From Abstract Programs to Precise Asymptotic Closed-Form Bounds

PH.D. THESIS DISSERTATION
Post-Defense Revised Version

A PH.D. Thesis Dissertation, presented to obtain
the degree of PH.D. *in Computer Science* by
Diego Esteban Alonso Blas

with the advice of Professors
Purificación Arenas Sánchez
Samir Genaim

Department of Computational Systems
School of Computer Science
Complutense University of Madrid

May 2014

Preface

This document is a Dissertation of a PH.D. Thesis, for obtaining the degree of PH.D. *in Computer Science*, issued by the Complutense University of Madrid (UCM), with the *Special Mention of European Ph.D.*

Supporting Publications. A PH.D. Thesis is backed by a set of **Supporting Publications**, which are scientific articles coauthored by the PH.D. candidate and published in journals or in the proceedings of international conferences of adequate prestige. This PH.D. Thesis is supported by the following articles. For each article, we show such quality indicators as the number of citations [1], and the average acceptance rate and the CORE ranking[2] of the conference.

ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009. CORE: B. Acceptance rate: 37%. Citations: 6.

DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE). Citations: 2.

DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012. CORE: A. Acceptance rate: 38%. Citations: 10.

DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013. CORE A. Acceptance rate: 33%. Citations: 2.

[1] As recorded by the Google Scholar web site, `scholar.google.com`

[2] Ranking of conferences in Computer Science published in 2013 by the *Computer Research and Education Association (CORE)*, `www.core.edu.au`

Structure of the Dissertation. This PH.D. Dissertation is structured as a *Dissertation by Publications*. It contains a Summary of the Research, that includes an overview of the state of the art on the subject, the goals and hypothesis, a synthetic discussion, and the conclussions. This text is written in British English. A bibliography figures at the end of this Summary.

Differences with the Official Version: This post-defense version is only intended to international publication. It only contains the Part I of the final version (the Summary of the Research), but not the Spanish translation nor the supporting publications. Some references in the Bibliography have been changed (by request of the defense committee members) and others have been updated to account for the acceptance and publication of the referred works.

Publication History

The first draft of this dissertation was submitted in late January 2014, and it was reviewed for their originality and quality by the following reviewers:

- Ricardo Peña Mari (*Universidad Complutense de Madrid*).
- Fausto Spoto (*Università di Verona*).
- Carsten Fuhs (*University College London*).

The draft was also reviewed by the Commission for Postgraduate Studies of the School of Computer Science of the Complutense University, headed by Narciso Martí Oliet. I would like to thank the reviewers and the members of the commission for their time and their feedback.

The official version of this dissertation was printed and delivered to the Complutense University in late March 2014.

The PH.D. Thesis Defense was held on the School of Computer Science of the Complutense University of Madrid, on May 29th 2014, before a committee made up by the following members:

- Ricardo Peña Mari (*Universidad Complutense de Madrid)*.
- Francisco Javier López Fraguas (*Universidad Complutense de Madrid)*.
- Marko C.J.D. van Eekelen (*Open Universiteit Nederland*).
- Pedro López García (*IMDEA-Software*).
- Germán Vidal (*Universidad Politécnica de Valencia*).

I would like to thank them for their time, their work, and their feedbak.

The present post-defense version was written in June 2014, incorporating all the feedback from the members of the committee.

Acknowledgements

This Dissertation has been made possible and successful thanks to the help and support of a lot of people, to whom I would like to express my gratitude.

First, I thank my PH.D. advisors Puri and Samir, for their advice. Without it, it would not have been possible to complete my PH.D.. I thank them for introducing me to the world of scientific research, and for describing me with a deep honesty the methods, practises, and compromises, of scientific writing and publishing. I am grateful for having benefitted from their expertise in writing in this field, expertise that allowed us to improve the quality of the articles. I also thank them by granting me their time whenever I felt that I needed it. In particular, Samir he deserves a special mention for revealing me how relevant our contribution in our 2012 article was, for this article came at a crucial moment and helped to ensure the viability of my PH.D.. I also thank Puri for her expertise on the fine points of mathematical writing, for her thoroughness and I also thank her for counseling me on how to deal with the red tape.

As no research is done in the void, I must acknowledge those whose work made it possible to start mine. In this sense, I've had the luck of being associated with the group of researches that have developed the COSTA system, which proveded me with an useful environment for developing the contributions of this Thesis. I thank them for their help in whatever problem I had with the installation and configuration of this system. I also owe a special recognision to Elvira, with whom a chance encounter six and a half years ago initiated me in the subject that would become the subject of my PH.D. and of this dissertation.

An important part of my experience was my research stay for three months in the city of Münich. Thus, I am very grateful to Steffen Jost for offering me the opportunity to work with him, as well as to the people with whom I socialised during my stay.

I also want to mention my experience as scholar and teacher in the Faculty, and the deep and interesting conversations on several subjects, which exposed me to the particular world of Academia, with its lights and shadows.

I would also like to thank the reviewers, Ricardo, Fausto, and Carsten, for volunteering their effort and dedication to help me improve the final version of the dissertation; to the members of the defense committee, Ricardo (again), Paco, Marko, Pedro, and Germán, for their time and their interest; and to Narciso, for patiently helping me to wade the red tape.

Finally, I thank the members of my family who have encouraged me to initiate, continue, and terminate my PH.D.

Institutional and Financial Support

Positions. The major support for my PH.D. was provided by the Complutense University of Madrid. In 2009 I was supported by a contract as affiliated researcher, from the Research Project MERIT-FORMS-UCM. Also, from 2010 to 2013 I was supported by a four-years PH.D. scholarship *(Beca Complutense Predoctoral)*, granted by the UCM in 2009.

Research Projects. During the realisation of my PH.D., I have been affiliated to the following Research Projects:

- DOVES (Development Of Verifiable and Efficient Software), a project financed by the Government of Spain, TIN-2008-05624.
- PROMETIDOS *(PROgrama de MÉTodos rIgurosos de DesarrollO de Software)*, S2009TIC-1465, financed by the Regional Government of Madrid.
- GPD *(Grupo de Programación Declarativa)*, a research group in the Complutense University of Madrd, UCM-BSCH-GR35/10-A-910502.

These research projects financed my assistance to the conferences in which the supporting publications were presented.

Abstract

Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the quantity of resources required to safely execute a given program, i.e., without running out of resources. By *resource*, we mean any quantitative aspect of the program, such as memory consumption, execution steps, etc. Several cost analysis frameworks are available, although different in their underlying theory, all of them usually report the cost of a program as a closed-form upper bound function. Inference of lower bound functions has also been considered. However, it has received less attention.

The important properties of a cost analyser are applicability, precision, and scalability. These properties are often antagonistic, e.g., achieving good scalability is typically done by sacrificing precision, and thus, a cost analyser might excel in one property, and fall behind in another. In this thesis, we aim at exploring the gap between the different existing approaches to cost analysis, to develop techniques that are able to infer *asymptotically* precise bounds, and, at the same time, have good scalability and applicability properties.

Our point of departure is the *classical approach* to cost analysis, which is widely applicable, scalable, and has a reasonable precision in practice. However, for some programming patterns it infers bounds which are asymptotically less precise than other approaches. The workflow of this approach is divided into several phases: (i) transforming the input program into an abstract program by replacing data structures by some notion of *size*; (ii) transforming the abstract program to a set of recursive equations that describes the program's cost in terms of the input data; and (iii) solving these equations into closed-form bounds.

Our contributions are related to steps ii and iii above, and consist of analyses, transformations, and solving techniques that allow going ***From Abstract Programs to Precise Asymptotic Closed-Form Bounds***. We argue that they close the precision gap with respect to other approaches, in particular the amortised analysis approach, while they still exhibit good scalability and applicability properties. In addition, some of the contributions bring a methodological innovation in this thesis, which is to apply *The Calculus of Computation* to cost analysis.

Keywords: Static Analysis. Cost Analysis. Automatic Complexity Analysis. Closed-Form Bounds.

Contents

1 | Introduction

Every machine is built to perform its functionality by interacting with its environment. A resource, in this context, is any entity in the environment that can be used by the machine. To safely run a machine, we must supply its environment with all the resources it needs, and thus, it is often crucial to know the quantity of those resources beforehand.

In the concrete case of a computer program, its functionality is to process some input data and generate some output. Its runtime environment includes, among others, the hardware on which it is being executed such as memory, processors, etc. A **resource** in this context can be a physical magnitude like time or energy; a logical element of hardware like a CPU cycle, a word of memory, or a network packet; it may be a logical unit of software, like an object, a method call, or a file; a resource may even be the money that a user has to pay for a billable operation, like sending a text message on a mobile phone. These resources are **consumed** in different ways: time passes during the execution, memory cells are temporarily occupied, method calls are performed, etc.

There are two ways to estimate the **resource consumption** (or **cost**) of a program: dynamic and static. The dynamic or experimental approach uses simulators to run the program on some input data, and measures the quantity of resources consumed. This approach is easy to perform, and gives exact results for the chosen input. However, a major drawback is its incompleteness since the results are valid only for the chosen input. Unlike the dynamic approach, the static or analytical approach can obtain information on the resource consumption of *all* possible executions of a program without executing it. This approach is based on the following idea:

> "Computer programming is an exact science in that [...] all the consequences of executing [a program] in any given environment can, in principle, be found out from the text of the program itself." [74]

Discovering such consequences "from the text of the program itself" without executing it, is known as a **static analysis**.

The benefit of the static approach is that, since it gives information about all possible executions, a good static analysis is by itself enough to verify that a program always works as expected. The major drawback of this approach is that it is too difficult and expensive, so manually analysing large programs is often unfeasible. A static analysis requires a mathematical expertise that only a few highly trained specialists have, and even for them a manual analysis of a simple program is tedious, error-prone, and takes a long time. For a static

program analysis to be widespread, we need to perform it automatically, by a computer program.

1 Static Cost Analysis

Automatic static analysis [101] deals with building a program, called **static analyser**, that takes as input the code of a program (in a given language) and computes some information about its runtime behaviour. This thesis lies in the field of **automatic static cost analysis**, which deals with building a computer program, called **cost analyser**, whose functionality is to take as input a program, and compute[1] the query of resources that the program would consume when executed on some given input data. A cost analyser is usually parametric on the resource to be measured, this is done by using **cost models** which are functions that map instructions to their corresponding costs.

The output of a cost analyser is a **cost function** that maps program executions to their costs. Due to the undecidability of the underlying problems, this function typically does not describe the exact cost of the program, but it is rather an **upper bound function** (UBF for short) or a **lower bound function** (LBF for short) on the actual cost[2]. Moreover, as it is customary in the field of analysis of algorithms [45, 119], these bound functions are defined over a set of numerical input variables, and are given in **closed-form**, that is, deterministic and without recursion. To handle non-numerical input data, cost analysers use **size measures** to map such data to numerical values, e.g., length of a list.

Due to the use of size measures, a cost function is actually a UBF on the *worst-case* cost, or an LBF on the *best-case* cost. This is because two different inputs with different cost might be mapped to the same abstract value. This is also true when the programming language under consideration has some nondeterministic features, because in such case a given input might have different costs depending on some nondeterministic choice.

In the field of analysis of algorithms, a major concern is the scalability of the cost of an algorithm or a program with respect to the size of its input. To answer this kind of questions, it is common to use the asymptotic notations Big O, Big Omega and Big Theta [90]. Intuitively, these notations indicate that, for big values of the input, one function is smaller, greater, or proportional to another one. Thus, it is natural to expect bound functions to be reported in **asymptotic form**, in addition to or instead of non-asymptotic ones.

Due to the inherent incompleteness of automatic static analysis, sometimes the cost analyser is unable to compute a bound function, even if a non-trivial

[1] Subject to the limits of computability theory.

[2] We use **bound function** or **bound** to refer to either a UBF or an LBF in closed form.

one exists. The most important properties of a cost analyser are its applicability, its precision, and its scalability. Concretely,

- The **applicability** refers to the class of programs for which the analyser can infer non-trivial bound functions.
- For those programs in the applicability range, the **precision** is the relation between the bound function computed by the analyser and the worst-case or best-case cost.
- The **scalability** refers to the ability of the analyser to retain performance levels, with the growth in the size of programs.

The applicability may be restricted by language features, like fields [6], or by semantic properties, like termination [5, 81, 141], or by the complexity class of the bound function, for instance, if it can only infer a bound function that is a linear [81, 35] or polynomial [66, 86, 75] function. Regarding the precision of the bound functions, some works focus on improving the functional precision [75]. However, the most important goal is to improve the **asymptotic** precision, that is to infer bounds that are in the exact asymptotic order of the program's cost.

There are several approaches to automatic static cost analysis. However, the most relevant ones for this thesis are the classical approach, and the automated amortised approach. In what follows we explain each of these approaches.

Classical Approach.

The classical approach to manually analyse the cost of a program [45, §2.2] is to first derive from the program a system of equations that defines its cost, and then solve those equations into a bound function. For instance, using the cost measure of the number of steps, the instructions, sequences and branches can be naturally represented as constants, sums and maxima, but each loop is usually written as a recurrence relation[3] (RR for short).

> "Recurrence relations arise frequently [...] through a direct mapping from a recursive representation of a program to a recursive representation of a function describing its properties." [119]

Interestingly, RRs only capture the essential properties of an algorithm, and these are the same for all of its implementations. After we have written the cost of a program as a system of RRs, in the manual approach one only needs solve them into a closed-form solution [62, 112, 59].

Based on this approach, Cohen and Zuckerman [44] present an analyser for a simple imperative language, which is automatic only on the phase that generates RRs. The first fully *mechanical analysis of programs* was presented by

[3] Also called recurrence equations or difference equations.

Wegbreit [135]. He describes METRIC, a cost analyser for LISP programs that works in two phases: the first phase automatically transforms a LISP program into a system of RRs that describes its cost; and the second phase uses an automatic solver of RRs to derive a closed-form solution from the system.

Following Wegbreit, the classical approach to automatic static cost analysis splits the analysis in two independent phases: analysis and resolution.

1. The **analysis** transforms the program into a system of RRs. The construction of this phase may be different for different programming languages, for different kinds of cost model analysis, or different size measures used.
2. The **resolution** of a system of RRs computes a closed-form solution. Since RRs are a purely numerical representation, this phase is independent of the programming language or the cost model.

The benefit of the classical approach is that, by putting all the special features of the language, cost model, or size measure, into the first phase, the methods of the second phase are equally applicable to all languages or cost models. So, we can see a system of RRs as an *abstract program*, that serves as a common target for cost analysers of any language. Another benefit is that, if we have several programs implementing the same algorithm, and on each program we run an analyser for its respective language, then the systems of RRs tend to be (almost) the same. So, the first phase serves to clear the implementation details and to leave only the essence of the algorithm.

After Wegbreit's seminal article, many works have extended or adapted both phases of the classical approach. As these phases are independent, in what follows we briefly comment on the existing work on each one, by separate.

Generation of Recurrence Relations. Research on the first phase, on generating RRs, is concerned with dealing with the specific semantic features of each language, cost model, size measure, or the kind of cost function (worst, best or average case). Regarding programming languages, a lot of work exists on analysing declarative languages, because in these languages a program already is in a recursive representation. Analysers thus exist for functional [73, 134, 113, 63, 31, 133, 32, 51, 56], and logic programming languages [54, 100, 60]. In contrast, analysers for imperative or object-oriented languages are more scarce [58, 99, 88]. COSTA [10] is the first approach to analysing programs written in an object-oriented subset of Java Bytecode. The COSTABS [3] system is an analyser for the concurrent object-oriented language ABS. Regarding the cost model used, although most work has focused on models associated with time or time complexity, there are also analysers for other models like heap memory [13], concurrent tasks [11] or energy [88]. Regarding the size abstraction, some works on functional languages use advanced type systems like de-

pendent types [63] or sized types [120, 133]. Regarding the kind of cost function looked for, most works focus on inferring a UBF on the worst-case cost, but there are some that are able to infer an LBF on the best-case cost [55], or both UBF and LBF [113, 121]. There are also some works that apply the classical approach to analysing average case analysis [73, 58]. These works have to deal with two difficult problems: the first one is to manage probability distributions on the data, which requires a special semantics [107]; the second is to consider the combinatorial properties of data structures [119, 59].

Recurrence Relations vs. Cost Relations. One major problem faced by analysers for this phase, starting from Wegbreit's [135], is dealing with the semantic gap between deterministic RRs and program semantics. In a program there usually are many execution paths, and these paths may be controlled by such guards or conditions that cannot be represented in the corresponding RR. As a result, for some values of the size of the input, the equations that correspond to different paths may overlap (i.e., are applicable on the same input). Several solutions exist for this problem. Some works [63, 51] directly use maxima operators to keep determinism. Karp proposes using Probabilistic RRs [84]. Rosendahl [113] proposes a more elegant solution, based on the theory of Abstract Interpretation [46], which is to consider a nondeterministic system of RRs as an *abstract program*, whose abstract semantics overapproximates the set of costs of the executions of the program. This solution is also used in COSTA [10], in which the system of nondeterministic RRs is called a **cost relation system** (CRS for short). The benefit of using a nondeterministic system of RRs or CRS as an abstract program, is that we can infer both UBFs and LBFs from the same abstract program.

Solving Recurrence Relations. Research on this phase deals with developing automatic methods to solve a system of RRs, or a nondeterministic CRS, into a bound function. Unlike the intense activity in the phase of generating RRs, research on this second phase has been slower and less extensive, because even *manually* solving a system of RRs into a closed-form solution is already too difficult. At the time of Wegbreit's article, there were no such automatic methods. The first methods were developed for computer algebra systems, like MACSYMA [80], MATHLAB [42], or MAPLE [58, 114]. Other works on methods to solve a system of RRs into closed-form solutions focus on handling special functions, like polynomials, or binomials [105, 106, 122]. Other recent works include [36, 39]. The PURRS system [27] implements some methods to solve or approximate a given system of RRs.

