by SMT solvers in this case is not guaranteed to terminate, we set a timeout.

which infers a candidate closed-form solution for the recurrence. Then, using a combination of a SMT-solver and a Computer Algebra System, it *checks* if the candidate solution is actually a solution. We have applied this approach to a set of recurrences that are either not solvable or over-approximated by our previous solver, obtaining exact results in a reasonable time. Since our technique uses linear regression with a randomly generated training set (by evaluating the recurrence to obtain the dependent value), it is not guaranteed that a solution can be found. Even if a exact solution is found in the first stage, it is not always possible to prove its correctness in the second stage. Therefore, in this sense, this approach is *not complete*. However, it is able to find some solutions that current state-of-the-art solvers are unable to find. As a proof of concept, we have considered a particular deterministic evaluation for constrained recurrence relations, and the verification of the candidate solution is consistent with this evaluation. However, it is possible to implement different evaluation semantics for the recurrences, adapting the verification stage accordingly. Note that we need to require the termination of the recurrence evaluation as a precondition for the conclusions obtained. This is also due to the particular evaluation strategy of recurrences that we are considering. In practice, non-terminating recurrences can be discarded in the first stage, by setting a timeout. Our approach can also be combined with a termination prover in order to guarantee such precondition. As a final remark regarding this extension, note that an alternative use of our tool is to omit the verification stage, using only the closed-form function inferred by the first stage, together with an error measure. Interestingly, this can be useful in some applications where it is enough to have good but unsafe approximations.

The second extension presented in this chapter consists on a method for solving recurrence relations involving a maximization operator. This class of recurrences arise, for example, when expressing sizes of inner terms in data structures, when expressing upper bounds in conditionals, or in the analysis of parallel programs.

In conclusion, both extensions, and in particular the new solver based on the combination of Machine Learning (Linear Regression with Lasso Regularization), CASs, and SMT-solvers, improve our analysis framework significatively by extending the class of recurrences that can be solved. It is also a general approach to the recurrence solving problem, which, besides static program analysis, could be useful for other applications.

6 Application: Estimation and Verification of Run-time Checking Overheads

6.1 Introduction and Motivation

In this chapter we use the parametric cost analysis framework developed in the thesis in a novel application: the static estimation and verification of the overhead introduced by run-time checking. We also extend an existing assertion verification framework to specify "admissible" overheads, and statically and automatically check whether the instrumented program for run-time checking conforms with such specifications.

Dynamic programming languages are a popular programming tool for many applications, due to their flexibility. They are often the first choice for web programming, prototyping, and scripting. The lack of inherent mechanisms for ensuring program data manipulation correctness (e.g., via full static typing or other forms of full static built-in verification) has sparked the evolution of flexible solutions, including assertion-based approaches [17, 20, 29, 30, 47, 61, 90, 94] in (constraint) logic languages, soft- and gradual-typing [21, 28, 32, 85, 86, 96, 104, 109, 110, 111, 112, 115] in functional languages (also applied to, e.g., Prolog [100] or Ruby [56]), and contract-based approaches [31, 62, 63, 66, 80, 85, 88] in imperative languages.

A trait that many of these approaches share is that some parts of the specifications may be the subject of *run-time checking* (e.g., those that cannot be discharged statically in the case of systems that support this functionality). However, such run-time checking comes at the price of overhead during program execution, that can affect a number of resources, such as execution time, memory use, energy consumption, etc., often in a significant way [96, 110]. If these overheads become too high, the whole program execution becomes impractical and programmers may opt for sacrificing the checks to keep the required level of performance.

Dealing with excessive run-time overhead is a challenging problem. Proposed approaches in order to address this problem include discharging as many checks as possible via static analysis [20, 31, 42, 47, 93, 94, 108], optimizing the dynamic checks themselves [60, 88, 97, 107], or limiting run-time checking points [74]. Nevertheless, there are cases in which a number of checks cannot be optimized away and must remain in place, because of software architecture choices (e.g., the case of the external interfaces of reusable libraries or servers), the need to ensure a high level of safety (e.g., in safety-critical systems), etc.

At the same time, low program performance may not always be due to the run-time checks. Consider for example two basic database access operations: insertion and query. Consider also a program that follows the pattern of rare inserts and frequent querying. In this case it can perhaps be fine to perform complex run-time checks in the first operation, provided that the checks in the second are inexpensive enough.

A technique that can help in this context is *profiling*, often used to detect performance "hot spots" and guide program optimization. Prior work on using profiling in the context of optimizing the performance of programs with run-time checks [34, 75, 106] clearly demonstrates the benefits of this approach. Still, profiling infers information that is valid only for some particular input data values (and their execution traces). I.e, the profiling results thus obtained may not be valid for other input data values. Since the technique is by nature not exhaustive, detecting the worst cases can take a long time, and is impossible in general.

We develop and evaluate a static analysis-based approach aimed at delivering guarantees on the costs introduced by the run-time checks in a program (i.e., on the run-time checking overhead). The resulting method provides the programmer with feedback at compile-time regarding the impact that run-time checking will have on the program costs. Furthermore, we propose an assertion-based mechanism that allows programmers to specify bounds on the admissible run-time checking overhead introduced in programs. The approach then compares the inferred run-time checking overhead against the admissible one and provides guarantees on whether such specifications are met or not. Such guarantees can be given as constraints (e.g., intervals) on the size of the input data. We provide the formalization of the method and present also results from its implementation and experimental evaluation. As already said, our proposal builds on *static cost analysis* [8, 25, 26, 27, 71, 92, 102] instead of (or as a complement to) dynamic profiling. This type of analysis is aimed at inferring statically (i.e., without actually running the program with concrete data) *safe upper and lower bounds on execution costs*, i.e., bounds that are guaranteed and will never be violated in actual executions. Since such costs are data-dependent, these bounds take the form of functions that depend on certain characteristics (generally, data sizes) of the inputs to the program. These functions encode (bound) how the program costs change as the size of the input grows.

To the best of our knowledge, this is the first work that proposes, implements, and benchmarks a method for expressing the admissible costs introduced by the run-time checks in a program (the run-time checking overhead) and producing statically (i.e., at compile time) guarantees of such overheads meeting these specifications or identifying errors with respect to them. In the following, we will present our proposal for concreteness in the context of the Ciao system and apply it to logic programs. However, the approach is general and can be applied directly to other languages and systems.

The rest of the chapter proceeds as follows: as preliminaries, Section 6.2 gives an overview of the assertion language used, its relation with run-time and static checking, and the types of run-time checks generated with instrumentation. Then, Section 6.3 presents the proposed method for analyzing, specifying limits on, and verifying the run-time checking overhead. These issues are covered in subsections 6.3.1, 6.3.2, and 6.3.3. Also, subsection 6.3.4 proposes a method for applying accumulated cost (presented in Chapter 3) analysis for detecting hot spots. Section 6.4 describes our implementation and presents results from the experimental evaluation. Finally, Section 6.5 presents our conclusions.

(C)LP Notation Used in the Chapter We recall below some common (C)LP notation used throughout the chapter:

- variable names start with a capital letter: `L`, `Xs`;
- predicate and functor names start with a lower-case letters: `app1C`, `rev`, `warn_if_false`;
- each predicate and functor symbol has a number associated with it, called *arity*, that denotes the number of arguments of that symbol. E.g., the notation `app1/3` means that the predicate `app1` and 3 arguments;
- `[X|Xs]` denotes a list with head `X` and tail `Xs`.

System properties appearing in the examples:

- `term(X)`: `X` is any program term (variable, constant, number, structure, etc.);
- `var(X)`: `X` is a free variable;
- `nat(X)`: `X` is a natural number;
- `list(X)`: `X` is a list (see property definition below);
- `length(L,N)`: list `L` has `N` elements.

```
1 list([]). % empty list
2 list([Head|Tail]) :- list(Tail).
```

Arithmetic expressions appearing in the examples:

- `A /\ B` : integer bitwise AND;
- `A \/ B` : integer bitwise OR;
- `A # B` : integer bitwise exclusive OR (XOR);
- `(A#1)\/B`: integer bitwise implication ($A \rightarrow B \Leftrightarrow \neg\ A\ \vee\ B$).

6.2 Assertions and Run-time Checking

Assertions are linguistic constructions that allow expressing properties of programs. For concreteness we use the `pred` assertions of the Ciao assertion language [46, 47, 93], following the presentation of [108]. Such `pred` assertions allow defining the set of all admissible preconditions for a given predicate, and for each such precondition a corresponding postcondition. These pre- and postconditions are formulas containing literals defined by predicates specially labeled as *properties*, to which we refer to as *prop* literals.

Recalling from Chapter 2, every assertion has a status indicating whether the assertion refers to intended or actual properties. Programmer-provided assertions by default have status `check`, and only assertions with this status generate run-time checks. Static analysis can prove or disprove properties in assertions for a given class of input queries, statically verifying assertions (if all the prop literals are proved to be true, in which case their status is changed to `checked`) or flagging errors (if any prop literal is proved to be false, and then the status is changed to `false`). Assertions can also be simplified by eliminating the prop literals proved to be true, so that only the remaining ones need to be checked. Other information inferred by static analysis is communicated by means of `true` assertions (e.g., see Example 9).

Example 5 (Program with Assertions)**.** *Consider the following implementation of a predicate for reversing a list and its assertions (note that in this running example we are using* `app1` *which appends* one *new element at the end of a list):*

```
:- check pred rev(X,Y)        % \
     : (list(X), var(Y))      %  A1
    => (list(X), list(Y)).    % /

rev([], []).
rev([X|Xs], Y) :-
     rev(Xs, Ys),
     app1(Ys,X,Y).

:- check pred app1(Y,X,Z)                % \
     : (list(Y), term(X), var(Z))        %  A2
    => (list(Y), term(X), list(Z)). % /

app1([],X,[X]).
app1([E|Y],X,[E|T]) :-
     app1(Y,X,T).
```

Assertion `A1` *states that if* `rev/2` *is called with a list* `X` *and a free variable* `Y`*, on its success the second argument* `Y` *will also be a list. Assertion* `A2` *says if* `app1/3` *is called with a list* `Y`*, a term* `X`*, and a free variable* `Z`*, on success the third argument* `Z` *will be a list. The algorithmic complexity of* `rev/2` *is* $O(N^2)$ *in the size (list length in this case)* N *of its input argument* `X`*. While this implementation is obviously not optimal, we use it as a representative of the frequent case of nested loops with linear costs.*

Example 6 (Assertions After Static Checking)**.** *The following listing shows a possible result (only the assertions) after performing static assertion checking for the code of* `rev/2`*. We assume that the code is in a module, exporting only* `rev/2`*, and that it is analyzed in isolation, i.e., we have no information on the callers to* `rev/2`*.*

```
:- check calls rev(X,Y)
     : (list(X), var(Y)).
:- checked pred rev(X,Y)     % \
     : (list(X), var(Y))     %  A1 (proved)
    => (list(X), list(Y)).   % /

rev([], []).
rev([X|Xs], Y) :-
     rev(Xs, Ys),
     app1(Ys,X,Y).
```

```
11
12 :- checked pred app1(Y,X,Z)            % \
13       : (list(Y), term(X), var(Z))     %  A2 (proved)
14      => (list(Y), term(X), list(Z)).   % /
15
16 app1([],X,[X]).
17 app1([E|Y],X,[E|T]) :-
18       app1(Y,X,T).
19
20
```

Here, the interface assertion (`calls`*) for the* `rev/2` *predicate remains active and generates run-time checks (i.e., calls into the module are sanitized). This contrasts with the situation in Example 5, where all assertions generate run-time checks.*

6.2.1 Run-time Check Instrumentation

We recall the definitional source transformation of [107], that introduces *wrapper* predicates that check calls and success assertions, and also groups all assertions for the same predicate together to produce optimized checks. Given a program, for every predicate p the transformation replaces all clauses $p(\bar{x}) \leftarrow body$ by $p'(\bar{x}) \leftarrow body$, where p' is a new predicate symbol, and inserts the wrapper clauses given by $\mathsf{wrap}(p(\bar{x}), p')$:

$$\mathsf{wrap}(p(\bar{x}), p') = \left\{ \begin{array}{l} p(\bar{x}) \text{ :- } p_C(\bar{x}, \bar{r}), p'(\bar{x}), p_S(\bar{x}, \bar{r}). \\ p_C(\bar{x}, \bar{r}) \text{ :- } ChecksC. \\ p_S(\bar{x}, \bar{r}) \text{ :- } ChecksS. \end{array} \right\}$$

Here $ChecksC$ and $ChecksS$ are the optimized compilation of pre- and post-conditions $\bigvee_{i=1}^{n} Pre_i$ and $\bigwedge_{i=1}^{n}(Pre_i \rightarrow Post_i)$ respectively; and the additional *status* variables $\bar{r}$ are used to communicate the results of each Pre_i evaluation to the corresponding $(Pre_i \rightarrow Post_i)$ check, thus avoiding double evaluation of preconditions.

The compilation of checks for assertions emits a series of calls to a `reify_check(P,Res)` predicate, which accepts as the first argument a property `P` and unifies `Res` with `1` or `0`, depending on whether the property check succeeds or not. The results of those reified checks are then combined and evaluated as Boolean algebra expressions using bitwise operations and the Prolog `is/2` predicate. That is, the logical operators $(A \vee B)$, $(A \wedge B)$, and $(A \rightarrow B)$ used in encoding assertions are replaced by their bitwise logic counterparts `R is A \/ B`, `R is A /\ B`, `R is (A # 1) \/ B`, respectively.

Example 7 (Run-time Checks (a)). *The program transformation that introduces the run-time checking harness for the program fragment from Example 5 (assuming none of the assertions has been statically discharged by analysis) is essentially as follows:*

```
rev(A,B) :-
      revC(A,B,C),
      rev'(A,B),
      revS(A,B,C).

revC(A,B,E) :-
      reify_check(list(A),C),
      reify_check(var(B), D),
      E is C/\D,
      warn_if_false(E,'calls').

rev'([],[]).
rev'([X|Xs],Y) :-
      rev(Xs,Ys),
      app1(Ys,X,Y).

revS(A,B,E) :-
      reify_check(list(A),C),
      reify_check(list(B),D),
      F is C/\D,
      G is (E#1)\/F,
      warn_if_false(G,'success').


app1(A,B,C) :-
      app1C(A,B,C,D),
      app1'(A,B,C),
      app1S(A,B,C,D).

app1C(A,B,C,G) :-
      reify_check(list(A),D),
      reify_check(term(B),E),
      reify_check(var(C), F),
      G is D/\(E/\F),
      warn_if_false(G,'calls').

app1'([],X,[X]).
app1'([E|Y],X,[E|T]) :-
      app1(Y,X,T).

```

```
41 app1S(A,B,C,G) :-
42       reify_check(list(A),D),
43       reify_check(term(B),E),
44       reify_check(list(C),F),
45       H is D/\(E/\F),
46       K is (G#1)\/H,
```

The `warn_if_false/2` *predicates raise run-time errors terminating program execution if their first argument is* `0`*, and succeed (with constant cost) otherwise. We will refer to this case as the* worst *performance case in programs with run-time checking.*

Example 8 (Run-time Checks (b))**.** *This example represents the run-time checking generated for the scenario of Example 6, i.e., after applying static analysis to simplify the assertions (see code below). Run-time checks are generated only for the interface calls of the* `rev/2` *predicate. Note that* `rev'/2` *here is a point separating calls to* `rev/2` *coming* outside *the module from the* internal *calls (now made through* `rev_i/2`*).*

```
1  rev(A,B) :-
2       revC(A,B),
3       rev'(A,B).
4
5  revC(A,B) :-
6       reify_check(list(A),C),
7       reify_check(var(A), D),
8       E is C/\D,
9       warn_if_false(E,'calls').
10
11 rev'(A,B) :-
12      rev_i(A,B).
13
14 rev_i([],[]).
15 rev_i([X|Xs],Y) :-
16      rev_i(Xs,Ys),
17      app1(Ys,X,Y).
```

Note also that ***`app1/3`*** *is called directly (i.e., with no run-time checks). Clearly in this case there are fewer checks in the code and thus smaller overhead. We will refer to this case, where only interface checks remain, as the* base *performance case.*

Static Cost Analysis As we mentioned before, we use our static cost analysis to infer the resource usage of both the instrumented and non-instrumented version of a program. In this chapter we add some *sugar syntax* to the resource-related

properties in assertions, to adapt better these properties with the extensions introduced for specifying an admissible run-time checking overhead. Here, we show an example of the output of our static cost analysis. A detailed explanation of the different resource-related properties is given in Section 6.3.2.

Example 9 (Static Cost Analysis Result)**.** *The following assertion is part of the output of the resource usage analysis performed by CiaoPP for the* `rev/2` *predicate from Example 5:*

```
:- true pred rev(X,Y)
      : (list(X), var(Y), length(X,L))
     => (list(X), list(Y),
         length(X,L), length(Y,L))
      + cost(exact(0.5*(L)**2+1.5*L+1),
             [steps]).
```

It includes, in addition to the precondition (`:`*Pre) and postcondition (*`=>`*Post) fields, a field for* computational *properties (*`+`*Comp), in this case cost. The assertion uses the* `cost/2` *property for expressing the* `exact` *cost (first argument of the property) in terms of resolution* `steps` *(second argument) of any call to* `rev(X,Y)` *with the* `X` *bound to a list and* `Y` *a free variable. Such cost is given by the function* $0.5\ L^2 + 1.5\ L + 1$*, which depends on* L*, i.e., the length of the (input) argument* `X`*, and is the argument of the* `exact/1` *qualifier. It means that such function is both a lower and an upper bound on the cost of the specified call. This aspect of the assertion language (including the* `cost/2` *property) and our proposed extensions are discussed in Section 6.3.*

6.3 Specifying, Analyzing, and Verifying Run-time Checking Overhead

Our approach to analysis and verification of run-time checking overhead consists of three basic components: using static cost analysis to infer upper and lower bounds on the cost of the program with and without the run-time checks; providing the programmer with a means for specifying the amount of overhead that is admissible; and comparing the inferred bounds to these specifications. Such approach is depicted in Figure 6.1. In the following, we describe each of these components in more detail.