However, all these works handle *deterministic* RRs, not the CRSs used, for instance, in COSTA [10]. There is too little work on solving CRSs or similar ab-

stract programs. Rosendahl [113], who uses abstract programs to capture the cost, used program transformations inspired from ACE [97]. The PUBS system [5] is the first system that can handle a wide class of CRSs. It is based on viewing CRSs as an abstract programming language with an operational semantics, and using program analysis techniques to bound its worst case execution. In particular, it uses the connection between RR computation and termination analysis first described in [85]. The extension of PUBS described in [14] allows for inferring more precise UBFs or even LBFs.

Amortised Approach.

In the 60's and the 70's, the classical approach to cost analysis was (manually) applied to obtain accurate bound functions for many programs. However, in the early 80's some computer scientists noticed that when they applied the classical approach to analyse the performance of some programs that repeatedly operate on some data structures, like self-balancing binary search trees, they usually got an asymptotically imprecise UBF. The special feature of these programs is that, depending on the state of the data structure, each single operation could be expensive (have a high cost) or cheap (have a low cost). An analysis following the classical approach would consider all of them expensive, and thus would lead to asymptotically imprecise UBF if such scenario is not possible. This lead Robert Tarjan to develop an alternative approach to cost analysis.

Amortised cost analysis [131] is an approach to cost analysis in which the focus is not on the cost of one operation, but on the cost of a sequence of operations. This approach is based on the observation that, although one single operation can be expensive or cheap, in every sequence of operations there are enough cheap operations so as to *amortise* the cost of the expensive operations among the cheap ones. To prove that this *amortisation* happens, the methods of this approach use the following metaphor: they assume that the data structure stores some kind of savings, and they study the relation between the cost of an operation and its effect on the savings of the data structure. There are two main methods for amortised cost analysis that differ in how to represent those savings: in the banker's method it is represented as credit associated to each element in the data structure; whereas in the physicist's method, the savings are represented as a potential of the whole data structure. These methods are equivalent, so choosing one method or another depends on the program being analysed. The amortised approach is more precise than the classical approach, in that it obtains an asymptotically precise UBF for the kind of programs mentioned above.

After Tarjan's seminal paper, amortised analysis has been applied to analyse

the cost of operations on data structures, both in functional [104] and imperative [45, §17] languages. It application to an automatic cost analysis, called an *automated amortised analysis*, was introduced by Jost [77, 81]. His work considers a strict first-order functional language, and he presents an analysis, based on a type system for that language, that infers for each procedure in the program a couple of potential functions: the first one is a potential function over the input of the procedure, which gives a UBF on the cost of the procedure, and the second one is a potential function over the output of the procedure, which is used to pay for the cost of the subsequent operations on the output. Being based on types, this approach is very modular, since each function can be analysed separately.

The approach of Jost was later extended in several directions. Campbell [35] considers inferring, for a similar language, a bound on the stack space, for which he had to modify the type system. Rodriguez [110] applies the automated amortised analysis to an object-oriented language, for which it is necessary to handle inheritance. The system RAML [75] is able to infer multivariate polynomial UBFs, by considering a wide set of type-directed norms, which includes products of polynomials on the lengths of two lists, or introspective measures like the sum of the lengths of the lists in a matrix. Jost et al. [82] consider a strict higher-order functional language, whose analysis requires taking into account the cost of evaluating each partial function application. Scherer and Hoffmann [118] consider the case in which the cost depends on those arguments already stored in the closure of a partial function application. Simões et al. [124] consider a language with lazy evaluation, for which they need to handle *thunks*, that is, function calls whose evaluation is suspended. Hoffmann and Shao [76] extend the analysis to an imperative language with arrays and integer arithmetic operations. Atkey [23] presents a method to analyse the cost of linked data structures, like lists, inspired on the banker's method and based on separation logic [108].

2 Applications of Cost Analysis

Cost analysis has many potential applications. In hardware development, cost analysis is used to analyse and verify the worst-case execution time of processors [139], especially relevant for real-time systems. Nowadays, there is also an interest in energy consumption [88] for embedded systems.

There are also different applications for cost analysis in parallel computing (multiprocessor or distributed). One application is that of load balancing and granularity control of programs, that is, to estimate the optimal way to divide a computation into parallel tasks and how to split those tasks among the

processors. One way to do this is to use an analysis to estimate a good work load for each processor [96]. A more flexible technique is to use *autonomous mobile programs (AMP)* [56], where each AMP analyses the cost of its pending computations, and decides by itself to migrate to another processor with smaller workload. In a distributed computation, as the communication network between nodes can become a performance bottleneck, it is also important to estimate how many messages are sent across the network [12]. Also, if we write our programs in a concurrent language, we may want to know the peak number of active concurrent tasks [11], as this is the maximum parallelism that can be exploited for this program. Cost analysis can also be applied in the context of cloud computing [22], in which a company provides virtual computing capabilities by hour. In this context, cost analysis serves as a tool to estimate, before the transaction, the price of the computation to purchase.

Programmers can also use cost analysis when developing programs, by writing assertions on the expected cost, and letting the cost analyser verify these assertions: such an approach to resource usage verification [95] is integrated in the CIAO compiler [71]. Also cost analysis can be applied when deploying and running a program, in the context of resource bound certification [50, 9]. This is based on the idea of attaching to the program a proof that witnesses that it meets a specification, a general technique known as constructive program verification [137].

Liang [93] applies cost analysis to query optimisation of logic programs. Cost analysis also serves to automatically adapt a program to the hardware platform in which it runs: in the PetaBricks [20] language, a programmer can write several procedures to perform a given task, each one implementing a different algorithm. The compiler for this language and the *autotuner* automatically choose the most efficient procedure for a given architecture and size of data. Then a cost analyser can be used to guide the *autotuning*.

3 Thesis Objectives

The field of this dissertation is automatic static cost analysis using the classical approach, which is based on the two phases, one for generating the abstract programs, and another for solving these abstract programs into bound functions. The main issues in this field are the applicability and the precision, especially the asymptotic one, of these analysers.

The main goal of this dissertation is to improve the precision and applicability of cost analysers based on this approach, to make their precision similar to those based on amortised analysis. In particular, we are interested in improving the techniques on the second phase, to go ***from abstract programs to precise***

asymptotic closed-form bounds. In short, we seek to build[4] an analyser that infers more asymptotically precise UBFs for more programs.

The main challenge is the precision gap between the classical and amortised approaches. Although it is known that there are programs for which analysers based on the latter infer more precise UBFs than those based on the former, it is not clearly explained why that happens, nor a criterion to know for which programs. This lead to some misconceptions about the amortised approach, so that some texts [104] use the notion of *amortised cost* in contraposition to *worst-case* cost, as if they were different properties of programs instead of different methods of analysis. One of this misconceptions is that amortised cost analysis is only applicable to data structures, not to simple algorithms or loops. In fact, the methods of automated amortised analysis [81] define the potential as a measure related to data structures or constructors.

As a result of these misconceptions, for a long time, it was assumed that in order to infer a precise bound function it was necessary to use the techniques of the amortised approach. The idea that it was possible to infer precise bounds *without* using the techniques of amortised analysis, was first observed in the context of the SPEED project [66]. Research on this project was focused on cost analysis[5] of imperative programs, consisting in nested loops in which inner and outer loops share some counter variables. For these programs, analysers based on the classical approach, like COSTA [7, 1], as well as other analysers that do not follow the classical approach, like [86], infer an asymptotically imprecise UBF, or even fail to infer any UBF at all. Instead, the members of the SPEED project achieve the automatic inference of a precise UBF using techniques related to termination analysis of loops [66, 65, 67, 141]. This suggests that it is possible to achieve what an amortised analysis does *without* using the techniques of amortised analysis. The starting challenge for this thesis was to develop cost analyses techniques so as to infer UBFs as precise as those of the amortised approach, while keeping the separation of phases as in the classical approach. This separation is essential to obtain a cost analyser that is generic on the programming language, parametric on the cost models, and adaptable to compute UBFs or LBFs either in asymptotic or non-asymptotic form.

[4] That is, learn what problems need to be considered and how to solve them in order to build.

[5] The SPEED project only considered one cost model, the number of iterations of loops, so their analyses were called *bound* or *reachability bound analyses* [141].

4 Contributions and Outline

As a background, in Chapter 2 we present COSTA and PUBS. COSTA is a cost and termination analyser for Java Bytecode programs, and PUBS is a CRS solver, which follow the division of work in the classical approach. The contributions of this thesis consist in improvements and extensions to the different parts of COSTA and PUBS, but they can be applied to any other approach with similar architecture. In Chapters 4-6 we describe the contributions of this thesis, and relate them to the supporting articles. Each of these chapters also includes a discussion of related works and a discussion on further points that can be found in the article. Finally, in Chapter 7 we draw our conclusions and discuss possible future work. Next we briefly describe the main contributions.

An Asymptotic Transformation. Bound functions can be big and intricate expressions, difficult to read or process. Asymptotic bounds are smaller and simpler, and they succinctly describe how the cost scales on the size of the input of the program. Unfortunately, many cost analysers do not infer asymptotic bounds. Our first contribution is an algorithm to transform bound functions to asymptotic form, and a scalable asymptotic CRSs solver that is based on this transformation. This contribution was presented in the following article:

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

This contribution is described in Chapter 3.

A Value Analysis for Nonlinear Operations. In order to infer precise bounds, COSTA uses value relations, as conjunctions of linear constraints, that approximate the values that program variables can take at runtime. Our second contribution studies the limitations of this approximation step, in particular its inability to model nonlinear arithmetic operations, and develop a technique to overcome these limitations, in a scalable way. This contribution was presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

This contribution is described in Chapter 4.

An Approach to handle the Output-Cost Codependency in Cost Analysis. In most cost analysers, the cost of a program is captured only in terms of the input of the program. However, in some programs another codependency appears between the cost and the output of a procedure, and any cost analysis that ignores this codependency necessarily obtains a UBF that is asymptotically imprecise. Our third contribution gives a detailed example of this codependency, and explains how it affects the precision of the analysis of COSTA. We then present an approach to cost analysis that keeps this codependency in order to obtain a precise UBF. This approach is based in the automatic techniques of logical analysis of programs [34], namely satisfiability modulo theory (SMT) and quantifier elimination (QE) [91, §9]. Importantly, we also expose a strong relation between the output-cost codependency and amortised analysis. This contribution was presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

This contribution is described in Chapter 5.

A Precise and Practical Method to Solve CRSs. Our last contribution is a new method to solve CRSs (abstract programs) into UBFs, that obtains, for many abstract programs, UBFs that are asymptotically accurate. This method is based on complete and scalable techniques for quantifier elimination and satisfiability in the theory of linear real arithmetic, which makes the key for scalability. This contribution is presented in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

This contribution is described in Chapter 6.

2 | The COSTA System

This thesis presents techniques to compute precise asymptotic bounds from nondeterministic abstract programs. We developed these techniques in the context of COSTA. This chapter describes the architecture and the inner workings of COSTA that are necessary to explain our contributions.

1 The Architecture of COSTA

COSTA [7] is an automatic static analyser for Java Byte-Code (JBC for short), a low-level object-oriented language [94]. It takes as input a program written in JBC, and can prove its termination or compute a bound on its cost, with respect to the given cost model. This bound can be either a UBF on its worst-case cost, or a LBF on its best-case cost. These bounds are numeric functions on the size of the input, and they are in closed-form, i.e., deterministic and nonrecursive.

The architecture of COSTA is illustrated in Figure 2.1. The rectangular boxes with rounded corners represent its main components (transformations, analyses, etc.), and the rectangular boxes represent intermediate information that is passed between these components. The ellipses are annotations that highlight the contributions of this thesis. The rest of this chapter describes the inner workings of COSTA, by applying each of the following components on a simple program that we introduce in Example 2.1:

- The **Rule-Based Representation** (RBR for short) is obtained by a declarative compilation of the Control Flow Graph (CFG for short) of the input JBC program. An RBR program is a set of rules that define several procedures, with one rule per basic block in the program, and one procedure per method, loop or branching statement. The RBR provides a uniform representation of the control-flow as guarded rules and interprocedural calls, in which all iterations are performed via recursive calls. Section 2.2 discusses the RBR and the result of the decompilation for our example.
- The **Abstract Cost Rules** (ACR for short) program is obtained from the RBR by abstract compilation: a **size abstraction** maps each data structure to a size variable and each operation to a constraint between size variables; and a **cost model** maps each RBR instruction to a **cost annotation** that models its cost. Section 2.3 describes the syntax and semantics of ACR programs, and the abstract compilation for our example.
- The **Cost Relation System** (CRS for short) is obtained from the ACR by removing the output variables from each inter-procedural call, and replacing

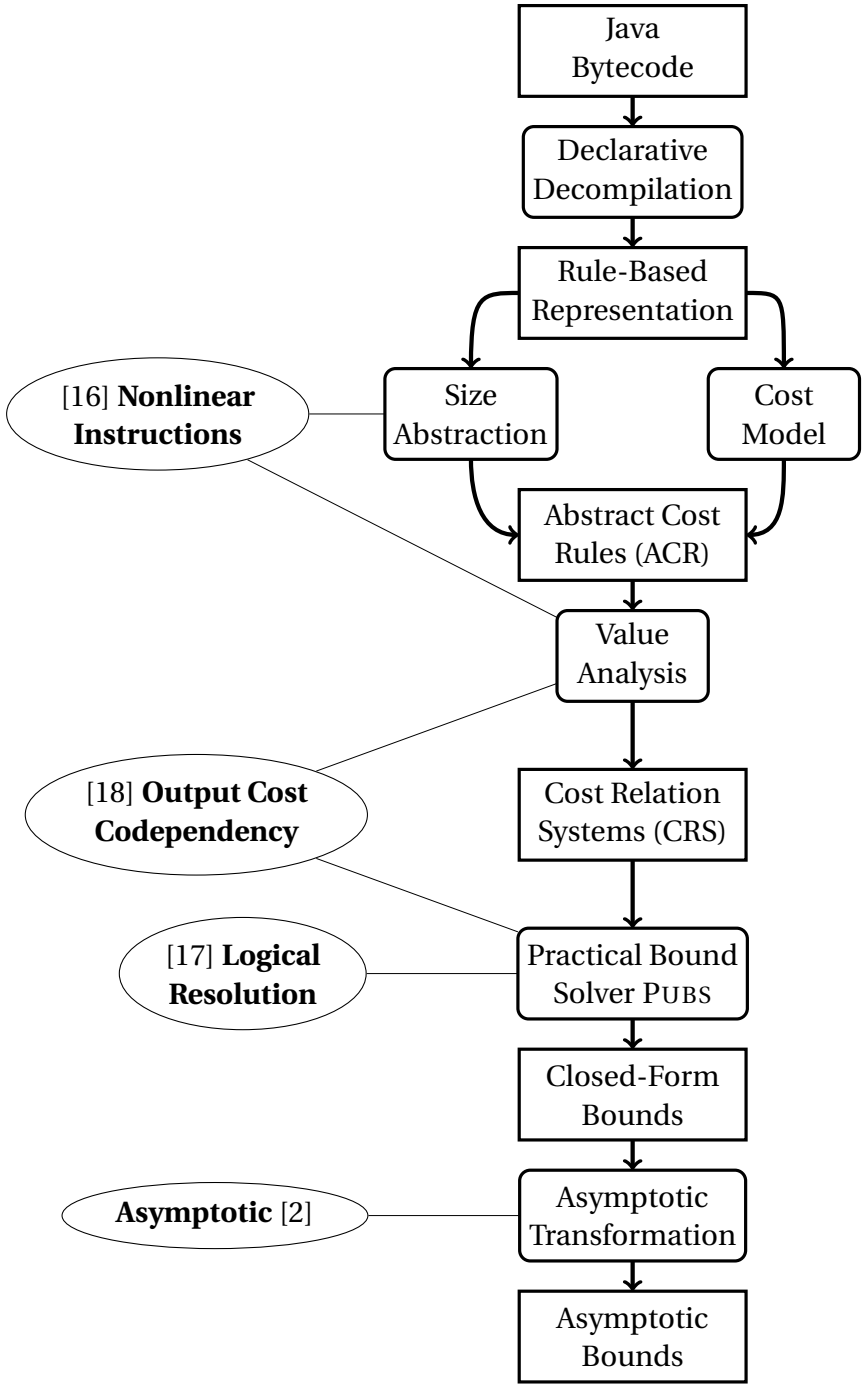

Figure 2.1: The architecture of COSTA: analyses, transformations, intermediate information formats, and the contributions of this thesis.

those output variables with a postcondition formula computed by a **value analysis**. The CRS thus represents the relation between the input and the cost of each procedure. Section 2.4 discusses the syntax and semantics of the CRS, the ACR to CRS transformation, and the CRS generated for our example.

- The final step is to solve the CRS into UBFs or LBFs. For this, COSTA relies on its subsystem PUBS (Practical Upper Bound Solver) [5], which takes as input a CRS and computes its corresponding UBFs or LBFs. Section 2.5 describes the solving procedure of PUBS, and applies it to our example.

Note that although COSTA was originally designed to analyse JBC programs, it is easy to adapt this architecture to other languages.

Example 2.1. The Java code of our running example for this chapter is depicted in Figure 2.2. It is an implementation of the insertion sort algorithm. The method `inSort` sorts the array elements from position 0 to position $\mathtt{l}-1$, using the auxiliary method `insert`. We have compiled this method to JBC and analysed it with COSTA to obtain bounds on the number of read and write accesses to the array `d`. We assume that the two read accesses to $\mathtt{d[i}-1]$ at Lines 9 and 10 are optimised by the compiler to one access using an auxiliary local variable (this will be reflected in the RBR). For `insert(d,i)`, COSTA reports the UBF $2 * \mathtt{nat}(i-1)+3$, which can be explained as follows: the array accesses at Lines 8 and 13 contribute 2; the `while` loop contributes $2 * \mathtt{nat}(i-1)$, which is the product of the number of array accesses 2 by the number of iterations $\mathtt{nat}(i-1)$; and, finally, the access in the loop guard when it evaluates to *false* contributes 1. Here, $\mathtt{nat}(x)$ means $\max(x,0)$. For `inSort(d,l)`, COSTA reports the UBF $\mathtt{nat}(l-1) * (2 * \mathtt{nat}(l-1)+3)$. The factor $\mathtt{nat}(l-1)$ is the number of iterations of the `for` loop. The second factor is the worst-case cost of all iterations, which is the UBF of `insert` except that i is replaced by $l-1$, i.e., by the maximum value that the loop counter `i` can take. ■

Next we describe the steps that COSTA follows to infer the UBF of the above example. In order to abridge the presentation, we have simplified the intermediate results of the analysis, without affecting the correctness of these bounds. In addition, we skip many details that are not necessary to present our contributions in the subsequent chapters. For a comprehensive description of COSTA the reader may refer to [7, 1].

2 Rule-Based Representation

COSTA starts by constructing a CFG for the JBC program. The CFG for the (JBC of the) methods `inSort` and `insert` is depicted in Figure 2.3. Note that, for

```
1   // Precondition: 0 <= l <= d.length
2   void inSort(double[] d, int l) {
3       for (int i=1; i<l; i++)
4           insert(d,i);
5   }
6
7   void insert( double[] d, int i){
8       double x = d[i];
9       while (i > 0 && d[i - 1] < x) {
10          d[i] = d[i - 1];
11          i--;
12      }
13      d[i] = x;
14  }
```

Figure 2.2: Java code of the `inSort` method.

simplicity, the exceptional behaviour is excluded. A square box represents a basic block, which is a sequence of JBC instructions without branching. A diamond represents a branching point. A circled point marks a method end or a loop exit. For the sake of simplicity, we omit instructions in the basic blocks and show only the names of each procedure and the guards of each branch.

The CFG is split into five subgraphs or **procedures**: one for the method `inSort`, one for the `for` loop, one for the method `insert`, and two for the `while` loop (Lines 9 to 12). The `while` loop has two procedures because the condition $\texttt{d[i}-1\texttt{]} < \texttt{x}$ (Line 9) is evaluated only if $\texttt{i} > 0$ is *true*. Each subgraph (or procedure) has one or more control path. A control path represents a jump to a basic block, with a **guard** to specify in what cases this jump can be taken. The procedures for the methods `inSort` and `insert` have one control path with the *true* guard. Procedures that correspond to loops have two control paths: one to exit the loop and one to enter its body. For instance, in the procedure of the `for` loop the exit path is labelled with $\neg(\texttt{i} < \texttt{l})$, and the other one is labelled with $\texttt{i} < \texttt{l}$.

COSTA relies on points-to analysis [127] to resolve virtual method invocations when generating the CFG, since the method to be executed is known only at runtime. In addition, COSTA extracts and isolates each loop as if it were a method, when possible, in which case the exit path is empty instead of including the instructions executed after the loop. Loop extraction [138] is a technique that COSTA uses to allow compositional analyses [7].

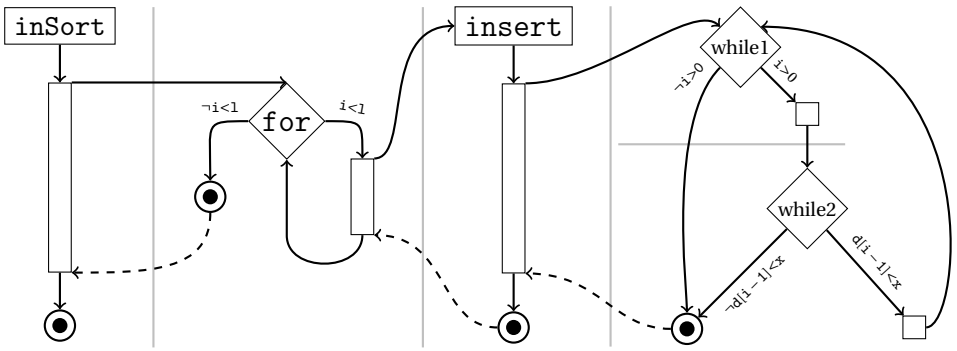

Figure 2.3: Control Flow Graph of the `inSort` method.

Translation to the RBR.

The next step of COSTA is to transform the CFG into an RBR program, that consists of rules of the form

$$p(\bar{x},\bar{y}) \leftarrow \{g\}, b_1, \ldots, b_n.$$

where p is a procedure name, $\bar{x}$ is a list of its input variables, $\bar{y}$ is a list of its output variables, g is a guard, and each b_i is an RBR instruction.

The guard is a logical formula defined over the input variables, and will be omitted if it is equal to *true*. An RBR instruction is either a call $q(\bar{w},\bar{z})$ to a procedure q; or a non-branching instruction that typically corresponds to a JBC instruction. We restrict ourselves to assignments of the form $\tau := e$, where e is an arithmetic expression over variables and array accesses, and τ is either a variable x or an array position $x[i]$. Note that the RBR of COSTA includes many other instructions, e.g., field accesses, object and array creation, etc. However, they are not necessary for explaining our contributions, so we omit the corresponding details.

To translate a CFG to an RBR, each procedure of the CFG is mapped to a procedure in the RBR. Each control path in the CFG is transformed into a rule in the RBR, with a guard that is the same as that of the path. Each interprocedural jump between blocks in the CFG is transformed to a corresponding procedure call. Each instruction in the JBC program is transformed to a corresponding RBR instruction. Variables in the RBR are local to the rule in which they appear, and they typically correspond to either variables in the Java program, or positions in the operands stack of the JVM.

Example 2.2. The RBR of our running example is depicted in Figure 2.4. It contains eight rules that define five procedures, which correspond to the procedures and control paths of the CFG of Figure 2.3.

$inSort(\langle d, l\rangle, \langle\rangle) \leftarrow$
$\quad i := 1,$
$\quad for(\langle d, l, i\rangle, \langle\rangle).$

$for(\langle d, l, i\rangle, \langle\rangle) \leftarrow$
$\quad \{\neg i < l\}.$

$for(\langle d, l, i\rangle, \langle\rangle) \leftarrow$
$\quad \{i < l\},$
$\quad insert(\langle d, i\rangle, \langle\rangle),$
$\quad i := i + 1,$
$\quad for(\langle d, l, i\rangle, \langle\rangle).$

$insert(\langle d, i\rangle, \langle\rangle) \leftarrow$
$\quad x := d[i],$
$\quad while_a(\langle d, i, x\rangle, \langle i\rangle),$
$\quad d[i] := x.$

$while_a(\langle d, i, x\rangle, \langle i\rangle) \leftarrow$
$\quad \{\neg i > 0\}.$

$while_a(\langle d, i, x\rangle, \langle i\rangle) \leftarrow$
$\quad \{i > 0\},$
$\quad y := d[i-1],$
$\quad while_b(\langle d, i, x, y\rangle, \langle i\rangle).$

$while_b(\langle d, i, x, y\rangle, \langle i\rangle) \leftarrow$
$\quad \{\neg y < x\}.$

$while_b(\langle d, i, x, y\rangle, \langle i\rangle) \leftarrow$
$\quad \{y < x\},$
$\quad d[i] := y,$
$\quad i := i - 1,$
$\quad while_a(\langle d, i, x\rangle, \langle i\rangle).$

Figure 2.4: Rule-Based Representation program for method `inSort`.

Procedures *inSort* and *insert* correspond to the Java methods of Figure 2.2, note that their input and output parameters coincide. The rule of *inSort* initialises variable i of the `for` loop and then calls procedure *for*. The rule of *insert* contains one instruction that corresponds to the array access at Line 8 of Figure 2.2, another one to initialise the variable i of the `while` loop, a call to the $while_a$ procedure, and one that corresponds to the array update at Line 13.

The RBR procedures *for*, $while_a$ and $while_b$ correspond to the `for` and `while` loops of the Java program. Each procedure has two rules: one to exit the loop, and one to enter its body. It can be seen that looping behaviour is done using tail-recursion in the RBR. Note that the two accesses to `d[i - 1]` in the `while` loop of method `insert` are optimised to one in the RBR, by storing the corresponding value in the auxiliary variable y (in the second rule of $while_a$). ■

RBR Semantics.

A trace semantics for RBR programs is presented in [10]; we skip the details since they are not important to explain our contributions. Informally, assuming that each RBR instruction is assigned some cost (using some cost model), the cost of an RBR trace t, denoted $rbrcost(t)$, is defined as the sum of all such costs contributed by the instructions executed within t. A function p^+ (resp. p^-) is a UBF (resp. LBF) for an RBR procedure p, if and only if for any (possibly partial) trace t that corresponds to executing p starting from an initial input of size $\bar{v}$, we have $p^+(\bar{v}) \geq rbrcost(t)$ (resp. $p^-(\bar{v}) \leq rbrcost(t)$).

The correctness of the JBC to RBR transformation relies on that there is a one-to-one correspondence between JBC and RBR traces. This means that the cost described above can be seen as the cost of the original JBC program as well.

3 Abstract Cost Rules

The next step of COSTA is the abstract compilation of the RBR program into an ACR program, that consists of rules of the form:

$$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n.$$

where p is a procedure name; $\bar{x}^\alpha$ and $\bar{y}^\alpha$ are respectively lists of its input and output (abstract) variables; and each a_i is an ACR instruction that can be either a procedure call $q(\bar{w}^\alpha, \bar{z}^\alpha)$, a linear constraint over the rule's variables (equality or non-strict inequality), or a cost annotation of the form acquire(e), which indicates that, at the corresponding program point, we accumulate e to the cost.

The abstract variables in an ACR rule are integer variables, and they represent the size of corresponding RBR variables. COSTA uses different size measures depending on the JBC type of the corresponding variable: the size of an `int` variable is its value; the size of an array is its length, the size of an object reference is its path-length [125], and variables of non-integer basic types are abstracted to "free" variables that represent any value, and thus not tracked at all. If the cost depends on these "free" variables, COSTA will not be able to infer a UBF for the ACR program.

The abstract compilation compiles each RBR rule "$p(\bar{x}, \bar{y}) \leftarrow \{g\}, b_1, \ldots, b_m$" into a corresponding ACR rule "$p(\bar{x}^\alpha, \bar{y}^\alpha) \leftarrow a_0, \ldots, a_n$" where: the abstract variables $\bar{x}^\alpha$ and $\bar{y}^\alpha$ correspond to the sizes of $\bar{x}$ and $\bar{y}$ respectively; the guard g is compiled to a corresponding linear constraint a_0, over the abstract input variables $\bar{x}^\alpha$, that overapproximates its behaviour; and each b_i is compiled to one or more consecutive a_j as follows:

- **Size abstraction.** If b_i is a call $q(\bar{w}, \bar{z})$ then it is compiled to $q(\bar{w}^\alpha, \bar{z}^\alpha)$; if it is an assignment of the form $x := e$ where e is a linear expression; then it is compiled to $x^\alpha = e^\alpha$ where e^α is obtained from e by replacing its variables by their corresponding abstract versions; otherwise it is compiled to the constraint *true*, which has no effect on the execution. The size abstraction uses a Static Single Assignment (SSA) transformation [21, §19] to simulate the effect of updates with linear constraints. Note that since COSTA supports several other RBR instructions, e.g., field access, the actual size abstraction is more elaborated. However, these details are not important for our contributions since we will assume a given ACR program independently from where it comes.
- **Cost annotation.** The selected **cost model** is applied to b_i to generate a cost annotation of the form acquire(e) that describes its cost. We omit these annotations from the ACR when $e = 0$. COSTA has several cost models, e.g.,

$inSort(\langle l_0\rangle,\langle\rangle) \leftarrow$
  $i_1 = 1,$
  $for(\langle l_0, i_1\rangle,\langle\rangle).$

$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 \geq l_0.$

$for(\langle l_0, i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 + 1 \leq l_0,$
  $insert(\langle i_0\rangle,\langle\rangle),$
  $i_1 = i_0 + 1,$
  $for(\langle l_0, i_1\rangle,\langle\rangle).$

$insert(\langle i_0\rangle,\langle\rangle) \leftarrow$
  acquire(1),
  $while_a(\langle i_0\rangle,\langle\rangle),$
  acquire(1).

$while_a(\langle i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 \leq 0.$

$while_a(\langle i_0\rangle,\langle\rangle) \leftarrow$
  $i_0 \geq 1,$
  acquire(1),
  $while_b(\langle i_0\rangle,\langle\rangle).$

$while_b(\langle i_0\rangle,\langle\rangle) \leftarrow$
  true.

$while_b(\langle i_0\rangle,\langle\rangle) \leftarrow$
  true,
  acquire(1),
  $i_1 = i_0 - 1,$
  $while_a(\langle i_1\rangle,\langle\rangle).$

Figure 2.5: Abstract Cost Rules (ACR) program for method `inSort`.

for counting the number of executed instructions, calls to a specific method, or occupied memory.

The abstract compilation ends by removing from the ACR program those variables that do not affect the cost [8]. This results in concise ACR programs that are more efficient to analyse.

Example 2.3. The ACR program obtained by applying the abstract compilation to the RBR program of Figure 2.4 is depicted in Figure 2.5. The size abstraction maps the RBR variables l and i to the ACR variables l and i that appear with subscripts introduced by SSA (see explanation in the next paragraph). The RBR variables x, y, d, and the output variables of procedures $while_a$ and $while_b$ are omitted because they do not affect the cost. Following Example 2.1, we use a cost model that counts the number of array accesses.

The relation between each RBR rule and its corresponding ACR rule is clear. Note that the assignment $i := i + 1$ in the second RBR rule of procedure *for* was compiled to the constraint $i_1 = i_0 + 1$. Here i_0 and i_1 represent those values of variable i before and after executing the instruction. Note also the use of the cost annotations acquire(1) to account 1 as the cost of each array access. ■

ACR Semantics.

Let us briefly summarise the important details of the semantics of ACR programs [18, 10]. A state s takes the form $\langle \psi, \bar{a} \rangle$, where $\bar{a}$ is a sequence of ACR instructions pending for execution, and ψ is a constraint over the variables in $\bar{a}$ and possibly other existentially quantified variables. The store ψ collects all constraints encountered during an execution. An execution starts from an initial state $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y}) \rangle$, where $\bar{v}$ is a sequence of integers, and proceeds ac-

cording to the following rules:

$$\frac{q(\bar{x},\bar{y}) \leftarrow \bar{a}' \in P}{\langle \psi, q(\bar{x},\bar{y})\cdot\bar{a}\rangle \overset{0}{\rightsquigarrow} \langle \psi, \bar{a}'\cdot\bar{a}\rangle} \qquad \frac{\psi\wedge\varphi \not\models \mathit{false}}{\langle \psi, \varphi\cdot\bar{a}\rangle \overset{0}{\rightsquigarrow} \langle \psi\wedge\varphi, \bar{a}\rangle} \qquad \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathsf{acquire}(e)\cdot\bar{a}\rangle \overset{v}{\rightsquigarrow} \langle \psi, \bar{a}\rangle}$$

These rules define a transition relation $s_1 \overset{v}{\rightsquigarrow} s_2$, meaning that there is a transition from s_1 to s_2 that costs v units. The rule on the left handles procedure calls, it (nondeterministically) selects a rule from the program P that matches the call, and adds its instructions $\bar{a}'$ to the sequence of pending instructions. Variables in $\bar{a}'$ (except $\bar{x}\cup\bar{y}$) are renamed such that they are different from the variables in $\bar{a}$ and ψ. The rule in the middle handles constraints by adding them to the store, if the resulting state is satisfiable. The rule on the right handles cost annotations, it evaluates e to a non-negative value v and labels the corresponding transition with v.

The execution stops when no rule is applicable, which happens when the execution reaches a) a final state $\langle \psi', \epsilon\rangle$ where ϵ is the empty sequence; or b) a blocking state $\langle \psi', \varphi\cdot\bar{a}\rangle$ where $\varphi\wedge\psi' \models \mathit{false}$. A trace t is a finite or infinite sequence of states, in which there is a valid transition between each pair of consecutive states. Traces that end in a final state and infinite traces are called complete. Namely, we exclude traces that end in a blocking state because such traces correspond to no RBR trace. The cost of a trace t, denoted $\mathit{acrcost}(t)$, is defined as the sum of all cost labels in its transitions. A function p^+ (resp. p^-) is a UBF (resp. LBF) for procedure p, if for any input $\bar{v}$ and complete trace t that starts in $\langle \bar{x}=\bar{v}, p(\bar{x},\bar{y})\rangle$, we have $p^+(\bar{v}) \geq \mathit{acrcost}(t)$ (resp. $p^-(\bar{v}) \leq \mathit{acrcost}(t)$).

The correctness of the RBR to ACR transformation relies on the idea that any RBR trace has a corresponding ACR trace with the same cost. Thus, a valid UBF or LBF for the ACR program is also valid for the RBR program.