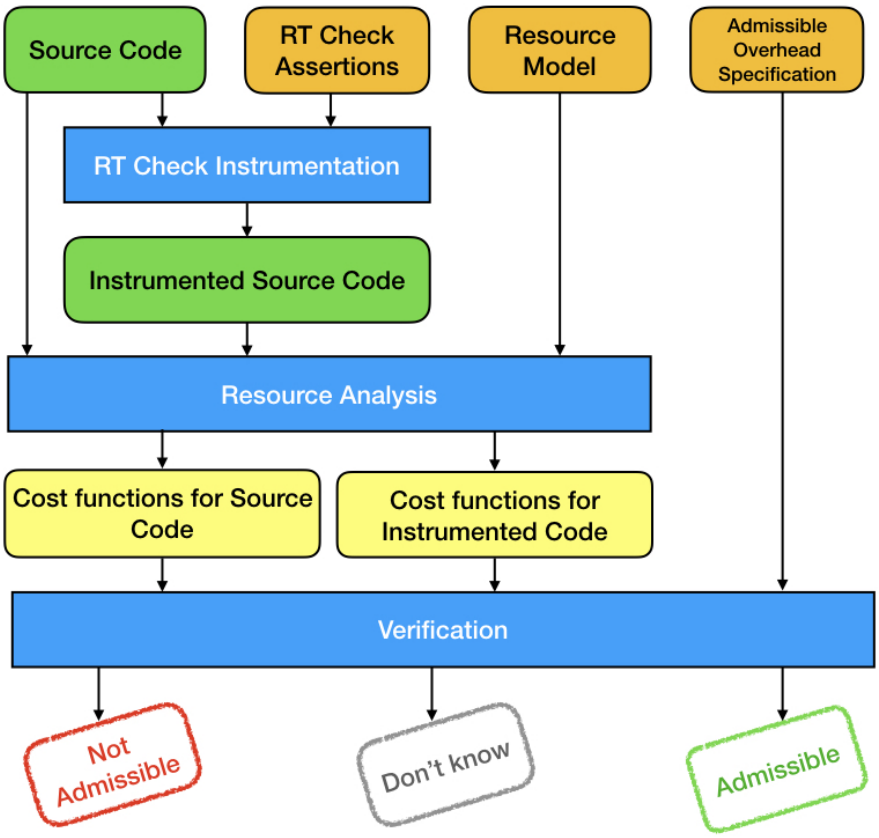


Figure 6.1: Run-time Checking Overhead Analysis and Verification Framework.

6.3.1 Computing the Run-time Checking Overhead (Ovhd)

The first step of our approach is to infer upper and lower bounds on the cost of the program with and without the run-time checks, using cost analysis. The inference of the bounds for the program without run-time checks was illustrated in Example 9. The following two examples illustrate the inference of bounds for the program with the run-time checks. They cover the two scenarios discussed previously, i.e., with and without the use of static analysis to remove run-time checks.

Example 10 (Cost with Run-time Checks (a))**.** *The code below is the result of cost analysis for the run-time checking harness of Example 7 for the* `rev/2` *predicate, together with a (stylized) version of the code analyzed, for reference. Note the change in the complexity order of* `rev/2` *from quadratic to cubic in* L*, the length of list* `A`*, which is most likely not admissible. The reason is that run-time checks are performed at each (recursive) call to* `app1/3`*, and they check the property* `list/1`*, for which the whole input list needs to be traversed. Thus, such run-time checks have linear complexity, and they are performed a linear number of*

times in `app1/3`*, and hence, the complexity order of* `app1/3` *changes from linear to quadratic. Since* `rev/2` *calls* `app1/3` *for each element of the input list (i.e., a linear number of times), its complexity order changes from quadratic to cubic. Note that the additional list traversals introduced by the run-time checks in the body of* `rev/2` *(which have linear complexity) do not affect the complexity order of* `rev/2` *because* `rev/2` *already called predicate* `app1/3` *that was linear. Such checks only increase the constant coefficients of the cost function for* `rev/2`*.*

```
:- true pred rev(A,B)
      : (list(A), var(B), length(A,L))
     => (list(A), list(B),
         length(A,L), length(B,L))
      + cost(exact(0.5*L**3+7*L**2+14.5*L+8),
             [steps]).
rev(A,B) :-
    revC(A,B,C),
    rev'(A,B),
    revS(A,B,C).

revC(A,B,C) :- list(A), var(B), bit_ops.

revS(A,B,C) :- list(A), list(B), bit_ops.

rev'([],[]).
rev'([X|Xs],Y) :-
    rev(Xs,Ys),
    app1(Ys,X,Y).

app1(A,B,C) :-
    app1C(A,B,C,D),
    app1'(A,B,C),
    app1S(A,B,C,D).

app1C(A,B,C,G) :- list(A), term(B), var(C), bit_ops.

app1S(A,B,C,G) :- list(A), term(B), list(C), bit_ops.

app1'([],X,[X]).
app1'([E|Y],X,[E|T]) :-
     app1(Y,X,T).

```

Example 11 (Cost with Run-time Checks (b))**.** *This example shows the result of cost analysis for the base instrumentation case of Example 8: although there are still some run-time checks present for the public interface, the overall cost of the* `rev/2` *predicate remains quadratic, which is probably* admissible.

```
:- true pred rev(A,B)
    : (list(A), var(B), length(A,L))
   => (list(A), list(B),
       length(A,L), length(B,L))
    + cost(exact(0.5*L**2+2.5*L+7),
           [steps]).

rev(A,B) :-
     revC(A,B),
     rev'(A,B).

revC(A,B) :- list(A), var(B), bit_ops.

rev'(A,B) :-
     rev_i(A,B).

rev_i([],[]).
rev_i([X|Xs],Y) :-
     rev_i(Xs,Ys),
     app1(Ys,X,Y).
```

In Figure 6.2 we can see graphically how the cost inferred for `nrev/2` changes, depending on the amount of run-time checks instrumented in its code.

6.3.2 Expressing the Admissible Run-time Checking Overhead (AOvhd)

We add now to our approach the possibility of expressing the admissible run-time checking overhead (AOvhd). This is done by means of an extension to the Ciao assertion language. As mentioned before, this language already allows expressing a wide range of properties, and this includes the properties related to resource usage.

Example 12 (Cost Specification)**.** *For example in order to tell the system to check whether an upper bound on the cost, in terms of number of resolution steps, of a call* `p(A, B)` *with* `A` *instantiated to a natural number and* `B` *a free variable, is a function in* $O(\mathtt{A})$*, we can write the following assertion:*

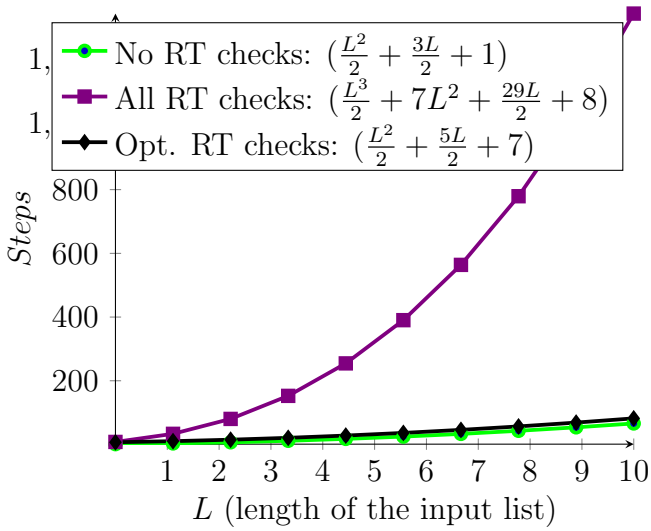


Figure 6.2: Graphical comparison of the cost functions inferred for the different versions of `nrev/2`.

```
:- check pred p(A, B)
     : (nat(A), var(B))
     + cost(o_ub(A), [steps, std]).
```

The first argument of the `cost/2` *property is a cost function, which in turn appears as the argument of a qualifier expressing the kind of approximation. In this case, the qualifier* `o_ub/1` *represents the complexity order of an upper bound function (i.e., the "big O"). Other qualifiers include* `ub/1` *(an upper-bound cost function, not just a complexity order),* `lb/1` *(a lower-bound cost function), and* `band/2` *(a cost band given by both a lower and upper bound). The second argument of the* `cost/2` *property is a list of qualifiers (identifiers). The first identifier expresses the resource, i.e., the cost metric used. The value* `steps` *represents the* number of resolution steps. *The second argument expresses the particular kind of cost used. The value* `std` *represents the* standard cost *(the value by default if it is omitted), the value* `acc` *the accumulated cost (see Chapter 3), etc.*

We introduce the possibility of writing assertions that are universally quantified over the predicate domain (i.e., that are applicable to all calls to all predicates in a program), which is particularly useful in our application. As an example, the following assertion:

```
:- check pred *
     + is_det.
```

states that all predicates in the program should be deterministic, i.e., produce at most one answer. An issue that appears in this context is that different predicates can have different numbers and types of arguments. To solve this problem we introduce a way to express symbolic complexity orders without requiring the specification of details about the arguments on which cost functions depend nor the size metric used, by means of symbols (identifiers) without arguments, such as `constant`, `linear`, `quadratic`, `exponential`, `logarithmic`, etc. For example, in order to extend the assertion in Example 12 to all possible predicate calls in a program (independently of the number and type of arguments), we can write:

```
:- check pred *
     + cost(so_ub(linear), [steps]).
```

In the context of the previous extensions, our objective is expressing and specifying limits on how the complexity/cost changes when run-time checks are performed, i.e., expressing and specifying limits on the run-time checking overhead. To this end we propose different ways to quantify this overhead. Let $\mathcal{C}_p(\bar{n})$ represent the standard cost function of predicate p without any run-time checks and $\mathcal{C}_{p\_rtc}(\bar{n})$ the cost function for the transformed/instrumented version of p that performs run-time checks, p_rtc. A good indicator of the relative overhead is the ratio:

$$\frac{\mathcal{C}_{p\_rtc}(\bar{n})}{\mathcal{C}_p(\bar{n})}$$

We introduce the qualifier `rtc_ratio` to express this type of ratios. For example, the assertion:

```
:- check pred p(A, B)
     : (nat(A), var(B))
     + cost(so_ub(linear),
            [steps, rtc_ratio]).
```

expresses that `p/2` should be called with the first argument bound to a natural number and the second one a variable, and the relative overhead introduced by run-time checking in the calls to `p/2` (the ratio between the cost of the predicate with and without run-time checks) should be at most a linear function. Similarly, using the universal quantification over predicates, the following assertion:

```
:- check pred *
     + cost(so_ub(linear),
            [steps, rtc_ratio]).
```

expresses that, for all predicates in the program, the ratio between the cost of the predicate with and without run-time checks should be at most a linear function.