4 Cost Relation Systems

In the next phase, the ACR program is transformed into a CRS, which consists of equations of the form:

$$C(\bar{x}) = e + D_1(\bar{y}_1) + \cdots + D_r(\bar{y}_r)\ ,\varphi$$

where $C, D_1, \ldots, D_r$ are relation symbols (like procedure names); $\bar{x}, \bar{y}_1, \ldots, \bar{y}_r$ are variables; φ is a conjunction (often written as a set) of linear constraints over these (and maybe other) variables; and e is a **cost expression** that adheres to the following grammar:

Cost Expression	$\mathsf{exp} ::= \mathsf{bexp} \mid \mathsf{oexp}$
Basic Expression	$\mathsf{bexp} ::= q \mid \mathsf{nat}(l) \mid \log_m(\mathsf{nat}(l)+1) \mid m^{\mathsf{nat}(l)}-1$
Compound Expression	$\mathsf{oexp} ::= \mathsf{exp}+\mathsf{exp} \mid \mathsf{exp}*\mathsf{exp} \mid \max(\mathsf{exp}_1,\ldots,\mathsf{exp}_n)$

in which $q \in \mathbb{Q}^+$, $m > 1 \in \mathbb{Z}^+$, l is a linear expression over the variables that appear in the equation, and $\mathsf{nat}(l) = \max(l, 0)$. Intuitively, the above equation states that the cost of C, with respect to the input $\bar{x}$, is e plus the sum of the costs of each D_i with respect to the input $\bar{y}_i$. The linear constraint φ specifies the values of $\bar{x}$ for which the equation is applicable, and defines relations among the different variables. Note the declarative nature of this intuition. Since CRSs originate from ACR programs, we may think of $C, D_1, \ldots, D_r$ as (nondeterministic) procedures, and say that C calls $D_1, \ldots, D_r$.

All equations that share the same left-hand side, e.g., $C(\bar{x})$, define the **cost relation** (CR for short) $C(\bar{x})$. Thus, a CRS consists of several CRs. When a CR uses only one relation symbol, formally $D_i = C$ for all $1 \leq i \leq r$, we call it **standalone** CR, that is, it depends on no other CR.

The ACR to CRS transformation maps each ACR procedure $p(\bar{x}, \bar{y})$ to a CR $p(\bar{x})$, defined over the same input $\bar{x}$, but without the output $\bar{y}$. This is done by transforming each ACR rule "$p(\bar{x}, \bar{y}) \leftarrow a_1, \ldots, a_n$" to a corresponding equation "$p(\bar{x}) = e + q_1(\bar{w}_1) + \cdots + q_n(\bar{w}_n), \varphi$" as follows:

1. the cost expression e is computed as the sum of all expressions e_i in instructions $\mathsf{acquire}(e_i)$ that appear in the ACR rule. Thus, each e_i must be a valid cost expression;
2. each call $q_j(\bar{w}_j, \bar{z}_j)$ in the ACR rule is transformed to a call $q_j(\bar{w}_j)$ in the equation, with the same input but without the output; and
3. the constraint φ is a conjunction of all linear constraints in the ACR rule.

In the second step above, when $q_j(\bar{w}_j, \bar{z}_j)$ has a non-empty list of output variables, removing these variables may result in a loss of information that is crucial for inferring a corresponding UBFs. To reduce the effect of this loss, COSTA uses an inter-procedural **value analysis**[1] to compute a postcondition for each ACR procedure, and then adds this postcondition to φ to compensate on the removal of the output variables. This postcondition is a conjunction of linear constraints that describe the relation between input and output variables, in any execution of the corresponding procedure. It is worth emphasising that postconditions in COSTA are *linear*, and thus they cannot model nonlinear relations. Our contribution of Chapter 4 improves in this direction.

Example 2.4. The CRS generated from the ACR program of Figure 2.5 is depicted in Figure 2.6. It has eight equations that define five CRs. The cost expression in each equation coincides with the cost annotations of its corresponding ACR rule: 2 for *insert*, 1 in the recursive rules of $while_a$ and $while_b$, and 0 elsewhere.

[1]The notion of value analysis is related to that of size analysis [123]. A size analysis infers relations not only between the values of numeric variables, but also between the sizes of data structures.

$$
\begin{array}{lcll}
inSort(l_0) & = & for(l_0, i_1) & \{i_1 = 1\} \\
for(l_0, i_0) & = & 0 & \{i_0 \geq l_0\} \\
for(l_0, i_0) & = & insert(i_0) + for(l_0, i_1) & \{i_0 + 1 \leq l_0, i_1 = i_0 + 1\} \\
insert(i_0) & = & 2 + while_a(i_0) & \{\} \\
while_a(i_0) & = & 0 & \{i_0 \leq 0\} \\
while_a(i_0) & = & 1 + while_b(i_0) & \{i_0 \geq 1\} \\
while_b(i_0) & = & 0 & \{\} \\
while_b(i_0) & = & 1 + while_a(i_1) & \{i_1 = i_0 - 1\}
\end{array}
$$

Figure 2.6: Cost Relation System (CRS) for the `inSort` method.

The calls in each equation are as those in the ACR, and the constraints in each equation are those which appear in the corresponding ACR rule. Note that, in this example, there was no need to perform value analysis because the ACR procedures in Figure 2.5 have no output variables. ■

CRS Semantics.

The semantics of the CRS is based on the notion of evaluation trees [5]. Let us denote a (possibly infinite) tree by $node(q, \langle T_1, \ldots, T_k \rangle)$, where $q \in \mathbb{Q}^+$ is the value of the root and $T_1, \ldots, T_k$ are subtrees. Given a CR C and a concrete input $\bar{v}$, we say that $node(v_e, \langle T_1, \ldots, T_k \rangle)$ is an **evaluation tree** for $C(\bar{v})$ if and only if there exists an equation "$C(\bar{x}) = e + \sum_{j=1}^{k} D_j(\bar{y}_j), \varphi$" and an assignment σ, for the variables of that equation, such that:

1. $\sigma(\bar{x}) = \bar{v}$ and σ is a satisfying assignment for φ;
2. e is evaluated to v_e when considering the assignment σ; and
3. each T_i is an evaluation tree for $C(\sigma(\bar{y}_i))$.

Intuitively, when viewing C as a procedure, an evaluation tree can be seen as a recursion tree where the call $C(\bar{v})$ is evaluated as follows: we pick an equation that defines C and an assignment σ that satisfies φ; we evaluate e into v_e, and recursively call all $C(\sigma(\bar{y}_i))$ simultaneously. Note that an evaluation tree can be infinite. Note also that $C(\bar{v})$ might have several evaluation trees, due to the nondeterminism induced by choosing an equation for C and a satisfying assignment σ for φ. Note that the internal nodes of an evaluation tree correspond to the application of recursive equations, and the leaves correspond to the application of nonrecursive ones.

The set of all evaluation trees for $C(\bar{v})$ is denoted by $Trees(C(\bar{v}))$, and the set of all possible costs is $Answers(C(\bar{v})) = \{\mathsf{Sum}(T) \mid T \in Trees(C(\bar{v}))\}$, where $\mathsf{Sum}(T)$ is the sum of all nodes of T. Then, a function C^+ (resp. C^-) is said to be a UBF

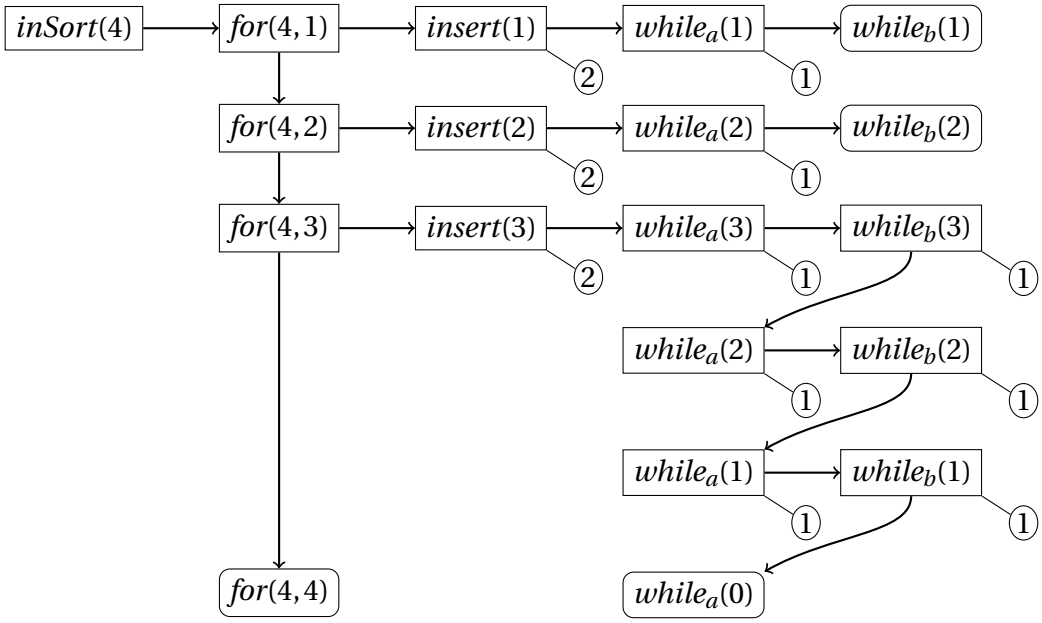

Figure 2.7: Evaluation Tree for a call to the CR *inSort* in Figure 2.6

(resp. LBF) for the CR C if and only if for any input $\bar{v}$ and $c \in \mathit{Answers}(C(\bar{v}))$, we have $C^+(\bar{v}) \geq c$ (resp. $C^-(\bar{v}) \leq c$).

Example 2.5. An evaluation tree for a call $\mathit{inSort}(4)$, using the CRS of Figure 2.6, is depicted in Figure 2.7. Each rectangle represents a call in the CR with some input values; each circle indicates the cost contributed by the equation that was chosen to resolve that call (as v_e in the explanation above); and directed edges represent calls to other CRs. The cost of this evaluation tree is 11, which coincides with the sum of all circles. ■

The correctness of the ACR to CRS transformation relies on the idea that every ACR trace has a corresponding evaluation tree with the same cost [10]. Thus, any valid UBF or LBF for the CRS is also valid for the ACR program.

5 Closed-form Upper Bound Functions

In the last step COSTA solves the generated CRS into UBFs or LBFs, one for each CR. For this, it relies on its subsystem PUBS [5, 14]. We first describe how it computes UBFs, and, at the end of this section, we comment on the case of LBFs, which is less important for presenting our contributions.

The solving procedure of PUBS is designed to handle CRSs in which all recursions are direct, i.e., without mutual calls between different CRs. To handle CRSs that do not meet this requirement, they are first transformed into such form [87, 140]. Essentially, this is done by considering each group of mutually recursive CRs and unfolding all CRs in such a group into one CR.

Example 2.6. Consider the CRS of Figure 2.6. This CRS has two groups of recursive CRs: one for the CR *for*, which already is in direct recursive form, and one for the CRs $while_a$ and $while_b$, which are mutually recursive. To transform these last CRs into direct recursion, we unfold the definition of $while_b$ into that of $while_a$. This results in a new definition for the CR $while_a$:

$$\begin{array}{ll} while_a(i_0) = 0 & \{i_0 \leq 0\} \\ while_a(i_0) = 1 & \{i_0 \geq 1\} \\ while_a(i_0) = 2 + while_a(i_1) & \{i_0 \geq 1, i_1 = i_0 - 1\} \end{array}$$

The first equation is as the first one of old $while_a$. The second and third equations come from unfolding the call to $while_b$, in the second equation of old $while_a$, using the two equations of $while_b$. The cost expression in the third new equation is the sum of the cost expressions in the old recursive equations of $while_a$ and $while_b$, and its constraint includes those from both equations. ■

In what follows, we may assume that the input CRS is in direct recursive form. The solving procedure of PUBS is an iterative procedure that solves one CR at a time. In particular, in each iteration it solves a stand-alone CR into a corresponding UBF, and then replaces any call to this stand-alone CR by its UBF, thus generating more stand-alone CRs to be solved. This process continues until all CRs are solved. Note that if the CRS is in direct recursion then there is at least one stand-alone CR to start the process, e.g. $while_a$. In what follows we describe how PUBS solves a stand-alone CR into a UBF.

Solving Stand-Alone CRs.

Assume a given stand-alone CR C with n equations, and 1) let bf be the maximum number of recursive calls in any equation of C, i.e., the maximum branching factor of the corresponding evaluation trees; 2) let $e_1, \ldots, e_k$ be the cost expressions of the nonrecursive equations of C; and 3) let $e_{k+1}, \ldots, e_n$ be the cost expressions of the recursive equations of C. PUBS solves C into a UBF by constructing a pessimistic evaluation tree, whose cost is bigger than the cost of any evaluation tree of C, as we explain next. Recall first that, for any evaluation tree, the cost of any internal node is an instance of e_i with $k+1 \leq i \leq n$, and that the cost of any leaf is an instance of e_j with $1 \leq j \leq k$. Now suppose that we have the following:

- a cost expression $h(\bar{x})$ that bounds the height of any evaluation tree. That is, for any input $\bar{v}$ and any evaluation tree $T \in \mathit{Trees}(C(\bar{v}))$, $h(\bar{v})$ is larger than the height of T; and
- a cost expression $\hat{e}_i(\bar{x})$, for each $1 \leq i \leq n$, that bounds the contributions of the i-th equation. Namely, for any input $\bar{v}$ and any evaluation tree $T \in \mathit{Trees}(C(\bar{v}))$, $\hat{e}_i(\bar{v})$ is larger than the cost of any node in T that corresponds to the i-th equation.

Now we construct a pessimistic evaluation tree T (parametric in the input $\bar{x}$) as follows:

1. T is a complete tree, with branching factor bf and height $h(\bar{x})$;
2. each leaf of T has cost $\max(\hat{e}_1(\bar{x}),\ldots,\hat{e}_k(\bar{x}))$; and
3. each internal node of T has cost $\max(\hat{e}_{k+1}(\bar{x}),\ldots,\hat{e}_n(\bar{x}))$.

The number of leaves $\mathscr{L}$ in T is $bf^{h(\bar{x})}$, and the number of internal nodes $\mathscr{N}$ is $\frac{bf^{h(\bar{x})}-1}{bf-1}$ if $bf > 1$ and $h(\bar{x})$ otherwise. The cost of T is

$$C^+(\bar{x}) = \mathscr{L} * \max(\hat{e}_{k+1}(\bar{x}),\ldots,\hat{e}_n(\bar{x})) + \mathscr{N} * \max(\hat{e}_1(\bar{x}),\ldots,\hat{e}_k(\bar{x}))$$

which is a valid UBF for C as well.

For automatically inferring $h(\bar{x})$, PUBS relies on the use of linear ranking functions [26], which are intensively used to bound the number of iterations of loops. It is easy to see that the height of a tree corresponds to the number of consecutive recursive call in C, which is a form of a loop. For automatically inferring $\hat{e}_i(\bar{x})$, PUBS relies on a **maximisation** procedure that is based on invariants generation. The details are not important to explain our contributions, the interested reader may find these details in [5].

Example 2.7. Let us demonstrate the different steps of PUBS on the CRS of Figure 2.6, assuming that it has been transformed already to direct recursive form as in Example 2.6:

- We start by solving CR while_a of Example 2.6, which is the only stand-alone CR. The height of its evaluation trees is bounded by $h(i_0) = \mathsf{nat}(i_0)$, and the cost expressions of the nonrecursive and recursive equations are (trivially) maximised to 1 and 2 respectively. Therefore, we get the UBF $\mathit{while}_a{}^+(i_0) = 2 * \mathsf{nat}(i_0) + 1$.
- Substituting $\mathit{while}_a{}^+(i_0)$ in the nonrecursive CR *insert* results in the stand-alone CR defined with this only equation:

  $$\mathit{insert}(i_0) = 2 + 2 * \mathsf{nat}(i_0) + 1 \quad \{\}$$

  which is trivially solved to the UBF $\mathit{insert}^+(i_0) = 2 * \mathsf{nat}(i_0) + 3$.

- Substituting $insert^+(i_0)$ in CR *for* results in the following stand-alone CR:

$$\begin{array}{ll} for(l_0, i_0) = 0 & \{i_0 \geq l_0\} \\ for(l_0, i_0) = 2 * \mathsf{nat}(i_0) + 3 + for(l_0, i_1) & \{i_0 + 1 \leq l_0, i_1 = i_0 + 1\} \end{array}$$

  The height of its evaluation trees is bounded by $h(l_0, i_0) = \mathsf{nat}(l_0 - i_0)$. The cost expression $2 * \mathsf{nat}(i_0) + 3$ is maximised to $\hat{e}(l_0, i_0) = 2 * \mathsf{nat}(l_0 - 1) + 3$ because the value of i_0 can be at most $l_0 - 1$, where l_0 refers to the initial value with which *for* is called, not to the parameter l_0. Therefore, we get the UBF $for^+(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.
- Substituting this UBF in *inSort* results in the following stand-alone CR:

$$inSort(l_0) = \quad \mathsf{nat}(l_0 - i_1) * (2 * \mathsf{nat}(l_0 - 1) + 3) \quad \{i_1 = 1\}$$

  The height of any evaluation tree is 0 since it is nonrecursive, and the maximisation of $\mathsf{nat}(l_0 - i_1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ results in $\mathsf{nat}(l_0 - 1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ since $i_1 = 1$. Therefore, for this CR we get the UBF $inSort^+(l_0) = \mathsf{nat}(l_0 - 1) * (2 * \mathsf{nat}(l_0 - 1) + 3)$.

Note that the UBFs are asymptotically tight, in the sense that the worst-case cost of `inSort` is quadratic in the parameter `l`. However, they are not functionally tight, because the overapproximation is too coarse. ■

Inference of LBFs in PUBS is done as described in [14]. It relies on approximating the behaviour of the input CRS by a corresponding system of RRs, and then using computer algebra systems to solve it. For example, it would infer the LBF $inSort^-(l_0) = 3 * \mathsf{nat}(l_0 - 1)$, which corresponds to the case in which the input array is sorted, and thus the body of `while` loop is never executed.