6.3.3 Verifying the Admissible Run-time Checking Overhead (AOvhd)

We now turn to the third component of our approach: *verifying the admissible run-time checking overhead (AOvhd)*. To this end, we leverage the general framework for resource usage analysis and verification of [67, 68], and adapt it for our purposes, using the assertions introduced in Section 6.3.2. The *verification* process compares the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential, or logarithmic functions) that may come from the specifications or from the analysis results. The possible outcomes of this process are the following:

1. The status of the original (specification) assertion (i.e., `check`) is changed to `checked` (resp. `false`), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,

2. the assertion is "split" into two or three assertions with different status (`checked`, `false`, or `check`) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (status `checked`), incorrect (status `false`), or the tool is not able to determine whether the assertion is correct or incorrect (status `check`), or

3. in the worst case, the assertion remains with status `check`, meaning that the tool is not able to prove nor to disprove (any part of) it.

In our case, the specifications express a band for the AOvhd, defined by a lower- and an upper-bound cost function (or complexity orders). If the lower (resp. upper) bound is omitted, then the lower (resp. upper) limit of the band is assumed to be zero (resp. ∞).

This implies that we need to perform some adaptations with respect to the verification of resource usage specifications for predicates described in [67, 68]. Assume for example that the user wants the system to check the following assertion:

```
1 :- check pred p(A, B)
2      : (nat(A), var(B))
3      + cost(ub(2*A), [steps, rtc_ratio]).
```

which expresses that the ratio defined in Section 6.3.2 (with $\bar{n} = \mathrm{A}$) $\frac{\mathcal{C}_{p\_rtc}(\bar{n})}{\mathcal{C}_p(\bar{n})}$ must be in the band $[0, 2 * A]$ for a given predicate `p`. The approach in [67, 68] uses static analysis to infer both lower and upper bounds on $\mathcal{C}_p(\bar{n})$, denoted $\mathcal{C}_p^l(\bar{n})$ and $\mathcal{C}_p^u(\bar{n})$ respectively. In addition, in our application, the static analysis needs to

infer, both lower and upper bounds on $\mathcal{C}_{p\_rtc}(\bar{n})$, denoted $\mathcal{C}^l_{p\_rtc}(\bar{n})$ and $\mathcal{C}^u_{p\_rtc}(\bar{n})$, and use all of these bounds to compute bounds on the ratio. A lower (resp. upper) bound on the ratio is given by $\frac{\mathcal{C}^l_{p\_rtc}(\bar{n})}{\mathcal{C}^u_{p}(\bar{n})}$ (resp. $\frac{\mathcal{C}^u_{p\_rtc}(\bar{n})}{\mathcal{C}^l_{p}(\bar{n})}$). Both bounds define an inferred (safely approximated) band for the actual ratio, which is compared with the (intended) ratio given in the specification (the band $[0, 2 * A]$) to produce the verification outcome as explained above.

6.3.4 Using the Accumulated Cost for Detecting Hot Spots

So far, we have used the standard notion of cost in the examples for simplicity. However, in our approach we also use the *accumulated cost* (see Chapter 3), inferred by CiaoPP, to detect which of the run-time check predicates (properties) have a higher impact on the overall run-time checking overhead, and are thus promising targets for optimization (or removal, if some reduction in safety guarantees is allowed). As already said in Chapter 3, the *accumulated cost* is based on the notion of *cost centers*, which in our approach are predicates to which execution costs are assigned during the execution of a program. The programmer can declare which predicates will be cost centers. Consider again a predicate p, and its instrumented version p_rtc that performs run-time checks, and let $\mathcal{C}_p(\bar{n})$ and $\mathcal{C}_{p\_rtc}(\bar{n})$ be their corresponding standard cost functions. Let ck represent a run-time check predicate (e.g., `list/1`, `num/1`, `var/1`, etc.). Let $\Diamond_{p\_rtc}$ be the set of run-time check predicates used by p_rtc. Assume that we declare that the set of cost centers to be used by the analysis, $\Diamond$, is $\Diamond_{p\_rtc} \cup \{p\_rtc\}$. In this case, the cost of a (single) call to p_rtc *accumulated in* cost center ck, denoted $\mathcal{C}^{ck}_{p\_rtc}(\bar{n})$, expresses how much of the standard cost $\mathcal{C}_{p\_rtc}(\bar{n})$ is attributed to run-time check ck predicate (taking into account all the generated calls to ck). The ck predicate with the highest $\mathcal{C}^{ck}_{p\_rtc}(\bar{n})$ is a hot spot, and thus, its optimization can be more profitable to reduce the overall run-time checking overhead. The predicate ck with the highest $\mathcal{C}^{ck}_{p\_rtc}(\bar{n})$ is not necessarily the most costly by itself, i.e., the one with the highest standard cost. For example, a high $\mathcal{C}^{ck}_{p\_rtc}(\bar{n})$ can be caused because ck is called very often. We create a ranking of run-time check predicates according to their accumulated cost. This can help in deciding which assertions and properties to simplify/optimize first to meet an overhead target.

Since p_rtc is declared as a cost center, we can express the standard cost of p_rtc as

$$\mathcal{C}_{p\_rtc}(\bar{n}) = \sum_{q \in \Diamond} \mathcal{C}^q_{p\_rtc}(\bar{n}) \tag{6.1}$$

Also, note that the standard cost of p can be expressed in terms of p_rtc as follows:

Lemma 7. *Let p be a predicate, and p_rtc an instrumented version of p with run-time check predicates in the set $\Diamond_{p\_rtc}$. Let $\mathcal{C}_p(\bar{n})$ and $\mathcal{C}_{p\_rtc}(\bar{n})$ be the standard*

cost functions of p and p_rtc, respectively. If we consider the set of cost centers $\Diamond = \Diamond_{p\_rtc} \cup \{p\_rtc\}$, then

$$\mathcal{C}_p(\bar{n}) = \mathcal{C}^{p\_rtc}_{p\_rtc}(\bar{n})$$

Proof Sketch. Considering the set of cost centers $\Diamond = \Diamond_{p\_rtc} \cup \{p\_rtc\}$, it follows from the definition of accumulated cost that $\mathcal{C}^{p\_rtc}_{p\_rtc}$ is the cost of a single call to p_rtc, excluding the cost of the run-time check predicates in $\Diamond_{p\_rtc}$ (as they are cost centers accumulating their own cost). As we defined p_rtc as p (only) instrumented with run-time checks in $\Diamond_{p\_rtc}$, the cost that remains after excluding the cost of the run-time checks is the cost of p, i.e., $\mathcal{C}_p$. □

The *absolute* run-time checking overhead introduced by run-time check properties in $\Diamond_{p\_rtc}$ is the difference of the costs between the two versions of the program, i.e., $\mathcal{C}_{p\_rtc}(\bar{n}) - \mathcal{C}_p(\bar{n})$. The following lemma shows how to express this difference in terms of the accumulated cost of cost centers in $\Diamond_{p\_rtc}$.

Lemma 8. *Considering $p, p\_rtc, \mathcal{C}_p, \mathcal{C}_{p\_rtc}$, and the set of cost centers $\Diamond_{p\_rtc} \cup \{p\_rtc\}$ as in lemma 7. Then*

$$\mathcal{C}_{p\_rtc}(\bar{n}) - \mathcal{C}_p(\bar{n}) = \sum_{q \in \Diamond_{p\_rtc}} \mathcal{C}^q_{p\_rtc}(\bar{n})$$

Proof. As $\Diamond = \Diamond_{p\_rtc} \cup \{p\_rtc\}$, we have that

$$\sum_{q \in \Diamond} \mathcal{C}^q_{p\_rtc}(\bar{n}) = \sum_{q \in \Diamond_{p\_rtc}} \mathcal{C}^q_{p\_rtc}(\bar{n}) + \mathcal{C}^{p\_rtc}_{p\_rtc}(\bar{n})$$

From lemma 7, we obtain

$$\sum_{q \in \Diamond} \mathcal{C}^q_{p\_rtc}(\bar{n}) = \sum_{q \in \Diamond_{p\_rtc}} \mathcal{C}^q_{p\_rtc}(\bar{n}) + \mathcal{C}_p(\bar{n})$$

Finally, from equation 6.1, and subtracting $\mathcal{C}_p(\bar{n})$ to both sides of the equality, we conclude that

$$\mathcal{C}_{p\_rtc}(\bar{n}) - \mathcal{C}_p(\bar{n}) = \sum_{q \in \Diamond_{p\_rtc}} \mathcal{C}^q_{p\_rtc}(\bar{n})$$

□

Thus, we only need to infer accumulated costs and combine them to both detect hot spots and compute the absolute run-time checking overhead, or the `rtc_ratio` described in Section 6.3.2.