3 | Asymptotic Closed-Form Bounds

In this Chapter we describe a method to transform a bound function into a reduced asymptotic form, and how to use this transformation to build an **asymptotic cost analyser**, which directly computes bounds in asymptotic form. This contribution has been published in the following article:

> ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009

Overview

Asymptotic notations are used to succinctly describe how the cost of a program scales with the size of its input. The key observation behind them is that:

> "Although we can sometimes determine the exact running time of an algorithm [...], the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself." [45, Chapter 3]

Asymptotic notations ignore those constants and lower terms, and focus instead on the proportionality, up to some scalar and for large input, between the worst-case (or best-case) cost of a program and a function of (the sizes of) the input of the program. This proportionality or **asymptotic equivalence** is a condition so lax that the worst-case (or best-case) cost can be asymptotically equivalent to infinitely many functions; yet, among them, there is a simple one like 1, n^2, $n \log n$, or 2^n, which we call the **asymptotic form**. A bound functions on the program's cost in asymptotic form is called an **asymptotic closed-form bound**[1], i.e., an asymptotic LBF (ALBF for short) or an asymptotic UBF (AUBF for short).

For some applications, we need a bound with all the *multiplicative constants and lower-order terms*, i.e., non-asymptotic. For instance, to split tasks among processors [96], to decide if and where to migrate a process [56], to avoid

[1] Also called the asymptotic complexity of the program.

running out of battery [88], or to avoid a heap overflow [13], we cannot rely on information that tells us that the cost is eventually linear by some constant. In general, we need non-asymptotic bounds for the verification, compilation, and optimizing execution of programs.

In the above scenarios, we usually consider programs whose development has terminated already. However, we should not wait until the end of the development process to (automatically) check whether a program meets its (cost) specifications or not, we should try to apply this check frequently in order to detect performance bugs during the development process as early as possible. This can be done, for instance, by running the analysis and providing the programmers with the inferred bounds; also, they may write in the code an assertion on the cost of a method, and run the analyser to verify it. In such cases, using asymptotic bounds have clear advantages over non-asymptotic ones:

- As it is more concise, an asymptotic bound is easier to read than a non-asymptotic one. Thus, using asymptotic bounds improves the usability for the programmers of the analyser.
- The programmers' main concern before writing the program is its scalability, about which they may have an idea clear enough to write it down as an assertion of an asymptotic bound. Thus, these assertions are easier to figure out. Instead, if they were asked to write an assertion of a non-asymptotic bound, they might not know what coefficients and minor expressions to put in it, because such information depends on implementation details that they can not predict before writing the program.
- The non-asymptotic bound for a program under development is volatile, because the *multiplicative constants and lower-order terms* can change due to any change in the program, in the compiler, or in the libraries. As these things happen often during the development, if non-asymptotic bounds were used then there would be too many and too frequent notifications. Asymptotic bounds are more stable, since only major changes or improvements have an impact on the program's asymptotic complexity.
- Once the program is written, the programmers have to improve its performance and scalability. A guiding principle for this task is to focus on the program's bottlenecks, that is the most frequent or expensive operations:

  > "Programmers waste enormous amounts of time thinking about [...] the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. [...] A good programmer [...]

> will be wise to look carefully at the critical code; but only after that code has been identified." [89, page 268]

We can use an asymptotic cost analyser to identify that critical code –the bottlenecks– at which we should direct the optimisation effort.

- Since asymptotic bounds are smaller, an asymptotic cost analyser can be faster and more scalable than a non-asymptotic one. Having a faster analyser is especially relevant for integrating it in an interactive tool that compiles and analyses the code just as it is written.

Thus, an asymptotic cost analyser can be an easy to use tool for analysing and improving the performance of the program.

In Section 3.1, we recall the relevant properties of the asymptotic notations, and we adapt these notations to the cost expressions of PUBS. In Section 3.2 we describe our asymptotic transformation, which is an algorithm to transform a cost expression into another one that is asymptotically equivalent. In Section 3.3 we describe our asymptotic comparison, which is an algorithm to prove that a cost expression is asymptotically greater than another one. In Section 3.4 we discuss how to use this transformation to build a cost analyser that directly generates asymptotic bounds. In Section 3.5 we discuss an experimental evaluation of the techniques. In Section 3.6 we overview related work, and in Section 3.7 we describe some further details that can be found in the article.

1 Asymptotic Notations for Cost Expressions

In this Section we recall the critical properties of the asymptotic notations for univariate functions, and we adapt these notations to the cost expressions of PUBS (see Section 2.4).

The asymptotic notations big Omicron (O) and big Theta (Θ), are commonly defined as follows. If g is a univariate function, that is a function in $\mathbb{N} \mapsto \mathbb{R}^+$, then $O(g)$ and $\Theta(g)$ denote sets of univariate functions. In particular, $f \in O(g)$ if, for large values of the input n, $f(n) \leq c_u g(n)$ for some constant $c_u > 0$. Similarly, $f \in \Theta(g)$ if $c_l g(n) \leq f(n) \leq c_u g(n)$ for large values of the input n and for some coefficients $c_l, c_u > 0$.

These notations are frequently used to compare functions, so we can read $f \in O(g)$ as "f is asymptotically smaller than g", or "g is an asymptotic upper bound of f"; and we can read $f \in \Theta(g)$ as "f is asymptotically equivalent to g", or "g is an asymptotically tight bound of f". In this sense, $f \in \Theta(g)$ is an equivalence relation and $f \in O(g)$ is an order relation [2], which have the following properties:

[2] Some authors [62, §4.1] [112, §9.1] use $f \preceq g$ and $f \approx g$ as shorthands of $f \in O(g)$ and $f \in \Theta(g)$.

1. The exact asymptotic order Θ of a function does not change if we multiply it by a constant d, formally $\Theta(d * f) = \Theta(f)$.
2. The product of functions is monotonic with respect to the order relation defined by O. Formally, if $f' \in O(f)$ and $g' \in O(g)$ then $f' * g' \in O(f * g)$.
3. The exact asymptotic order of a sum of two terms is that of its dominating term, formally if $f \in O(g)$ then $\Theta(f + g) = \Theta(g)$.

These properties are essential for automatically transforming a function to its asymptotic form, or for comparing two functions, because we can use them to deduce some easy to implement rules for comparing two expressions. For instance, we get the rule to compare the complexity of two polynomials by comparing their degrees, or we can compare two exponential expressions by comparing their bases[3]. Due to these properties, Θ of a simple function (like n or n^2) may contain more complex functions.

Example 3.1. Let $f(n) = 3n^2 + n\log_3 n + 5n$, we can omit the multiplicative coefficients 3 and 5 due to the first property. Then, since both n and $n\log_3 n$ are in $O(n^2)$, using the third property we get $f \in \Theta(n^2)$. ■

The classical definition refers to univariate functions, i.e., functions with one natural argument. However, a bound can be a multivariate function, that is, a function of many natural variables, either because the program has many input variables [67, 66, 86], or because the bound function is defined over several size measures for each input [75, 63, 6]. Since there is no standard definitions for the multivariate functions nor a proof of their properties, many people use the asymptotic notations with multivariate functions as if they had the same properties as the univariate ones. However, Howell [79] shows that some properties of the asymptotic notations, which are crucial for building an asymptotic transformation and comparison method, do not hold when the notations are generalised to all multivariate functions.

Fortunately, the work of Howell shows two conditions on a definition of the asymptotic notations for multivariate functions, under which the crucial properties are preserved:

- The asymptotic notations must be restricted to multivariate functions that are monotonic (non-decreasing) on all their inputs; and
- The asymptotic notations must refer to the relation between the functions for input vectors in which all components are large.

Following these conditions, our definition [2, Definition 2] of the the asymptotic notations for multivariate functions generalises the classical definitions for univariate functions. This is done by replacing the symbol n of an input

[3] Formally, for $0 \le a \le b$, we have that $n^a \in O(n^b)$, and for $1 < a \le b$ we get that $a^n \in O(b^n)$.

variable by a symbol $\bar{n}$ of an input vector. For instance, for any multivariate function g in $\mathbb{N}^m \mapsto \mathbb{R}^+$, $O(g)$ denotes the set of functions f in $\mathbb{N}^m \mapsto \mathbb{R}^+$ for which there is a real $c_u > 0$ such that $f(\bar{n}) \leq c_u g(\bar{n})$, for large input vectors. Here, large input vector means one in which all components are large.

Example 3.2. Consider the multivariate function $f(A,B) = A*(2*B+3)$. Using the above properties we can deduce that $2*B+3 \in \Theta(B)$, and therefore $f(A,B) \in \Theta(A*B)$. ■

Since cost expressions are functions over integer variables, before we use the above definition of asymptotic notations, we need first to transform them into (or approximate them by) ones over natural variables, that moreover are monotonic on each component. For that, we need to identify some elements of cost expression that we can abstract to natural variables, such that the resulting expression is monotonic on those elements: we chose the nat subexpressions as those elements (see Section 2.4). The next example explains this choice.

Example 3.3. Consider the cost expression $\mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ of Example 2.7. Since it is monotonic on its nat components, we can abstract it to the function $A*(2*B+3)$, where A and B are natural variables that abstract $\mathsf{nat}(l_0 - i_0)$ and $\mathsf{nat}(l_0 - 1)$ respectively. We cannot abstract this cost expression by mapping l_0 and i_0 to natural variables since the cost expression is not monotonic on i_0. ■

We define the asymptotic notations for cost expressions e as a multivariate function f_e, where every natural argument of f_e corresponds to a nat subexpression of e. This abstraction should map different nat subexpressions to different variables of f_e, but the same natural variable should be used for nat subexpressions that appear more than once. We call f_e the nat-free abstraction of e, and its variables the nat variables.

Example 3.4. In Example 3.3, for $e \equiv \mathsf{nat}(l_0 - i_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$, we used the nat-free abstraction $f_e \equiv A*(2*B*+3)$. ■

The asymptotic relation between cost expressions is defined just as the relation between their nat-free abstractions. Namely, if f_1 and f_2 are the nat-free abstractions of e_1 and e_2 respectively, then $e_1 \in O(e_2)$ if $f_1 \in O(f_2)$.

Using the nat-free abstraction to define the asymptotic notations carries a basic intuition in complexity analysis [15, 86, 66]. The asymptotic complexity of the program is related directly to the number of iterations that each loop performs. As we have seen in Section 2.5, the number of iterations of a loop is bounded by a cost expression $\mathsf{nat}(l)$ where l is a linear ranking function. Thus, the nat expressions are the main components that affect the cost of the program, so we may define the complexity of the program as the one in which the nat subexpressions take large values.

2 Asymptotic Transformation

In general, an asymptotic transformation [128, 2] is a procedure that takes a cost expression e and returns a simpler and asymptotically equivalent expression e'. Such a transformation consists of a series of steps to remove redundant operands and operations: if a and b are two expressions and $\diamond$ is an operation, we say that b is a redundant expression if $a \diamond b \in \Theta(a)$, so b does not change the asymptotic complexity of the expression. In our case, our asymptotic transformation takes as input a bound given as a pair $\{e, \varphi\}$, where e is a cost expression and φ is a a conjunction of linear constraints that restricts the values of the variables of e. In the following we say that φ is a context constraint. Our asymptotic transformation works in three steps:

1. **Computing the** nat**-free abstraction** f_e as described in the previous section. However, to avoid generating too many nat-variables, we first modify e such that proportional nat sub-expressions are replaced by a common one. Thus, they will be abstracted to the same nat-variable.
2. Transforming the nat-free expression f_e into an **asymptotic normal form** by performing the following operations: *1*) replace every max operation $\max(e_1, e_2)$ by $e_1 + e_2$; *2*) remove all multiplicative constants; *3*) replace each exponential cost expression $b^A - 1$ by b^A; *4*) replace each logarithmic cost expression $\log_b(A+1)$ by $\log A$; and *5*) rewrite the expression as a sum of products of basic cost expressions (Section 2.4) .
3. **Remove redundant terms** from the normalised cost expression. Those addends in the sum that are asymptotically smaller than others, and thus do not modify the asymptotic order of the sum.

The first two steps are easy to implement, since they are just two syntactic transformations. For the last step it is necessary to use an asymptotic comparison that we describe in the next section.

Example 3.5. Consider the cost expression $e = \mathsf{nat}(l_0) * (2 * \mathsf{nat}(l_0 - 1) + 3)$ of Example 2.7. Our transformation proceeds as follows: (*1*) $\mathsf{nat}(l_0 - 1)$ is replaced by $\mathsf{nat}(l_0)$ and then the nat-free abstraction results in $f_e = A * (2 * A + 3)$; (*2*) the constants in f_e are removed, and then f_e is transformed into the normal form $A^2 + A$; (*3*) the redundant term A is removed to obtain A^2 . To get A^2 in terms of the original nat expressions, we undo the nat-free abstraction which results in $\mathsf{nat}(l_0)^2$. ■

3 Asymptotic Comparison

Our asymptotic comparison procedure takes two nat-free expressions e_1 and e_2, together with a context constraint φ, and tries to prove that $e_1 \in O(e_2)$. It is based on the notions of asymptotic subsumption and asymptotic weight:

- **Asymptotic subsumption**. If A and B are nat-variables that correspond to $\mathsf{nat}(l_a)$ and $\mathsf{nat}(l_b)$ respectively, we say that A subsumes B (modulo φ), written as $A \succcurlyeq B$, if φ implies that $\mathsf{nat}(l_b) \in O(\mathsf{nat}(l_a))$.
- **Asymptotic weight**. It is the key measure to compare the growth of each expression, such as the base of the exponential factors, or the degree of the polynomials or poly-logarithmic[4] factors. Comparing the weights of expressions is done by first comparing the exponential base, then the degree of the polynomials and finally the degree of the poly-logarithms.

These notions provide a straightforward way to compare two cost expressions e_1 and e_2, modulo a context constraint φ:

(R1) To prove that $P \in O(b)$, where b is a basic cost expression and P is a product of basic expressions, we have to check 1) that the nat-variable of b subsumes (modulo φ) every nat-variable of P; and 2) that b has a greater asymptotic weight than P.

(R2) To prove that $P \in O(Q)$, where P and Q are products of basic cost expressions, we try to factorise P into k sub-products $P = p_1 * p_2 * \ldots * p_k$, such that there exists k different factors b_i in Q verifying $p_i \in O(b_i)$.

(R3) To prove that $S \in O(T)$, where S and T are sums of products of basic cost expressions, we only have to find, for each addend a in S, an addend a' in T for which $a \in O(a')$.

Let us see an example of the above comparison rules.

Example 3.6. Let us compare the nat-free expressions $e_1 \equiv 2^B C^2 + A^3 D + D^2 A$ and $e_2 \equiv B^7 C + A^2 \log^2 B + B^2 C^2 + D^2 \log C$, where $A = \mathsf{nat}(x+y)$, $B = \mathsf{nat}(x)$, $C = \mathsf{nat}(y)$, and $D = \mathsf{nat}(z)$. Assuming the context constraint $\varphi \equiv \{x \geq 0, y \geq 0, z \geq y\}$, we can deduce the subsumption relations $A \succcurlyeq B, A \succcurlyeq C, D \succcurlyeq C$. Now we can build a proof of $e_2 \in O(e_1)$ as follows:

- Using (R1) we deduce the following relations:

$$B^7 \in O\left(2^B\right) \quad A^2 \log^2 B \in O\left(A^3\right) \quad B^2 C \in O\left(A^3\right) \quad C \in O\left(C^2\right)$$
$$C \in O(D) \qquad \log C \in O(A) \qquad D^2 \in O\left(D^2\right)$$

  In the case of $A^2 \log^2 B \in O(A^3)$, we have that A subsumes both A and B, and the asymptotic weight of A^3 is greater than that of $A^2 \log^2 B$.

[4] A poly-logarithm is an expression like $(\log A)^2$.

- Using (R2) we deduce the following relations:

$$B^7C \in O(2^BC^2) \qquad B^2C^2 \in O(A^3D)$$
$$A^2\log^2 B \in O(A^3D) \qquad D^2\log C \in O(D^2A)$$

  In the case of $A^2\log^2 B \in O(A^3D)$, we only use one sub-product and one factor since $A^2\log^2 B \in O(A^3)$. In $B^2C^2 \in O(A^3D)$, we factorise B^2C^2 into $B^2C * C$ and we use the relations $B^2C \in O(A^3)$ and $C \in O(D)$.
- Using (R3) we deduce that

$$B^7C + A^2\log^2 B + B^2C^2 + D^2\log C \in O\left(2^BC^2 + A^3D + D^2A\right)$$

  because for every addend on the left there is one addend on the right that is asymptotically bigger (as shown in the previous step).

Thus, we conclude that $e_2 \in O(e_1)$. ■

The comparison procedure above is correct, however it is not complete. I.e., there exists cases in which $e_1 \in O(e_2)$ but the procedure is unable to prove it.

Example 3.7. For $e_1 \equiv A^2 + B^2$ and $e_2 \equiv AB$, it holds that $e_1 \in O(e_2)$, but our automatic comparison fails to prove it. ■

4 Asymptotic Cost Analysis

In this section we discuss how to build an asymptotic cost analyser that directly infers asymptotic bounds. A straightforward solution is to feed the output of an existing non-asymptotic cost analyser [75, 67, 58, 92, 132] to the input of the asymptotic transformation.

Example 3.8. If we apply the asymptotic transformation to the UBFs of Example 2.7, we get the following asymptotic bounds:

$$\mathit{inSort}(l_0) = \mathsf{nat}(l_0)^2 \qquad \mathit{insert}(i_0) = \mathsf{nat}(i_0)$$
$$\mathit{for}(l_0, i_0) = \mathsf{nat}(l_0 - i_0) * \mathsf{nat}(l_0) \qquad \mathit{while}_a(i_0) = \mathsf{nat}(i_0)$$

which are, precisely, what we would obtain by manually analysing the asymptotic cost of the methods and loops of the `inSort` program in Figure 2.2. ■

This solution is correct, but it is inefficient because the cost analyser generates detailed information that is thrown away by the asymptotic transformation. To avoid generating these details, we have developed an **asymptotic CRS**

solver that directly generates asymptotic bounds. The advantage of this approach is its efficiency, however, it can be applied only to cost analysers that are based on generating CRSs and then solving then into corresponding bounds.

Our solver follows the same phases as PUBS (see Section 2.5), however, it also applies the asymptotic transformation immediately after a cost expression is generated. In particular, we apply it at the following steps of PUBS:

- When transforming a CRS to direct recursive form, PUBS generates cost expressions that are collected from the unfolded equations. We apply the asymptotic transformation to such expressions.
- When transforming each CR in the CRS to a stand-alone form, PUBS replaces each external call by the bound computed for the called CR, and adds the bounds to the cost expression of the corresponding equation. We apply the asymptotic transformation to such expressions.
- When solving a standalone CR into a non-asymptotic bound, before storing this bound function we also apply the asymptotic transformation.

We do not change the way in which PUBS computes a ranking function, maximises cost expressions, and solves a standalone CR to bound functions.

Example 3.9. Let us apply the asymptotic resolution to solve the CRS of Figure 2.6. Transforming this CRS to direct recursion is done as in Example 2.6, except that when generating the third equation of $while_a$, the asymptotic transformation replaces the constant 2 with 1. The cost expressions of the other equations are already in asymptotic form. The next step is to solve the CRS, one CR at a time, similarly to what we have done in Example 2.7:

- The UBF of the CR $while_a(i_0)$ is $\mathsf{nat}(i_0)+1$. The asymptotic transformation reduces it to the AUBF ${while_a}^+(i_0) = \mathsf{nat}(i_0)$.
- Substituting ${while_a}^+(i_0)$ in the CR *insert*, results in a stand-alone CR with a single equation in which the cost expression is $1+\mathsf{nat}(i_0)$, which is then reduced by the asymptotic transformation to $\mathsf{nat}(i_0)$. Then, we get the UBF $insert^+(i_0) = \mathsf{nat}(i_0)$.
- Substituting $insert^+(i_0)$ in the CR *for*, results in a stand-alone CR in which the cost expression of the recursive equation is $\mathsf{nat}(i_0)$, which is already in asymptotic form. Solving this CR results in the UBF $\mathsf{nat}(l_0-i_0)*\mathsf{nat}(l_0-1)$, which is then reduced by the asymptotic transformation to $for^+(l_0,i_0) = \mathsf{nat}(l_0-i_0)*\mathsf{nat}(l_0)$.
- Substituting $for^+(l_0,i_0)$ in the CR *inSort*, and then solving it, results in the UBF $\mathsf{nat}(l_0-1)*\mathsf{nat}(l_0)$, which is then reduced by the asymptotic transformation to $inSort^+(l_0) = \mathsf{nat}(l_0)^2$.

Note that the UBFs computed above are as those of Example 3.8. ■

5 Experimental Evaluation

In this section we discuss an experimental evaluation of our techniques, which mainly compares our techniques to those of PUBS for inferring non-asymptotic bounds. Note that the existing techniques for transforming bounds to asymptotic form [128, 70] do not work with the kind of cost expressions used in COSTA.

Evaluation of the Asymptotic Transformation. We implemented our transformation as a back-end of COSTA. To experimentally evaluate it, we applied it to transform to asymptotic form the UBFs on memory consumption obtained by [13, §7]. For all benchmarks, the transformation obtained an accurate and minimal asymptotic form, and it took a negligible time. This shows how our transformation enables the transfer of existing work and software on computing non-asymptotic closed-form bounds into generating asymptotic closed-form bounds. This technique is applicable both to UBFs and LBFs, for any cost model and for any size measure.

Evaluation of the Asymptotic Resolution. In a second set of experiments, we studied the scalability of our approach, that is, how the size of the cost expressions and the time required to compute it increase when solving larger CRs. We have used the same set of benchmarks that were used in [5, §10.2] to study the scalability of PUBS. For each benchmark, we computed both a UBFs (using PUBS) and an AUBFs, and observed the following: (1) the time required to compute a UBF grows significantly with the size of the CRs, while the time to compute an AUBF remains small; (2) the size of each UBF is significantly larger than its corresponding AUBF; (3) the ratio between the size of the computed bounds and the number of equations grows faster for non-asymptotic UBFs than for asymptotic ones; and (4) for some of the biggest benchmarks, it was not possible to compute a non-asymptotic UBF in a reasonable time, but it was possible to compute the asymptotic form. These observations demonstrate that our approach is scalable.

6 Related Work

Asymptotic notations. Knuth [90] gave the formal definition of the asymptotic notations for univariate functions. Some authors [45, §3.1],[112, §9.2] use this definition because it is easy to translate to a logical formula from which to prove some basic properties. Others [119, §1.2], [62, §4.1.1], use another equivalent definition of O and Θ, based on the limit of the quotient $f(n)/g(n)$ when n tends to infinity. This is a more intuitive definition, as it better conveys the

idea that we study the proportionality between functions for arbitrarily large values of the input. Our work is centered on the big-O and big-Theta notations. However, it can be extended to other notationse.

A fundamental part of our work was to extend these notations to multivariate natural functions, and, at the same time, preserve the corresponding algebraic properties. We solved this issue by restricting the definition to functions that are monotonic on all input components as shown by Howell [79].

Asymptotic Transformation and Comparison. Stoutemyer [128] presents another method for the asymptotic transformation and comparison of functions. His procedure is built on the MACSYMA computer algebra system, and uses some of its advanced features. For instance, to compute limits of function quotients or Taylor series that cover a wide set of expressions. On the other hand, our method can handle context information (in the form of linear constraints). Our comparison method has some similarities with the method in PUBS for comparing non-asymptotic cost expressions [4].

Asymptotic Cost Analysis. Early works on static cost analysis [41, 43] distinguish between macro and micro analysis of programs, where the first one focuses on a dominant operation and the latter considers all operations. Under this view, an asymptotic bound falls into the category of macro analysis. In [102], an automatic asymptotic cost analysis for Horn clauses was presented, which has to take into account the sparsity of the relations in the logic program. The ACE system [97] performs an asymptotic cost analysis, which is based on program transformation, for programs in a first-order functional language. Complexity analyses have also been developed for other languages, such as simple loop programs [86], or term rewriting systems [103], and through the later for logic programming languages [60].

7 Further Reading

In this Chapter we have discussed the main contributions of [2], which are 1) the extension of the asymptotic notations to cost expressions; 2) an asymptotic transformation and comparison for cost expressions; and 3) an asymptotic CRS solver. We have illustrated the transformation and analysis with a simple example. The article contains the definitions of the asymptotic notations for cost expressions, and the definitions of the asymptotic transformation and the asymptotic comparison for cost expressions. It also provides correctness statements and their corresponding proofs.

4 | Nonlinear Operations

In this Chapter we describe a value analysis that is able to handle nonlinear integer arithmetic operations. When used within COSTA, it allows inferring precise asymptotic bound functions for programs on which COSTA failed or inferred imprecise ones before. This contribution has been published in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)

Overview

Our major goal in this dissertation is to improve the precision of COSTA, so that it computes precise bound functions for a wider class of programs. In this Chapter we focus on those components that abstract RBR instructions to linear constraints, infer postconditions for the different ACR procedures by means of linear constraints, and incorporate all these linear constraints in the corresponding CRS. Our goal is to improve the accuracy of such components so that they compute tighter approximations, or equivalently stronger constraints. The strength of these constraints has a direct impact on the precision of the inferred bounds: if they are too weak, then the CRS might include spurious evaluations that correspond to no program execution.

The constraints in the CRS approximate the values of the program variables at corresponding program points, and also define relations between the values of these variables. As we have seen in Chapter 2, these constraints come from either the abstract compilation or the value analysis. The value analysis of COSTA is based on the theory of Abstract Interpretation [48, 46]. This theory starts from the observation that the exact set of value relations is not computable. Instead, an abstract interpretation uses an abstract domain, which is a syntactic class of formulas [34, §12.1.4], that (typically) restricts the shape of the relations that can be used to either approximate the effect of executing a sequence of instructions, or to represent the inferred value relations.

Both the abstract compilation and the value analysis are based on the use of conjunctions of linear constraints which are enough, in practice, for efficiently

```
1   int clog(int x, int b){
2       int z = 0 ;
3       if (b > 1) {
4           int y = 1 ;
5           while ( y < x ){
6               y = y * b ;    ★
7               z = z + 1 ;
8           } ;
9       }
10      return z ;
11  }
```

Figure 4.1: Java code for the `clog` example. We mark the product with ★.

and precisely handling a wide class of programs. However, they are not enough for modeling the effect of nonlinear operations, like for instance the product $z = x * y$, whose semantics cannot be precisely modeled with a conjunction of linear constraints. This clearly affects the precision of the inferred bounds if the cost depends on such instructions.

The main problem we are dealing with, in this Chapter, is the imprecision in the abstract compilation and value analysis caused by such nonlinear arithmetic instructions. To solve this problem, we notice that although we cannot model these operations using a *single* conjunction of linear constraints, we can do so if we use a finite *disjunction* of such conjunctions by splitting the semantics of each nonlinear operation into several cases. Moreover, we can encode such disjunctions as auxiliary procedures in the ACR program, and thus, we can still use a scalable value analysis, and at the same time benefit from the disjunctive information to infer more precise value relations.

In Section 4.1 we explain the inner workings of our approach; In Section 4.2 we discuss an experimental evaluation of the techniques. in Section 4.3 we overview related work; and in Section 4.4 we describe some further details that can be found in the article.

1 Handling a Nonlinear Operation

The abstract compilation of COSTA abstracts nonlinear operations to *true*, the constraint that represents the space of all program states. This results in a significant loss of precision that leads to inferring asymptotically imprecise bounds, as demonstrated in the following example.

$clog(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $z := 0,$
  $if_c(\langle x, b, z\rangle, \langle r\rangle).$
$if_c(\langle x, b, z\rangle, \langle z\rangle) \leftarrow$
  $\{\neg b > 1\}.$
$if_c(\langle x, b, z\rangle, \langle z\rangle) \leftarrow$
  $\{b > 1\},$
  $y := 1,$
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle).$
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{\neg y < x\}.$
$while_c(\langle x, b, y, z\rangle, \langle z\rangle) \leftarrow$
  $\{y < x\},$
  $y := y * b,$ ★
  $z := z + 1,$
  $while_c(\langle x, b, y, z\rangle, \langle z\rangle).$

$clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $true,$
  $if_c(\langle x_0, b_0\rangle, \langle\rangle).$
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \leq 1.$
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \geq 2,$
  $y_0 = 1,$
  $while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle).$
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 \geq x_0.$
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 + 1 \leq x_0,$
  $true,$ ★
  $\mathsf{acquire}(1),$
  $while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle).$

Figure 4.2: The RBR (left) and the ACR (right) obtained by COSTA for the `clog` example. We mark the nonlinear instruction with ★.

Example 4.1. Figure 4.1 depicts a Java method `clog`, which takes as input two positive numbers `x` and `b`, and returns the value of $\lceil \log_b(x) \rceil$. We are interested in analysing the cost of this program with respect to a cost model that counts, for example, the number of visits to the loop body. Note that the precise asymptotic UBF in such case is in $\Theta(\log(x))$. The corresponding RBR and ACR programs, generated by COSTA, are depicted in Figure 4.2. Note that the RBR variables r and z were removed in the ACR because they do not affect the cost. In this example, the loss of crucial information happens in the abstract compilation step. In the Java method (resp. RBR program), the instruction `y = y * b` (resp. $y := y * b$) updates `y` (resp. y) to hold the value of `y * b` (resp. $y * b$). However, the constraint *true* in the ACR program means that variable y_1 may take any value, and this causes the cost of this ACR program to be unbounded. ■

In order to solve the above precision problem, and infer a precise UBF for method `clog`, we need to handle the product operation more precisely. That is, we want to improve the abstract compilation and value analysis to obtain a more precise value relation for that instruction. For this, we could use abstract domains that are able to track nonlinear relations [116, 28, 64]. However, operations in these domains are computationally expensive, which would render the value analysis impractical in most cases.

Our solution is based on the following observation: although the instruction

is not linear, it can be approximated by a linear relation under certain contexts. That is, if a linear constraint over the values of the operands (the context) holds, then an inequality over the result, or a relation between its input and output variables, holds as well.

Example 4.2. Consider the instruction $y := y * b$ in the RBR program of Figure 4.2. If we have no information about the values of y and b, then the constraint *true* is the only correct way of abstracting the semantics of $y := y * b$ with a conjunction of linear constraints. Instead, if the constraint $y \geq 1 \wedge b \geq 2$ is known to hold before the instruction, then we can use this context information to refine the abstraction of $y := y * b$ to $y_1 \geq 2 * y_0$, where y_0 and y_1 represent the values of y before and after the instruction, respectively. ■

The essence of our solution is to use context-sensitive information to refine the abstraction of nonlinear operations. Namely, we abstract each such instruction to a disjunction of cases, where each case specifies a possible scenario using a conjunction of linear inequalities. Note that for this abstraction to be correct, these cases must cover the whole input domain. The actual challenge here is how to represent these disjunctions and at the same time keep the value analysis practical.

An immediate solution would be to use disjunctive abstract domains, like powerset of polyhedra. However, these domains usually come with a performance overhead, which renders the analysis impractical in many cases. Moreover, we note that this disjunctive information is not required globally, but only locally when analysing the effect of nonlinear instructions. Thus, our solution, inspired by [115], is to encode disjunctions directly in the ACR without using disjunctive constraints, but rather taking advantage of the disjunctive nature of procedures in the ACR. For example, we replace the nonlinear arithmetic instruction $x := e_1 * e_2$ by a call $op_*(\langle e_1, e_2\rangle, \langle x\rangle)$, and define the auxiliary abstract procedure op_* by several rules that cover all possible input and simulate the corresponding disjunction. Importantly, each rule of op_* uses only a conjunction of linear constraints.

Example 4.3. Figure 4.3 depicts the ACR program obtained by applying the new abstract compilation to the RBR of method `clog`. This ACR is almost identical to the one in Figure 4.2, except that the instruction $y := y * b$ is now abstracted to a call $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$ which is defined in Figure 4.3 (at the bottom). The rules of op_* were carefully designed to partition its input domain, such that, the postcondition of each case propagates accurate information about constancy, equality and progression (e.g., multiplication by a constant). In particular, we distinguish the cases in which $x = 0$ (constancy), $x = \pm 1$ (equality) and those in which $|x| > 1$ and $|y| > 1$ (progress). Note that in case (g) the constraints $z \geq 2x$ and $z \geq 2y$ are crucial for finding a logarithmic UBF for our example. ■

$clog(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $if_c(\langle x_0, b_0\rangle, \langle\rangle)$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \leq 1$.
$if_c(\langle x_0, b_0\rangle, \langle\rangle) \leftarrow$
  $b_0 \geq 2$,
  $y_0 = 1$,
  $while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle)$.

$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 \geq x_0$.
$while_c(\langle x_0, b_0, y_0\rangle, \langle\rangle) \leftarrow$
  $y_0 + 1 \leq x_0$,
  $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle,$ ★
  acquire(1),
  $while_c(\langle x_0, b_0, y_1\rangle, \langle\rangle)$.

$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = 0$, $z = 0$. (a)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = 0$, $z = 0$. (b)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = 1$, $z = y$. (c)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = 1$, $z = x$. (d)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x = -1$, $z = -y$. (e)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $y = -1$, $z = -x$. (f)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \geq 2, y \geq 2$, $z \geq 2x, z \geq 2y$. (g)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \geq 2, y \leq -2$, $z \leq -2x, z \leq 2y$. (h)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \leq -2, y \geq 2$, $z \leq 2x, z \leq -2y$. (i)
$op_*(\langle x, y\rangle, \langle z\rangle) \leftarrow$
  $x \leq -2, y \leq -2$, $z \geq -2x, z \geq -2y$. (j)

Figure 4.3: The ACR program obtained by our modified abstract compilation for the `clog` program. The instruction $y := y * b$ is abstracted to $op_*(\langle y_0, b_0\rangle, \langle y_1\rangle)$. Below, we show the rules of the auxiliary procedure op_*.

After the abstract compilation, which abstracts each nonlinear operation to a call to an auxiliary procedure, the value analysis infers which of the cases of this procedure are enabled in the context of that call. This is done using an abstract interpretation to infer a precondition that describes how each ACR procedure is called. Technically, this is done using a fixpoint algorithm that computes the least fixed point (LFP) of a corresponding abstract collecting semantics. To keep the analysis efficient, we use the non-disjunctive abstract domain of polyhedra. Using this precondition, the value analysis can obtain a more precise postcondition for the nonlinear operation, as the least upper bound (lub) of the cases that are enabled. Once these pre and postconditions are computed, the CRS is generated as described in Section 2.4, except that preconditions are incorporated in the CRS as well. Then, the corresponding CRS

$$
\begin{array}{rll}
clog(x_0, b_0) = & if_c(x_0, b_0) & \{\} \\
if_c(x_0, b_0) = & 0 & \{b_0 \leq 1\} \\
if_c(x_0, b_0) = & while_c(x_0, b_0, y_0) & \{b_0 \geq 2, y_0 = 1\} \\
while_c(x_0, b_0, y_0) = & 0 & \{y_0 \geq x_0\} \\
while_c(x_0, b_0, y_0) = & 1 + while_c(x_0, b_0, y_1) & \{y_0{+}1 \leq x_0, \underbrace{b_0 \geq 2, y_0 \geq 1}_{\text{pre}-op_*}, \underbrace{y_1 \geq 2 * y_0}_{\text{post}-op_*}\}
\end{array}
$$

Figure 4.4: CRS inferred for the `clog` method, using our value analysis.

can be solved by PUBS to infer a UBF, just as done in Section 2.5.