Example 13 (Detecting hot spots)**.** *Let* `app1_rtc/3` *denote the instrumented version for run-time checking of predicate* `app1/3` *in Example 5. The following table shows the cost centers automatically declared by the system, which are the predicate* `app1_rtc/3` *itself and the run-time checking properties it uses (first column), as well as the accumulated costs of a call to* `app1_rtc(A,B,_)` *in each of those cost centers, where l_X represents the length of list X (second column):*

Table 6.1: Description of the benchmarks.

`app1(A,B,_)`	list concatenation
`oins(E,L,_)`	insertion into an ordered list
`mmtx(A,B,_)`	matrix multiplication
`nrev(L,_)`	list reversal
`ldiff(A,B,_)`	2 lists difference
`sift(A,_)`	sieve of Eratosthenes
`pfxsum(A,_)`	sum of prefixes of a list of numbers
`bsts(N,T)`	membership checks in a binary search tree

Cost center (ck)	$\mathcal{C}^{ck}_{app1\_rtc}(l_A, l_B)$
`app1_rtc/3`	$l_A + 1$
`list/1`	$3 \times (l_A - 1)^2 + 6 \times (l_A + 1) \times (l_B + 1) + 8 \times (l_A + 1) - 12$
`var/1`	$l_A + 1$
`bit_ops/1`	$3 \times (l_A + 1)$

With these results, from the formula 6.1 and lemmas 7 and 8, we obtain the following costs:

$$\begin{aligned} \mathcal{C}_{app1\_rtc}(l_A, l_B) - \mathcal{C}_{app1}(l_A, l_B) &\leq 3 \times (l_A - 1)^2 \\ &\quad + 6 \times (l_A + 1) \times (l_B + 1) \\ &\quad + 12 \times (l_A + 1) - 12 \\ \mathcal{C}_{app1\_rtc}(l_A, l_B) &\leq 3 \times (l_A - 1)^2 \\ &\quad + 6 \times (l_A + 1) \times (l_B + 1) \\ &\quad + 13 \times (l_A + 1) - 12 \\ \mathcal{C}_{app1}(l_A, l_B) &\leq l_A + 1 \end{aligned}$$

It is clear that the hot spot *is the* `list/1` *property, which is responsible for the change in complexity order of the instrumented version* `app1_rtc/3` *from linear to quadratic.*

6.4 Implementation and Experimental Evaluation

We have implemented a prototype of our approach by modifying the Ciao system, and in particular CiaoPP's abstract interpretation-based resource usage analysis

and CiaoPP's libraries implementing different components for static and dynamic verification (run-time checking transformation, function comparison, etc.).

Table 6.1 contains a list of the benchmarks that we have used in our experiments.[1] Each benchmark has assertions with properties related to shapes, instantiation state, variable freeness, and variable sharing, as well as in some cases more complex properties such as, for example, sortedness. The benchmarks and assertions were chosen to be simple enough to have easily understandable costs but at the same time produce interesting cost functions and overhead ratios.

As stated throughout the chapter, our objective is to exploit static cost analysis to obtain guarantees on program performance and detect cases where adding run-time checks introduces overhead that is not admissible. To this end, we have considered the code instrumentation scenarios discussed previously, i.e. (cf. Examples 7 and 8):

performance	static checking	run-time checking instr.
Original	no	no (off)
Worst	no	yes (full)
Base	*eterms + shfr*	yes (opt)

and we have performed for each benchmark and each scenario run-time checking overhead analysis and verification, following the proposed approach. The optimization in the opt case consists in statically proving some of the properties appearing in the assertions, using different static analyses and using this information to eliminate the checks that are proved to always succeed, as in Example 6. In our experiments we apply this to two classes of properties. The first one is the *state of instantiation of variables*, i.e., which variables are bound to ground terms, or unbound, and, if they are unbound, the *sharing (aliasing) patterns*, i.e., which variables point to each other ("share"). This is a property that can appear in assertions (typically stating that a variable is independent of others) but, more importantly, it is also very important to track grounding information ("strong update"), to ensure the correctness and precision of the state of instantiation information. These properties are approximated using the *sharing and freeness (shfr)* domain [81, 82]. The second class of properties we will be using refers to the shapes of the data structures constructed by the program in memory. To this end we use the *eterms* [114] abstract domain which infers safely these shapes as regular trees. The inferred abstractions are useful for simplifying properties referring to the types/shapes of arguments in assertions.

Regarding the cost analysis, the resource inferred in these experiments is the *number of resolution steps* (i.e., each clause body is assumed to have unitary cost). While in practice other resources can be of interest (time, memory, energy, etc.), the number of resolution steps is a good abstraction for our purposes and the

[1] Sources and additional information available at http://cliplab.org/papers/rtchecks-cost/.

Table 6.2: Experimental results (benchmarks for which analysis infers exact cost functions).

Bench	RTC	Bound Inferred	T_A(ms)	Ovhd	Verif.
	off	$l_A + 1$	98.13		
app1(A,B,_)	full	$3 \cdot {l_A}^2 + 6 \cdot l_A \cdot l_B + 16 \cdot l_A + 6 \cdot l_B + 13$	521.18	$l_A + l_B$	false
	opt	$3 \cdot l_A + 2 \cdot l_B + 8$	311.98	$\frac{l_B}{l_A} + 1$	false
	off	$\frac{1}{2} \cdot {l_L}^2 + \frac{3}{2} \cdot l_L + 1$	218.15		
nrev(L,_)	full	$\frac{1}{2} \cdot {l_L}^3 + \frac{17}{2} \cdot {l_L}^2 + 21 \cdot l_L + 11$	885.08	l_L	false
	opt	$\frac{1}{2} \cdot {l_L}^2 + \frac{5}{2} \cdot l_L + 7$	756.82	1	checked
	off	$\frac{1}{2} \cdot {l_A}^2 + \frac{3}{2} \cdot l_A + 1$	255.55		
sift(A,_)	full	$\frac{2}{3} \cdot {l_A}^3 + 7 \cdot {l_A}^2 + \frac{49}{3} \cdot l_A + 10$	980.63	l_A	false
	opt	$\frac{1}{2} \cdot {l_A}^2 + \frac{7}{2} \cdot l_A + 5$	521.65	1	checked
	off	$l_A + 2$	146.98		
pfxsum(A,_)	full	$2 \cdot {l_A}^2 + 15 \cdot l_A + 20$	749.94	l_A	false
	opt	$3 \cdot l_A + 7$	469.71	1	checked

techniques carry over straightforwardly to the other resources. The times in the tables are given in milliseconds. The experiments were performed on a MacBook Pro with 2.5GHz Intel Core i5 CPU, 10 GB 1333 MHz DDR3 memory, running macOS Sierra 10.2.6.

Tables 6.2 and 6.3 show the results that our prototype obtains for the different benchmarks. In Table 6.2 we group the benchmarks for which the analysis is able to infer the exact cost function, while in Table 6.3 we have the benchmarks for which the analysis infers a safe upper-bound of their actual resource consumption. The analysis also infers lower bounds, but we do not show them and concentrate instead on the upper bounds for conciseness. Note that in those cases where the analysis infers exact bounds (Table 6.2), the inferred lower and upper bounds are of course the same. Column **Bench** shows the name of the entry predicate for each benchmark. Column **RTC** indicates the scenario, as defined before, i.e., no run-time checks (off); full run-time checks (full); or only those left after optimizing via static verification (opt).

Column **Bound Inferred** shows the resource usage functions inferred by our resource analysis, for each of the cases. These functions depend on the input data sizes of the entry predicate (as before, l_X represents the length of list X).

In order to measure the precision of the functions inferred, in Column **%D** we show the average deviation of the bounds obtained by evaluating the functions in Column **Bound Inferred**, with respect to the costs measured with dynamic profiling. The input data for dynamic profiling was selected to exhibit worst case executions. In those cases where the inferred bounds are exact, the deviation is always 0.0%. In Column **Ovhd** we show the relative run-time checking overhead as the ratio (`rtc_ratio`) between the complexity order of the cost of the instrumented code (for full or opt), and the complexity order of the cost corresponding to the original code (off). Finally, in Column $\mathbf{T_A}(\mathbf{ms})$ we list the *cost* analysis time for each of the three cases.[2]

From the results shown in Column **Ovhd** we see that the analysis correctly detects that the full run-time checking versions of the benchmarks (full case) are asymptotically worse than the original program, showing for example a linear asymptotic ratio (run-time checking overhead) for `oins/3`, or even exponential for `bsts/2`. In the case of `app/3`, we can see that the asymptotic relative overhead is linear, but the instrumented versions become dependent on the size of both arguments, while originally the cost was only depending on the size of the first list (though probably it is still worthwhile performing the checks since a list check on the second argument should have been performed anyway in the code). On the other hand, for all the benchmarks except for `app/3` and `bsts/2`, the resulting asymptotic relative overhead of the optimized run-time checking version (opt case), is null, i.e., Ovhd $= 1$.

In the case of `bsts/2`, the overhead is still exponential because the type analysis is not able to statically prove the property *binary search tree*. Thus, it is still necessary to traverse the input binary tree at run-time in order to verify it. However, the optimized version traverses the input tree only once, while the full version traverses it on each call, which is reflected in the resulting cost function. In any case, note that the exponential functions are on the depth of the tree d_T, not on the number of nodes. Analogously, in `oins/3` the static analysis is not able to prove the *sorted* property for the input list, although in that case the complexity order does not change for the optimized version, only the constant coefficients of the cost function are increased. We have included optimized versions of these two cases (marking them with `*`) to show the change in the overhead if the properties involved were verified; however, the *eterms+shfr* domains used cannot prove these complex properties.