Example 4.4. Consider the ACR program of Figure 4.3. For the ACR procedure $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$, the LFP algorithm infers the precondition $\varphi \equiv \{b_0 \geq 2, y_0 \geq 1\}$. For each rule of the op_* procedure, the algorithm computes the conjunction of φ with the constraints in the rule. A rule is enabled if this conjunction is not equal to *false*. Only the rules (d) and (g) are enabled, and their conjunctions are $\varphi_d \equiv \{y_0 \geq 2, b_0 \geq 1, y_1 = y_0\}$ and $\varphi_g \equiv \{y_0 \geq 2, b_0 \geq 2, y_1 \geq 2y_0, y_1 \geq 2b_0\}$, respectively. The algorithm computes the postcondition of $op_*(\langle b_0, y_0\rangle, \langle y_1\rangle)$ as the lub (convex hull) of φ_d and φ_g, which is $\varphi' \equiv \varphi \wedge \{y_1 \geq 2 * y_0\}$. From the ACR program, we now generate the CRS depicted in Figure 4.4. Note that the pre and postcondition of the auxiliary ACR procedure op_* are added to the second equation of the CR $while_c$. Then, PUBS is able to solve it to the UBF $clog^+(x_0, b_0) = \log_2(\mathsf{nat}(x_0) + 1)$ which is asymptotically precise. ■

Note that the original value analysis of COSTA follows a bottom-up strategy similar to that of [30], but for our method to handle nonlinear instructions, we need a top-down approach [40] in order to infer preconditions as well. Finally, note that in our abstraction we ignore the possibility of arithmetic overflow. Like other works in this field, we assume that overflow is an erroneous behaviour that should be handled independently in a previous step.

2 Experimental Evaluation

We implemented our value analysis in COSTA, for this (1) we implemented the top-down LFP algorithm that is described in the corresponding paper; and (2) modified the abstract compilation of nonlinear RBR operations so as to insert the auxiliary procedures in the ACR program. We use the *Parma Polyhedra Library* [24] for representing the corresponding abstract domains.

For our experiments, we used typical benchmarks that use nonlinear and bitvector operations. These benchmarks come from the literature on program analysis and from the Java standard libraries.

Each benchmark was analysed for termination and cost with respect to the *number of instructions* cost model, using COSTA with and without our extension. Without our extension COSTA failed on all benchmarks as expected, and with the extension it succeeded to prove termination and infer UBFs for all benchmarks. We observed that our value analysis is slower than the original one of COSTA, this is mainly due to the use of a top-down LFP algorithm, instead of the bottom-up one [10].

We also applied two other termination analysers for JBC, JULIA [125] and AProVE [61], on the same set of benchmarks. JULIA could not prove any of them terminating. AProVE could not prove termination of benchmarks that use bitvector operations, and proved termination of the rest. However, it required significantly more time (an order of magnitude larger).

These experiments confirm that, by using a value analysis that 1) abstracts nonlinear operations to disjunctions of linear cases, 2) encodes these disjunctions in the ACR programs as extra rule, 3) uses a non-disjunctive domain like polyhedra, and 4) follows a top-down LFP algorithm; we can greatly improve the precision of the value analysis of programs that use nonlinear operations.

3 Related Work

The value analysis of COSTA is based on Abstract Interpretation [48, 46], a theory for semantic-based program analysis, which allows systematic derivation of sound program analysers. This theory has been used to develop industrial analysers, such as ASTRÉE [49] and Julia [125].

Abstract Compilation. The notion of abstract compilation was first proposed in [72] as a syntactic transformation of programs into abstract programs, that can be then analysed for inferring a property of interest. Boucher [33] extends this notion to include optimisations on the abstract program.

Numerical Abstract Domains. Apart from the domain of polyhedra [68], the LFP algorithm of our method can also use any weakly relational abstract domain [25] to represent linear inequalities. The benefit of using them is that, by restricting the shape of the linear constraints, the abstract operations for these domains are computationally cheaper than those of polyhedra, so the LFP algorithm scales better. However, the disadvantage of doing this is that the results may not be precise enough. For instance, the inequality $y_1 \geq 2 * y_0$ from Example 4.4, which is crucial for inferring a precise AUBF, cannot be represented in some of these domains.

Some nonlinear instructions, like the product, could be represented using

an abstract domain of conjunctions of equalities [111] and inequalities [28] between multivariate polynomials of bounded degree. Also, we could use the method of [64] to lift an abstract domain of linear inequalities to constraints between logarithmic, exponential, radical and max expressions. However, these domains are computationally expensive.

The abstract domain of linear constraints between the absolute values of variables [37] could be used to abstract some nonlinear operations. For instance, we could abstract z := x mod y to $|z| < |y| \wedge |z| \leq |x|$.

Abstract Interpretation with Disjunctive Information A fundamental point in our technique is how to handle disjunctive information, a problem for which many solutions exist. On the side of abstract domains, one can use powerset domains [24] to represent finite disjunctions of elements. However, in [115] the authors argue that operations with these domains do not scale. Instead, they propose to handle disjunctive information via a program transformation, which they call an elaboration. The basic idea is to replace each node and its disjunctive invariant in the original CFG, by several nodes in the elaborated CFG, each with an invariant that corresponds to a disjunct of the original one. Trace partitioning [109] splits the set of all traces in the program semantics in several subsets, where each subset admits a non-disjunctive invariant.

4 Further Reading

In this Chapter we have briefly described how to handle nonlinear operations in the abstract compilation and the value analysis of COSTA, with the goal of inferring precise bounds for programs that use such operations. However, in the article the value analysis is presented towards the goal of proving termination by means of synthesising ranking functions.

In the article, we describe how to apply the method to handle not only the product z = x ∗ y, but also the nonlinear operations of integer quotient z = x/y, integer modulo z = x%y, and bitvector operations such as bitwise and z = x&y, bitwise or z = x|y, left shift z = x << y, and right shift z = x >> y. For inferring the pre and postcondition of each auxiliary procedure, the article also describes an interprocedural LFP algorithm that follows a top-down strategy [40].

5 | Amortised and Beyond

In this chapter we explore the limits of the classical approach to cost analysis which is used in COSTA, i.e., the approach that first abstracts the input program to a CRS and then solves this CRS into UBFs. It is known that this approach might infer UBFs that are asymptotically less precise than the actual cost. As yet, it was assumed that this imprecision is due to the way CRSs are solved into UBFs. We show that this assumption is partially true, and identify the reason due to which CRSs cannot precisely model the cost of some programs, independently from the precision of the underlying components. Then, to overcome these limitations, we develop a new approach to cost analysis, that is based on the use of satisfiability modulo theory (SMT for short) solvers and quantifier elimination (QE for short) procedures. Our approach is developed in a context in which, in addition to acquiring resources, programs can release resources as well. This gives rise to the notion of peak-cost. Our results have a strong relation to amortised cost analysis. This contribution has been published in the following article

> DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012

Overview

Our main goal in this dissertation is to improve the precision of COSTA in several directions. In this Chapter we focus on well-known precision issues of the classical approach to cost analysis, on which COSTA is based.

A core assumption in COSTA, and Wegbreit's classical approach [135] in general, is that the input of an ACR procedure determines both its output and its cost. This is reflected in the fact that each ACR procedure is abstracted into the following two *separate* pieces information: (1) a CR that models how the cost depends on the input; and (2) a value postcondition that approximates the relation between the input and the output of the ACR procedure. In Section 5.1 we show that this assumption is the source of some well-known precision issues of COSTA. In particular, we provide examples for which there is an implicit codependency between the output of an ACR procedure and its cost, which is

```
1   //@requires s >= 0 && m >= 0
2   void main(int s, int m) {
3       for( ; m > 0 ; m-- )
4           while( s > 0  && coin() )
5               s = s - 1 ;          ★
6   }
```

Figure 5.1: Java code for our guiding example.

crucial to infer precise UBFs. COSTA abstracts this output-cost codependency away when it transforms the ACR into the CRS, and thus introduces spurious evaluations that exist in the CRS but not in the ACR. This, in turn, leads to inferring asymptotically imprecise UBFs.

To solve this imprecision, we turn the implicit output-cost codependency into an explicit one by allowing UBFs to use the output parameters as well. We refer to such bounds as **net-cost** UBFs, which describe the cost of complete finite executions. In Section 5.2 we develop a novel technique for inferring such UBFs that is based on the use of QE and SMT solving. In Section 5.3 we consider another notion of cost, which we call the peak-cost, that estimates the quantity of resources that a program can hold simultaneously. This is useful for programs that can release resources and not only acquire them. Importantly, it also complements the net-cost bounds to handle non-terminating executions. We develop a novel technique for inferring peak-cost UBFs that is also based on the use of QE and SMT solving. In Section 5.4 we discuss an interesting relation between our approach and that of *automated amortised analysis* [81]. Briefly, we show that the potential functions in amortised analysis are just a restricted case of net-cost UBFs. In Section 5.5 we discuss an experimental evaluation of the techniques. In Section 5.6 we overview related work, and in Section 5.7 we describe some further details that can be found in the article.

1 The Output-Cost Codependency

Let us start our discussion with an example that illustrates the precision problem, of the classical approach to cost analysis, that we are dealing with in this chapter. The example is adapted from [131].

Example 5.1. Figure 5.1 includes a Java method in which the outer `for` loop performs `m` iterations, and in each iteration the inner `while` loop decrements variable `s` an arbitrary number of times – assuming that method `coin` non-deterministically returns `true` or `false`. We are interested in inferring a UBF

(a) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 = 0,$
$s_0 \geq 0.$

(b) $for(\langle s_0, m_0\rangle, \langle\rangle) \leftarrow$
$m_0 \geq 1,$
$s_0 \geq 0,$
$\underline{while(\langle s_0\rangle, \langle s_1\rangle)},$
$m_1 = m_0 - 1,$
$for(\langle s_1, m_1\rangle, \langle\rangle).$

(c) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 0,$
$s_1 = s_0.$

(d) $while(\langle s_0\rangle, \langle s_1\rangle) \leftarrow$
$s_0 \geq 1,$
acquire(1),
$s_2 = s_0 - 1,$
$while(\langle s_2\rangle, \langle s_1\rangle).$

Figure 5.2: ACR program for our guiding example.

(a)	$for(s_0, m_0) = 0$	$\{m{=}0, s_0 \geq 0\}$
(b)	$for(s_0, m_0) = \underline{while(s_0)} + for(s_1, m_1)$	$\{m_0 \geq 1, \underline{s_0 \geq s_1 \geq 0}, m_1 = m_0{-}1\}$
(c)	$while(s_0) = 0$	$\{s_0 \geq 0\}$
(d)	$while(s_0) = 1{+}while(s_2)$	$\{s_0 \geq 1, s_2 = s_0 - 1\}$

Figure 5.3: CRS for our guiding example.

on the number of visits to Line 5 (marked with ★). Note that the precise UBF is in $\Theta(s)$.

Using COSTA, we generate the ACR program depicted in Figure 5.2 and the CRS depicted in Figure 5.3 (for simplicity we skip the RBR). We omit procedure *main* since it only calls procedure *for*. In the ACR program, the output variable s_1 of *while* represents the value of `s` upon exit from the `while` loop. Note that when generating Equation (b) of the CRS, from Rule (b) of the ACR, the output parameter s_1 of the call to *while* is removed and a corresponding postcondition $s_0 \geq s_1 \geq 0$ is added to the equation's constraints. Applying PUBS on this CRS results in the UBFs $while^+(s_0) = s_0$, which is asymptotically precise, and $for^+(s_0, m_0) = s_0 * m_0$, which is asymptotically imprecise. Note that, for simplicity, in this Chapter we write UBFs without using the `nat` operator, however, we will guarantee that the expressions are always non-negative. ■

In order to overcome the precision problem of the above example, we first need to identify which component of COSTA is responsible for this imprecision. There are four possibilities: (1) the abstract compilation of the RBR into the ACR; (2) the value analysis that infers postconditions at the level of the ACR; (3) the transformation of the ACR into the CRS; (4) the resolution of the CRS into the final UBF. Next we rule out three of these four possibilities:

- The ACR in Figure 5.2 is a precise abstraction of the Java program of Figure 5.1. Namely, every ACR trace corresponds to an actual execution of

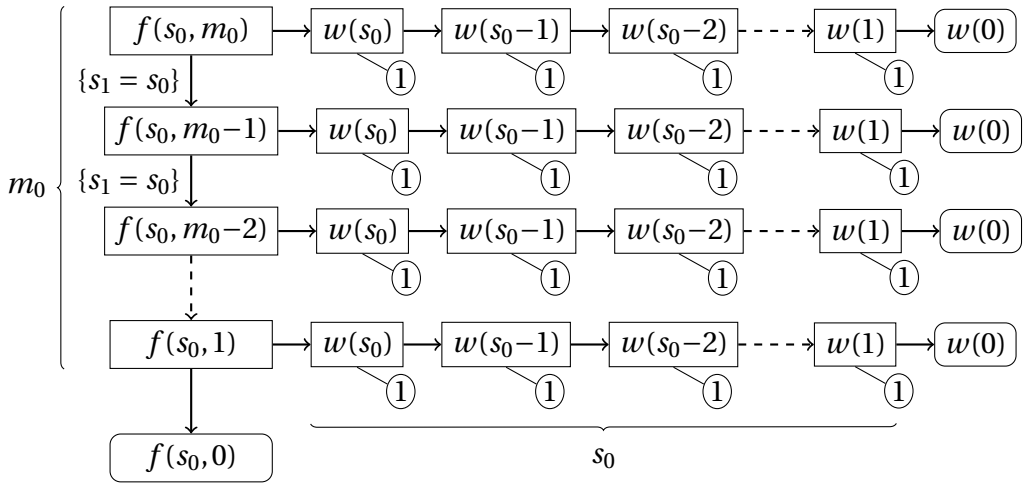

Figure 5.4: Evaluation tree of $for(s_0, m_0)$. We abbreviate *for*, *while* with f, w.

`main`. Thus, the abstract compilation is ruled out;

- The post condition $s_0 \geq s_1 \geq 0$ for procedure *while*, which is inferred by the value analysis, is the most precise input-output relation for the `while` loop. Thus, the value analysis is ruled out as well; and
- Examining the CRS of Figure 5.3, one can see that any call to $for(s_0, m_0)$ has a corresponding evaluation tree as the one in Figure 5.4, which has a total cost of $s_0 * m_0$. This means that $for^+(s_0, m_0) = s_0 * m_0$ is a precise UBF for this CRS, so the resolution process of PUBS is ruled out as well.

The only component that has not been ruled out above is the one that transforms the ACR to the CRS, and thus it must be the one responsible for the above imprecision. Specifically, the CRS contains some spurious evaluations that correspond to no ACR trace, as we show in the next example.

Example 5.2. Consider again the CRS of Figure 5.3 and the evaluation tree of Figure 5.4, which correspond to an initial call $for(s_0, m_0)$. The evaluation tree includes m_0 nodes that correspond to applications of Equation (b) – the vertical chain on the left side. Each node has two out-edges that correspond to calling *while* and recursively calling *for*. Note that when recursively calling *for*, we choose a value s_1 for its first parameter such that $s_1 = s_0$ (this is explicitly written on the edges). It is important to note that this choice is valid according to the constraints attached to Equation (b). Each call to $while(s_0)$ creates a chain of recursive calls, which result in a sub-tree with s_0 nodes (the horizontal chains). These nodes correspond to applications of Equation (d), so each one has a local cost 1 and thus the total cost of each such sub-tree is s_0. Since we have m_0 sub-trees, the total cost is $s_0 * m_0$.

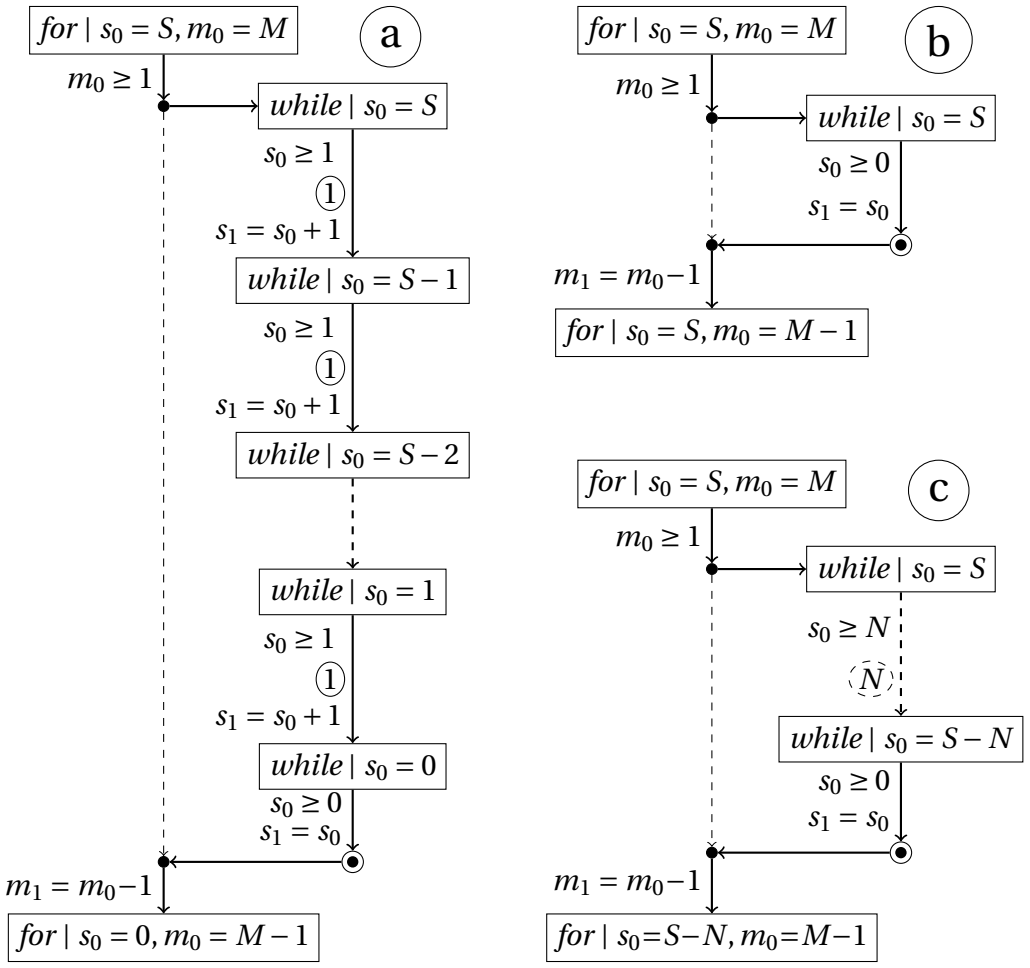

Figure 5.5: ACR trace fragments of a call to $for(\langle s_0\rangle, \langle m_0\rangle)$.