Column **Verif.** shows the result of verification (i.e., `checked/false/check`) assuming a global assertion for all predicates in all the benchmarks stating that the relative run-time checking overhead should not be larger than 1 (Ovhd ≤ 1). Finally, Column $\mathbf{T_A}(\mathbf{ms})$ shows that the analysis time

[2]This time does not include the static analysis and verification time in the opt case, performed with the *eterms+shfr* domains, since the process of simplifying at compile-time the assertions is orthogonal to the work presented in this chapter. Recent experiments and results on this topic can be found in [108].

Table 6.3: Experimental results (rest of the benchmarks; we show the upper bounds).

Bench	RTC	Bound Inferred	%D	T_A(ms)	Ovhd	Verif.
oins(E,L,_)	off	$l_L + 2$	0.09	142.55		
	full	$3 \cdot (l_L + 1)^2 + 10 \cdot L_L + 11$	99.93	917.39	l_L	false
	opt*	$3 \cdot l_L + 6$	50.14	340.15	1	checked
mmtx(A,B,_)	off	$r_A \cdot c_A \cdot c_B + 3 \cdot r_A \cdot c_B + 2 \cdot r_A - 2 \cdot c_B$	7.58	460.21		
	full	$4 \cdot {r_A}^2 c_A \cdot c_B + 4 \cdot {r_A}^2 \cdot c_A + 4 \cdot {r_A}^2 \cdot c_B + 4 \cdot {r_A}^2 + r_A \cdot {c_A}^2 \cdot c_B + 4 \cdot r_A \cdot {c_A}^2 + 2 \cdot r_A \cdot c_A \cdot {c_B}^2 + 11 \cdot r_A \cdot c_A \cdot c_B + 20 \cdot r_A \cdot c_A + 15 \cdot r_A + 7$	0.0	1682.54	$N^\dagger$	false
	opt	$r_A \cdot c_A \cdot c_B + 2 \cdot c_A \cdot c_B + 2 \cdot r_A \cdot c_A + 4 \cdot r_A \cdot c_A + 6 \cdot r_A + 2 \cdot c_A + 11$	0.0	1120.23	1	checked
ldiff(A,B,_)	off	$l_A \cdot l_B + 2 \cdot l_A + 1$	2.06	786.22		
	full	${l_A}^2 + 3 \cdot l_A \cdot l_B + 13 \cdot l_A + 2 \cdot l_B + 10$	0.27	1769.22	$\frac{l_A}{l_B} + 1$	false
	opt	$l_A \cdot l_B + 5 \cdot l_A + 2 \cdot l_B + 6$	0.0	1226.15	1	checked
bsts(N,T)	off	$d_T + 3$	0.1	714.83		
	full	$3 \cdot 2^{(d_T+2)} + \frac{3}{2} \cdot d_T^2 + \frac{27}{2} \cdot d_T + 20$	1.19	438.72	$\frac{2^{d_T}}{d_T}$	false
	opt*	$3 \cdot 2^{(d_T+1)} + 4 \cdot d_T + 14$	4.01	245.09	$\frac{2^{d_T}}{d_T}$	false

$\dagger N = max(r_A, c_A, c_B)$

is ≈ 4 times slower on versions with full instrumentation, and ≈ 2 times slower on versions instrumented with run-time checks after static analysis, respectively, but in any case all analysis times are small.

We believe that these results are encouraging and strongly suggest that our approach can provide information that can help the programmer understand statically, at the algorithmic level whether the overheads introduced by the run-time checking required by the assertions in the program are acceptable or not.

6.5 Conclusions

We have proposed a method that uses static analysis to infer bounds on the overhead that run-time checking introduces in programs. The bounds are functions parameterized by input data sizes. Unlike profiling, this approach can

provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input grows. We have also extended the Ciao assertion verification framework to express "admissible" overheads, and statically and automatically check whether the instrumented program conforms with such specifications. Our experimental evaluation suggests that our method is feasible and also promising in providing bounds that help the programmer understand at the algorithmic level the overheads introduced by the run-time checking required for the assertions in the program, in different scenarios, such as performing full run-time checking or checking only the module interfaces.

Since our static analysis is compositional, there are no theoretical limits to the size of programs it can be applied to. Our approach incorporates a mechanism, the trust assertions of Ciao, that allows the programmer to provide the cost of any predicate for which the analysis infers an imprecise result, so that the imprecision does not propagate to the rest of the code. This is of course a burden, but it is obviously less work than the alternative without the tool, i.e., having to reason about every predicate. The approach is in any case useful even for small programs, since it can uncover (changes in) costs that are not immediately obvious even in such programs (see Example 10).

In general, the user should reason about the cost of the run-time checking performed by the program in the same way as about that of the rest of the code. Our tool addresses both of these tasks. Note that, since both tasks are undecidable, the best it can do is compute *safe* approximations. Since our tool cannot possibly solve the problem completely, its objective is instead to assist the programmer in these two tasks in a formally correct way. Again, the underlying argument is that it will always be better to have this tool take care of a good part of both tasks, rather than having to do everything by hand.

We believe that the application of static cost analysis for estimating the impact of run-time checks on program cost and complexity is an important contribution, and that an interesting synergy emerges from this combination. The use of run-time checks is unavoidable in many situations where it is not feasible to verify statically a given property and it is still necessary to guarantee that no incorrect execution is allowed. In this scenario our approach allows the programmer to annotate the program with pre- and post-conditions, but additionally with conditions about the admissible impact of run-time checking, in such a way that some alerts and guarantees can be received statically regarding the final performance of the program. We believe that this is essential for making design decisions, specially regarding performance-correctness trade-offs.

Note also that a useful aspect of our approach is that a change in an implementation of a predicate with the same interface but introducing an undesirable cost can be detected through an assertion violation.

Finally, as argued in the introduction and in the context of the discussion of Horn clauses as intermediate representation (and illustrated by our previous work with Java, Java bytecode, or XC), although we have presented our proposal for

concreteness in the context of the Ciao system and applied it to logic programs, we believe the approach is general and can be applied directly to other languages and systems.

7 Conclusions and Future Work

This thesis addresses the construction of effective analysis tools capable of assisting in the development and optimization of a wide range of software systems, in terms of resource consumption, considering a very general notion of resource. In particular, we improve and extend state-of-the-art static cost analysis techniques by developing a novel, general and flexible framework for cost analysis that can be easily instantiated to infer a wide range of resources, notions of costs, and approximations, which can deal with different programming languages, platforms and execution models.

In Chapter 3 we have presented a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing a wide range of resource usage analyses, including both *static profiling* and the standard notion of cost. Our proposal is an extension and generalization of the standard resource analysis techniques in which the cost relations produced and solved include additional Boolean control variables which allow switching on or off different terms in the relations, as required by the desired type of resource usage analysis. We have also shown how, with a particular Boolean variable assignment, our framework can be instantiated for performing static profiling of accumulated cost, with the results also parameterized by input data sizes. We have reported on an implementation of this general framework within the CiaoPP system, and its instantiation for accumulated cost, and provided some experimental results. The results show that this instantiation of the proposed framework for accumulated cost, in addition to being able to provide results for non-deterministic/multiple solutions programs, is also more efficient than the previous approach based on program transformation, and has a good number of additional advantages. Since our approach is quite general, it can be applied to other logic-based formalisms, such as Constraint Handling Rules, where cost bounds can be inferred and accumulated on constraints/rules. In particular, the fact that cost relations provide a language-independent representation makes such application much easier than, e.g., the application of our previous ad-hoc

transformation-based approach. In summary, our tool helps developers to detect the hot spots with the highest impact in resource usage, potentially minimizing the optimization effort, while maximizing the gains in performance.

In the future, we plan to improve and extend the static profiling presented in several aspects. Currently it is able infer how the total cost of a single call to a predicate is distributed over a set of points of interest, called *cost centers*. However, we have defined cost centers as predicates in the program under analysis, although a more general notion of cost center can have great utility. For example, we could define sets of predicates as cost centers representing different *aspects of interest*, such as the set {`login_user`, `authorize_access`, `revoke_access`,...} representing the cost center `security`. Another interesting extension is to be able to define a (set of) literals as a cost center, in case we are interested in particular calls to predicates. Another possibility is to extend the definition of cost centers with conditions involving the current (abstract) execution environment.

In Chapter 4 we have presented an extension to our parametric framework that can be instantiated for estimating the resource usage of parallel logic programs, for a wide range of resources, platforms, and execution models. To the best of our knowledge, this is the first approach to the cost analysis of *parallel logic programs*. Such estimations include both lower and upper bounds, given as functions on input data sizes. In addition, our analysis also infers other information which is useful for improving the exploitation and assessing the potential and actual parallelism of a program.

We have developed a prototype implementation of our general framework, and instantiated it for the analysis of logic programs with Independent And-Parallelism. However, we have left the instantiations for other types of parallelism as future work. For example, we plan to explore the analysis of *unrestricted* And-parallel programs [22], and dealing with the more general and flexible parallel execution operators provided by this execution model.

In Chapter 5 we have extended the recurrence solver present in the CiaoPP system. Concretely, we have integrated specialized solvers into a modular solver architecture, in order to extend the capabilities of our analysis. We have also presented a novel solver that follows a *guess and check* approach, using linear regression with Lasso regularization for the *guess* stage, and a combination of an SMT-solver and a Computer Algebra System for the *check* step. We illustrate with a set of examples how this approach is useful for improving the scalability and applicability of our static cost analysis. Additionally, we have proposed a method to solve a subclass of recurrences which include a maximization operator. Together, these methods can augment the capabilities of existing recurrence solvers.

In Chapter 6 we have applied our parametric cost analysis framework in a novel application. In particular, we use the static analysis to infer bounds on the overhead that run-time checking introduces in programs. The bounds are functions parameterized by input data sizes. Unlike profiling, this approach can provide guarantees for all possible execution traces, and allows assessing how the

overhead grows as the size of the input grows. We have also extended the Ciao assertion verification framework to specify "admissible" overheads, and statically and automatically check whether the instrumented program conforms with such specifications. Our experimental evaluation suggests that our method is feasible and also promising in providing bounds that help the programmer understand at the algorithmic level the overheads introduced by the run-time checking required for the assertions in the program, in different scenarios, such as performing full run-time checking or checking only the module interfaces. Since our static analysis is compositional, there are no theoretical limits to the size of programs it can be applied to. The approach has proven to be useful even for small programs, since it can uncover (changes in) costs that are not immediately obvious even in such programs.

In the future, we plan to consider other applications of our parametric cost analysis framework, such as the development of resource-oriented automatic optimizations. For example, we plan to define effective methods that use static profiling for identifying hot-spots and the possible causes of performance bugs. In this sense, a tool could automatically select the most appropriate transformations that improve the performance of these parts of the program, while preserving its functionality.

Finally, as an overall conclusion, we have shown how to further generalize and extend the classical parametric cost approach in order to address the challenges posed by current computing platforms and applications, paving the way to the rapid and effective implementation and maintenance of cost analyzers that adapt to continuous changes in many aspects, including the execution models and the type of resource usage information required by applications.

8

Bibliography

[1] (2012). *MPI: A Message-Passing Interface Standard, Version 3.0*. Message Passing Interface Forum.

[2] (2013). *OpenMP Application Program Interface, Version 4.0*. OpenMP Architecture Review Board.

[3] (2014). *CUDA Toolkit Documentation v6.5*. NVIDIA Corporation.

[4] Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., and Puebla, G. (2011a). Cost Analysis of Concurrent OO programs. In *The 9th Asian Symposium on Programming Languages and Systems (APLAS'11)*, volume 7078, pages 238–254. Springer.

[5] Albert, E., Arenas, P., Genaim, S., and Puebla, G. (2008). Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *8th Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

[6] Albert, E., Arenas, P., Genaim, S., and Puebla, G. (2011b). Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203.

[7] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. (2007). Cost Analysis of Java Bytecode. In Nicola, R. D., editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer.

[8] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. (2012). Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159.

[9] Albert, E., Arenas, P., Genaim, S., and Zanardini, D. (2011c). Task-Level Analysis for a Language with Async-Finish parallelism. In Vitek, J. and Sutter, B. D., editors, *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 21–30. ACM.

[10] Albert, E., Correas, J., Johnsen, E. B., Pun, K. I., and Román-Díez, G. (2018). Parallel cost analysis. *ACM Trans. Comput. Logic*, 19(4):31:1–31:37.

[11] Albert, E., Genaim, S., and Masud, A. N. (2011d). More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag.

[12] Alonso, L., Reingold, E. M., and Schott, R. (1995). Multidimensional divide-and-conquer maximin recurrences. *SIAM J. Discret. Math.*, 8(3):428–447.

[13] Annavaram, M. (2006). Energy per instruction trends in intel microprocessors. *Technology Intel Magazine*.

[14] Bjørner, N., Fioravanti, F., Rybalchenko, A., and Senni, V., editors (2014). *Workshop on Horn Clauses for Verification and Synthesis*. Electronic Proceedings in Theoretical Computer Science.

[15] Blelloch, G. E. and Greiner, J. (1996). A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225.

[16] Boogerd, C. and Moonen, L. (2008). On the use of data flow analysis in static profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88.

[17] Boye, J., Drabent, W., and Małuszyński, J. (1997). Declarative Diagnosis of Constraint Programs: an Assertion-based Approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden. U. of Linköping Press.

[18] Brandner, F., Hepp, S., and Jordan, A. (2012). Static profiling of the worst-case in real-time programs. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS 2012, pages 101–110, New York, NY, USA. ACM.

[19] Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., and Giesl, J. (2014). Alternating runtime and size complexity analysis of integer programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 140–155. Springer.

[20] Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M. V., Maluszynski, J., and Puebla, G. (1997). On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden. U. of Linköping Press.

[21] Cartwright, R. and Fagan, M. (1991). Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI 1991)*, pages 278–292, New York, NY, USA. ACM.

[22] Casas, A., Carro, M., and Hermenegildo, M. V. (2008). A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In García de la Banda, M. and Pontelli, E., editors, *24th International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag.

[23] Cousot, P. and Cousot, R. (1977). Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press.

[24] de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

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

[26] Debray, S. K., Lin, N.-W., and Hermenegildo, M. V. (1990). Task Granularity Analysis in Logic Programs. In *Proc. 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 174–188. ACM Press.

[27] Debray, S. K., Lopez-Garcia, P., Hermenegildo, M. V., and Lin, N.-W. (1997). Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA.

[28] Dimoulas, C. and Felleisen, M. (2011). On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(5):1–29.

[29] Drabent, W., Nadjm-Tehrani, S., and Małuszyński, J. (1988). The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581.

[30] Drabent, W., Nadjm-Tehrani, S., and Maluszynski, J. (1989). Algorithmic debugging with assertions. In Abramson, H. and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press.

[31] Fähndrich, M. and Logozzo, F. (2011). Static Contract Checking with Abstract Interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software, FoVeOOS'10*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30, Berlin, Heidelberg. Springer-Verlag.

[32] Findler, R. B. and Felleisen, M. (2002). Contracts for Higher-Order Functions. In Wand, M. and Jones, S. L. P., editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 48–59. ACM.

[33] Flores-Montoya, A. (2016). Upper and lower amortized cost bounds of programs expressed as cost relations. In Fitzgerald, J., Heitmeyer, C., Gnesi, S., and Philippou, A., editors, *FM 2016: Formal Methods*, pages 254–273, Cham. Springer International Publishing.

[34] Furr, M., An, J.-h. D., and Foster, J. S. (2009). Profile-guided Static Typing for Dynamic Scripting Languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 283–300, New York, NY, USA. ACM.

[35] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[36] Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., and Fuhs, C. (2012). Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *Proceedings of PPDP'12*, pages 1–12. ACM.

[37] Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. (2012). HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In Flanagan, C. and König, B., editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer.

[38] Grobauer, B. (2001). Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 253–264, New York, NY, USA. ACM.

[39] Gulwani, S., Mehra, K. K., and Chilimbi, T. M. (2009). SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *The 36th Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139. ACM.

[40] Gupta, G., Pontelli, E., Ali, K., Carlsson, M., and Hermenegildo, M. V. (2001). Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602.

[41] Haemmerlé, R., Lopez-Garcia, P., Liqat, U., Klemen, M., Gallagher, J. P., and Hermenegildo, M. V. (2016). A Transformational Approach to Parametric Accumulated-cost Static Profiling. In Kiselyov, O. and King, A., editors, *13th International Symposium on Functional and Logic Programming (FLOPS 2016)*, volume 9613 of *LNCS*, pages 163–180. Springer.

[42] Hanus, M. (2017). Combining Static and Dynamic Contract Checking for Curry. *CoRR*, abs/1709.04816.

[43] Harper, R. (2016). *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition.

[44] Henriksen, K. S. and Gallagher, J. P. (2006). Abstract Interpretation of PIC Programs through Logic Programming. In *SCAM '06, Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society.

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

[46] Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., and Puebla, G. (2012). An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252.

[47] Hermenegildo, M. V., Puebla, G., and Bueno, F. (1999). Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In Apt, K. R., Marek, V., Truszczynski, M., and Warren, D. S., editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag.

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

[49] Hoefler, T. and Kwasniewski, G. (2014). Automatic complexity analysis of explicitly parallel programs. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 226–235, New York, NY, USA. ACM.

[50] Hoffmann, J., Aehlig, K., and Hofmann, M. (2012). Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62.

[51] Hoffmann, J. and Shao, Z. (2015). Automatic static cost analysis for parallel programs. In Vitek, J., editor, *Programming Languages and Systems*, pages 132–157, Berlin, Heidelberg. Springer Berlin Heidelberg.

[52] Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., and Rümmer, P. (2012). A Verification Toolkit for Numerical Transition Systems - Tool Paper. In Giannakopoulou, D. and Méry, D., editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 247–251. Springer.

[53] Hwang, H.-K. and Tsai, T.-H. (2003). An asymptotic theory for recurrence relations based on minimization and maximization. *Theoretical Computer Science*, 290(3):1475 – 1501.

[54] Igarashi, A. and Kobayashi, N. (2002). Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 331–342, New York, NY, USA. ACM.

[55] Jayaseelan, R., Mitra, T., and Li, X. (2006). Estimating the worst-case energy consumption of embedded software. In *IEEE Real Time Technology and Applications Symposium*, pages 81–90. IEEE Computer Society.

[56] Kazerounian, M., Vazou, N., Bourgerie, A., Foster, J. S., and Torlak, E. (2018). Refinement Types for Ruby. In Dillig, I. and Palsberg, J., editors, *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'18, pages 269–290, Cham. Springer International Publishing.

[57] Kincaid, Z., Breck, J., Boroujeni, A. F., and Reps, T. (2017). Compositional recurrence analysis revisited. *SIGPLAN Not.*, 52(6):248262.

[58] Klemen, M., Lopez-Garcia, P., Gallagher, J., Morales, J., and Hermenegildo, M. V. (2020). A General Framework for Static Cost Analysis of Parallel Logic Programs. In Gabbrielli, M., editor, *Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, volume 12042 of *LNCS*, pages 19–35. Springer-Verlag.

[59] Klemen, M., Stulova, N., Lopez-Garcia, P., Morales, J. F., and Hermenegildo, M. V. (2018). Static Performance Guarantees for Programs with Run-time Checks. In *20th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'18)*. ACM Press.

[60] Koukoutos, E. and Kuncak, V. (2014). Checking Data Structure Properties Orders of Magnitude Faster. In Bonakdarpour, B. and Smolka, S. A., editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 263–268. Springer International Publishing.

[61] Laï, C. (2000). Assertions with Constraints for CLP Debugging. In Deransart, P., Hermenegildo, M. V., and Maluszynski, J., editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 109–120. Springer.

[62] Lamport, L. and Paulson, L. C. (1999). Should Your Specification Language be Typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526.

[63] Leavens, G. T., Leino, K. R. M., and Müller, P. (2007). Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189.

[64] Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M. V., Gallagher, J. P., and Eder, K. (2016). Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In Eekelen, M. V. and Lago, U. D., editors, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, volume 9964 of *Lecture Notes in Computer Science*, pages 81–100. Springer.

[65] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V., and Eder, K. (2014). Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In Gupta, G. and Peña, R., editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer.

[66] Logozzo et al., F. (Accessed: 2018). Clousot. http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

[67] Lopez-Garcia, P., Darmawan, L., and Bueno, F. (2010). A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. In Hermenegildo, M. V. and Schaub, T., editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*,

volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[68] Lopez-Garcia, P., Darmawan, L., Bueno, F., and Hermenegildo, M. V. (2012). Interval-Based Resource Usage Verification: Formalization and Prototype. In na, R. P., Eekelen, M., and Shkaravska, O., editors, *Foundational and Practical Aspects of Resource Analysis. Second International Workshop FOPARA 2011, Revised Selected Papers*, volume 7177 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag.

[69] Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., and Hermenegildo, M. V. (2018). Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, 18(2):167–223. arXiv:1803.04451.

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

[71] Lopez-Garcia, P., Klemen, M., Liqat, U., and Hermenegildo, M. V. (2016). A General Framework for Static Profiling of Parametric Resource Usage. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue*, 16(5-6):849–865.

[72] Méndez-Lojo, M., Navas, J., and Hermenegildo, M. (2007). A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in Lecture Notes in Computer Science, pages 154–168. Springer-Verlag.

[73] Mera, E., Lopez-Garcia, P., Carro, M., and Hermenegildo, M. V. (2008). Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press.

[74] Mera, E., Lopez-Garcia, P., and Hermenegildo, M. V. (2009). Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag.

[75] Mera, E., Trigo, T., Lopez-Garcia, P., and Hermenegildo, M. V. (2011). Profiling for Run-Time Checking of Computational Properties and Performance Debugging in Logic Programs. In *Practical Aspects of Declarative Languages*

(PADL'11), volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag.

[76] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

[77] Miné, A. (2012). Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63. http://www.di.ens.fr/~mine/publi/article-mine-LMCS12.pdf.

[78] Moore, G. E. et al. (1975). Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13.

[79] Morgan, R. G. and Jarvis, S. A. (1998). Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237.

[80] MSR (Accessed: 2018). Code contracts. http://research.microsoft.com/en-us/projects/contracts/.

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

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

[83] Navas, J., Méndez-Lojo, M., and Hermenegildo, M. (2008). Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32. Extended Abstract.

[84] Navas, J., Mera, E., Lopez-Garcia, P., and Hermenegildo, M. (2007). User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363. Springer. 10-year Test of Time Award.

[85] Nguyen, P. C., Tobin-Hochstadt, S., and Van Horn, D. (2014). Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 139–152, New York, NY, USA. ACM.

[86] Nguyen, P. C., Tobin-Hochstadt, S., and Van Horn, D. (2017). Higher-order Symbolic Execution for Contract Verification and Refutation. *Journal of Functional Programming*, 27(3).

[87] Nielson, F., Nielson, H., and Seidl, H. (2002). Automatic complexity analysis. In *Programming Languages and Systems*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer Berlin Heidelberg.

[88] Papi, M. M., Ali, M., Jr., T. L. C., Perkins, J. H., and Ernst, M. D. (2008). Practical pluggable types for java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 201–212.

[89] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[90] Pietrzak, P., Correas, J., Puebla, G., and Hermenegildo, M. V. (2006). Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag.

[91] Podelski, A. and Rybalchenko, A. (2004). A Complete Method for the Synthesis of Linear Ranking Functions. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, Lecture Notes in Computer Science, pages 239–251. Springer.

[92] Portillo, Á. R., Hammond, K., Loidl, H.-W., and Vasconcelos, P. (2002). Cost Analysis Using Automatic Size and Time Inference. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 232–247, Madrid, Spain. Springer-Verlag.

[93] Puebla, G., Bueno, F., and Hermenegildo, M. V. (2000a). An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M. V., and Maluszynski, J., editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag.

[94] Puebla, G., Bueno, F., and Hermenegildo, M. V. (2000b). Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag.

[95] Puebla, G. and Hermenegildo, M. V. (1996). Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag.

[96] Rastogi, A., Swamy, N., Fournet, C., Bierman, G. M., and Vekris, P. (2015). Safe & Efficient Gradual Typing for TypeScript. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM.

[97] Ren, B. M. and Foster, J. S. (2016). Just-in-time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 462–476, New York, NY, USA. ACM.

[98] Rosendahl, M. (1989). Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press.

[99] Sansom, P. M. and Jones, S. L. P. (1995). Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In Cytron, R. K. and Lee, P., editors, *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 355–366, New York, NY, USA. ACM.

[100] Schrijvers, T., Santos Costa, V., Wielemaker, J., and Demoen, B. (2008). Towards Typed Prolog. In Pontelli, E. and de la Banda, M. M. G., editors, *International Conference on Logic Programming*, number 5366 in LNCS, pages 693–697. Springer Verlag.

[101] Serrano, A., Lopez-Garcia, P., Bueno, F., and Hermenegildo, M. V. (2013). Sized Type Analysis for Logic Programs. In Swift, T. and Lamma, E., editors, *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement (technical communication)*, volume 13, pages 1–14. Cambridge U. Press.

[102] Serrano, A., Lopez-Garcia, P., and Hermenegildo, M. V. (2014). Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754.

[103] Shen, K. and Hermenegildo, M. (1996). High-level Characteristics of Or- and Independent And-parallelism in Prolog. *Int'l. Journal of Parallel Programming*, 24(5):433–478.

[104] Siek, J. G. and Taha, W. (2006). Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92.

[105] Sinn, M., Zuleger, F., and Veith, H. (2014). A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Biere, A. and Bloem, R., editors, *Computer Aided Verification*, pages 745–761, Cham. Springer International Publishing.

[106] St-Amour, V., Andersen, L., and Felleisen, M. (2015). Feature-Specific Profiling. In Franke, B., editor, *Proceedings of the 24th International Conference on Compiler Construction*, pages 49–68, Berlin, Heidelberg. Springer Berlin Heidelberg.

[107] Stulova, N., Morales, J. F., and Hermenegildo, M. V. (2015). Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue*, 15(04-05):726–741. http://arxiv.org/abs/1507.05986.

[108] Stulova, N., Morales, J. F., and Hermenegildo, M. V. (2018). Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis. *Science of Computer Programming*, 155:3–26. Selected and Extended papers from the 2016 International Symposium on Principles and Practice of Declarative Programming.

[109] Takikawa, A., Feltey, D., Dean, E., Flatt, M., Findler, R. B., Tobin-Hochstadt, S., and Felleisen, M. (2015). Towards Practical Gradual Typing. In Boyland, J. T., editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[110] Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J., and Felleisen, M. (2016). Is Sound Gradual Typing Dead? In Bodík, R. and Majumdar, R., editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 456–468. ACM.

[111] Tobin-Hochstadt, S. and Felleisen, M. (2008). The Design and Implementation of Typed Scheme. In Necula, G. C. and Wadler, P., editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM.

[112] Tobin-Hochstadt, S. and Horn, D. V. (2012). Higher-order symbolic execution via contracts. In Leavens, G. T. and Dwyer, M. B., editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming,*

Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, pages 537–554. ACM.

[113] Vasconcelos, P. and Hammond, K. (2003). Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag.

[114] Vaucheret, C. and Bueno, F. (2002). More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag.

[115] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S. (2014). Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA. ACM.

[116] Wang, B.-F. (2000). Tight bounds on the solutions of multidimensional divide-and-conquer maximin recurrences. *Theoretical Computer Science*, 242(1):377 – 401.

[117] Wegbreit, B. (1975). Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539.

[118] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3).

[119] Z3Py (2010). Z3 api in python. https://ericpony.github.io/z3py-tutorial/guide-examples.htm. Accessed: 2010-09-30.

[120] Zuleger, F., Gulwani, S., Sinn, M., and Veith, H. (2012). Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303.