Next we show why this evaluation tree is spurious, and thus the cost $s_0 * m_0$ as well, by comparing it to possible traces of the corresponding ACR. For this it is enough to consider only the first application of Equation (b), which corresponds to the top-most horizontal chain in Figure 5.4. Note that before applying the recursive call $for(s_1, m_0 - 1)$, the call $while(s_0)$ has already accumulated s_0 units to the cost. Note also that in the recursive call we have chosen a value for the first parameter s_1 such that $s_1 = s_0$. Let us see why this behaviour is not possible at the level of the ACR. Figure 5.5 depicts three ACR traces that correspond to the first iteration of $for(\langle s_0, m_0\rangle, \langle\rangle)$, these traces exhibit different behaviours depending on the behaviour of the call $while(\langle s_0\rangle, \langle s_1\rangle)$ which is nondeterministic due to the use of method `coin`:

- Trace (a) corresponds to the case in which *while* makes s_0 iterations, so its

cost is s_0 and upon exit $s_1 = 0$.
- Trace (b) corresponds to the case in which *while* makes 0 iterations, so its cost is 0 and upon exit $s_1 = s_0$.
- Trace (c) corresponds to a general case in which *while* makes N iterations, and thus the cost is N and upon exit $s_1 = s_0 - N$.

The evaluation tree of Figure 5.4 is spurious because, according to the above ACR traces, it is not possible that the cost of $while(\langle s_0 \rangle, \langle s_1 \rangle)$ is s_0 and at the same time to have $s_1 = s_0$ upon exit. Indeed, this behaviour is a combination of the traces (a) and (b). ■

The above example exposes a (somehow) surprising dependency between the cost of the ACR procedure *while* and its output, which turns out to be crucial to infer a precise UBF for procedure *for*. However, in COSTA, this dependency is lost when transforming the ACR Rule (b) into the CR Equation (b), mainly because output parameters are removed. In fact, the CR of *while* precisely models the relation between the cost and the input; and, the postcondition, which is used when removing the output parameters, precisely models the relation between the input and the output values, but none of them models the codependency between cost and output. This codependency can be modelled using a novel form of UBF that expresses the cost of a procedure in terms of both its input and output parameters.

Example 5.3. For procedure $while(\langle s_0 \rangle, \langle s_1 \rangle)$, the UBF $while^+(s_0) = s_0$, is the most precise UBF that depends only on its input parameter. However, the UBF $while^+(s_0, s_1) = s_0 - s_1$, in which we also use the output parameter s_1, describes the exact cost of *while*. ■

Defining a UBF in terms of the output parameters may seem counterintuitive. This is because a UBF is usually used to *statically* estimate the cost of the program, which typically corresponds to the quantity of resources required for safely executing it. However, requiring information on the output parameters in order to evaluate the UBF is like requiring to execute the program. This is not the case. When inferring UBFs, we distinguish between the entry procedure (e.g., *for*), and the rest (e.g., *while*). The UBF for the entry can always be defined in terms of its input parameters only, but to infer a precise UBF for the entry we may need to infer, for each other procedure, a UBF in terms of its input and output parameters. Moreover, later we will introduce the notion of peak-cost UBF which depends only on the input parameters, however net-cost UBFs are used when inferring it.

Inferring UBFs that depend on both input and output parameters requires a new form of CRSs that involve both kinds of parameters, instead of only the

input ones. Due to the declarative nature of CRSs, a possible solution is to add the output parameters of each ACR procedure to the input parameters of its corresponding CR. However, PUBS would still fail to infer the desired UBF for our example, because its underlying methodology is based on multiplying the worst-case of all iterations of procedure *for* (which is s_0) by its number of iterations (which is m_0). Thus, in addition to enriching CRSs with output parameters, we also need a new resolution technique that is able to take into account the relation between costs of the different iterations of procedure *for*.

In the next sections we develop new techniques for computing UBFs, for several notions of cost, that are able to cope with the imprecision problems described above. Instead of using a new form of CRSs, we develop our techniques directly at the level of the ACR which, in principle, can be seen as a CRS with input and output parameters.

2 Inference of Net-Cost UBFs

Using the notation of Section 2.3, a complete finite execution trace t for a call $p(\bar{v},\bar{y})$ is one that starts in a state $\langle \bar{x}=\bar{v}, p(\bar{x},\bar{y})\rangle$ and ends in $\langle \psi,\epsilon\rangle$, where ϵ is an empty sequence of instructions. The net-cost of t is defined as the sum of the costs induced by its acquire(e_1) and release(e_2) instructions, it was denoted by *acrcost*(t) in Section 2.3. The restriction to complete and finite execution traces is due to the use of output variables when inferring UBFs on the net-cost.

For programs that do not use release(e), the meaning of the net-cost coincide with the standard notion of cost, and can be used to estimate the quantity of resources required to safely executing a program. However, if programs use release(e), then the usefulness of net-cost is not clear yet, since they cannot be used to estimate the quantity of resources required to safely executing a program. For example, the net-cost of a program that releases every resource that it acquires is 0. This will become clear in Section 5.3, for the rest of this section we may even assume that programs do not use release(e).

Our approach for inferring net-cost UBFs, that use both input and output parameters, is based on logical program analysis techniques. In essence, we view UBFs as program specifications, and then use the inductive assertion method [34, 83] to verify and synthesise those specifications. Our approach is thus developed in two steps:

1. **Verification**: In the first step, given an ACR program and a set of candidate net-cost UBFs, we develop a verification procedure to verify the validity of the given UBFs. This is done by deriving a **verification condition** (VC for short), which is a formula in First Order Logic (FOL for short), whose validity implies the validity of the candidate UBFs.

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0: & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow \widetilde{f}(s_0, m_0) \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_1, m_1: & \left(\begin{array}{l} m_0{\geq}1 \wedge s_0 \geq 0 \\ \wedge m_1 = m_0{-}1 \end{array}\right) & \rightarrow \widetilde{f}(s_0, m_0) \geq \widetilde{w}(s_0, s_1) + \widetilde{f}(s_1, m_1) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1: & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow \widetilde{w}(s_0, s_1) \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2: & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow \widetilde{w}(s_0, s_1) \geq 1 + \widetilde{w}(s_2, s_1)
\end{array}
$$

Figure 5.6: Net Cost VC for the ACR program of Figure 5.2

2. **Inference**: In a second step, we turn the verification procedure into an inference procedure that is able to synthesise net-cost UBFs, by using UBF **templates** and QE.

Next we explain these two steps by applying them to the example discussed in Section 5.1. The next example explains how to generate the VC.

Example 5.4. For the ACR program of Figure 5.2, we derive the VC $\widetilde{\Phi}$, which is a conjunction $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ of the clauses defined in Figure 5.6. The functions $\widetilde{w}(s_0, s_1)$ and $\widetilde{f}(s_0, m_0)$ refer to candidate net-cost UBFs for procedures *while* and *for*, respectively. As expected, $\widetilde{w}(s_0, s_1)$ uses both the input and the output parameters of the ACR procedure *while*.

Each clause $\widetilde{\phi}_i$ is generated from the ACR rule with label (i) in Figure 5.2, and has the form $\forall \bar{x} : \varphi \rightarrow f \geq g$. The inequality $f \geq g$ states that f, the net-cost of the corresponding procedure, is greater than (or equal to) the sum g of the net-cost induced by each instruction in the corresponding rule body. To derive the sum g, we let the net-cost of acquire(e) be e, that of release(e) be $-e$, that of a procedure call be its candidate net-cost UBF, and that of a constraint be 0. The clause states that $f \geq g$ must hold in a context φ, which is the conjunction of all constraints in the corresponding rule. The universal quantifier means that the inequality must hold for all valid valuations (of the program variables) that satisfy φ.

For instance, clause $\widetilde{\phi}_b$ states that for the function $\widetilde{f}(s_0, m_0)$ to be a valid UBF on the net-cost of procedure *for*, it has to be at least as the net-cost $\widetilde{w}(s_0, s_1)$ of the call to procedure *while*, plus the net-cost $\widetilde{f}(s_1, m_1)$ of the recursive call to procedure *for*. Moreover, this must hold for any values of s_0 and m_0 that satisfy the constraint $m_0 \geq 1 \wedge s_0 \geq 0 \wedge m_1 = m_0 - 1$ of Rule (b). ■

Given concrete definitions for the candidate net-cost UBFs, we can verify their validity by substituting them in the VC and then check its validity using, for example, an SMT solver.

Example 5.5. Let $\widetilde{f}(s_0, m_0){=}s_0$ and $\widetilde{w}(s_0, s_1){=}s_0{-}s_1$, which are the optimal UBFs on the net-cost of procedures *for* and *while*, respectively. Substituting these

functions in the VC of Figure 5.6 results in:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_1, m_1\colon & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_1 = m_0{-}1 & \rightarrow s_0 \geq s_0 - s_1 + s_1 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 - s_1 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 - s_1 \geq 1 + s_2 - s_1
\end{array}$$

Since the formula $\widetilde{\Phi} = \widetilde{\phi}_a \wedge \widetilde{\phi}_b \wedge \widetilde{\phi}_c \wedge \widetilde{\phi}_d$ is valid, then $\widetilde{f}(s_0, m_0)$ and $\widetilde{w}(s_0, s_1)$ are valid UBFs. On the other hand, if instead of $\widetilde{w}(s_0, s_1) = s_0 - s_1$ we use $\widetilde{w}(s_0, s_1) = s_0$, which is the most precise "input only" UBF for *while*, we get:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_b \equiv \forall s_0, m_0, s_1, m_1\colon & m_0{\geq}1 \wedge s_0 \geq 0 \wedge m_1 = m_0{-}1 & \rightarrow s_0 \geq s_0 + s_1 \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow s_0 \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow s_0 \geq 1 + s_2
\end{array}$$

Now $\widetilde{\phi}_c$ and $\widetilde{\phi}_d$ are valid, since $\widetilde{w}$ is a valid UBF on the net-cost of *while*. However, $\widetilde{\phi}_b$ is not valid even though $\widetilde{f}(s_0, m_0) = s_0$ is a valid UBF on the net-cost of procedure *for*. This is because the validity of $\widetilde{f}$ cannot be proven using the "input only" UBF $\widetilde{w}(s_0, s_1) = s_0$. ■

Rather than verifying that some given candidates are valid, in automatic cost analysis the main interest is to directly synthesise those UBFs. This can be formulated as seeking net-cost UBFs $\{\widetilde{f}_1, \ldots, \widetilde{f}_k\}$ for which the corresponding VC is valid. However, this is a second order logical problem and solving it is impractical in general. A common approach to avoid solving a second order problem is to use template functions which restrict the form of functions that we are looking for [126, 129, 83, 130, 117]. A **template** is a function with a predefined structure, defined over the procedure parameters (as in the UBFs above), but contains as well some unknown template parameters. The use of such templates reduces the second order problem to that of seeking values for the unknown template parameters, which is a FOL problem.

Example 5.6. Consider the following UBF templates on the net-cost of procedures *for* and *while*:

$$\widetilde{f}(s_0, m_0) = f_\mathrm{s} s_0 + f_\mathrm{m} m_0 + f_\mathrm{c} \qquad \widetilde{w}(s_0, s_1) = w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c}$$

Note that s_0, m_0, s_1 are the parameters of the UBFs, which correspond to ACR variables, and $\{f_\mathrm{s}, f_\mathrm{m}, f_\mathrm{c}, w_\mathrm{s}, w_\mathrm{t}, w_\mathrm{c}\}$ are the template parameters. Substituting

these templates in the VC of Figure 5.6 results in the following FOL formulas:

$$\begin{array}{lll}
\widetilde{\phi}_a \equiv \forall s_0, m_0\colon & m_0{=}0 \wedge s_0 \geq 0 & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 + f_\mathrm{c} \geq 0 \\
\widetilde{\phi}_b \equiv \forall \left\{ \begin{matrix} s_0, m_0, \\ s_1, m_1 \end{matrix} \right\}\colon & \left(\begin{matrix} m_0 \geq 1 \wedge s_0 \geq 0 \\ \wedge m_1 = m_0 - 1 \end{matrix} \right) & \rightarrow f_\mathrm{s} s_0 + f_\mathrm{m} m_0 \geq \left(\begin{matrix} w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \\ + f_\mathrm{s} s_1 + f_\mathrm{m} m_1 \end{matrix} \right) \\
\widetilde{\phi}_c \equiv \forall s_0, s_1\colon & s_0 \geq 0 \wedge s_1 = s_0 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 0 \\
\widetilde{\phi}_d \equiv \forall s_0, s_1, s_2\colon & s_0 \geq 1 \wedge s_2 = s_0 - 1 & \rightarrow w_\mathrm{s} s_0 + w_\mathrm{t} s_1 + w_\mathrm{c} \geq 1 + w_\mathrm{s} s_2 + w_\mathrm{t} s_1 + w_\mathrm{c}
\end{array}$$

In these formulas, the ACR variables are universally quantified but the template parameters are free.

Solving $\widetilde{\Phi}$ in this context means finding some values for $\{f_\mathrm{s}, f_\mathrm{m}, f_\mathrm{c}, w_\mathrm{s}, w_\mathrm{t}, w_\mathrm{c}\}$ for which the formula is valid. In turn, these values define instances of the templates that are valid UBFs. To solve $\widetilde{\Phi}$, we apply a QE procedure [83, §9], which eliminates the universally quantified variables (the ACR variables) and leaves an equivalent quantifier-free formula over the template parameters, namely $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \widetilde{\psi}_b \wedge \widetilde{\psi}_c \wedge \widetilde{\psi}_d$, where:

$$\begin{array}{ll}
\widetilde{\psi}_a \equiv f_\mathrm{s} \geq 0 \wedge f_\mathrm{c} \geq 0 & \widetilde{\psi}_c \equiv\ w_\mathrm{s} + w_\mathrm{t} \geq 0 \wedge w_\mathrm{c} \geq 0 \\
\widetilde{\psi}_b \equiv f_\mathrm{s} \geq w_\mathrm{s} \wedge f_\mathrm{m} \geq w_\mathrm{c} \wedge f_\mathrm{s} + w_\mathrm{t} = 0 & \widetilde{\psi}_d \equiv w_\mathrm{s} \geq 1
\end{array}$$

The formulas $\widetilde{\Phi}$ and $\widetilde{\Psi}$ are equivalent, so any solution for $\widetilde{\Psi}$ is a solution for $\widetilde{\Phi}$. For instance, $\{f_\mathrm{s} = 1, f_\mathrm{m} = 0, f_\mathrm{c} = 0, w_\mathrm{s} = 1, w_\mathrm{t} = -1, w_\mathrm{c} = 0\}$ is a solution of $\widetilde{\Psi}$, which gives the net-cost UBFs $\widetilde{f}(s_0, m_0) = s_0$ and $\widetilde{w}(s_0, s_1) = s_0 - s_1$. ■

Inference of Lower Bounds. It is easy to adapt our approach to compute LBFs on the net-cost, by turning all $\geq$ to $\leq$ in the VC of Figure 5.6.

Net-cost and Non-Terminating Programs. Since the net-cost is defined for *finite* executions, the approach described above is not adequate for inferring the cost of non-terminating programs. This problem will be addressed and solved at the end of the next section.

3 Inference of Peak-Cost UBFs

The consumption of a resource that is only acquired, like the number of instructions, can be analysed using an **accumulative** cost model, that is, one that only introduces acquire(e) instructions in the ACR. However, some kinds of resources can be acquired and released during the execution, for instance, heap memory in the presence of a garbage collector (GC) [13]. Consumption of such resources is analysed using a **non-accumulative** cost model, which introduces in the ACR program instructions for both acquiring and releasing resources. To

```
1   //@requires n>=1
2   void p(int n) {
3     if (n > 1) {
4       int m = q(n);
5       // gc A
6       p(n-m);
7       // gc B
8     }
9   }
```

```
10  //@requires n>=2
11  int q(int n) {
12    int i = n/2;
13    do {
14      A x = new A();
15      B y = new B();
16      i--;
17    } while(i>0 && coin());
18    return n/2 - i;
19  }
```

Figure 5.7: Java code for the example on Peak heap space.

allow such cost models, we extend the ACR with an instruction release(e), which releases e resources, and the semantics with this rule:

$$\frac{eval(e,\psi) = v \geq 0}{\langle \psi, \mathsf{release}(e) \cdot \bar{a} \rangle \overset{-v}{\rightsquigarrow} \langle \psi, \bar{a} \rangle}$$

which annotates the transition with a negative value $-v$. Let us see an example of the usage of this ACR instructions.

Example 5.7. Consider the Java program depicted in Figure 5.7. Method `q` receives an integer `n`, executes a loop at least 1 and at most `n/2` iterations, and returns the number of iterations that it has performed. In each iteration it creates one object of class `A` and one of class `B`. Method `p` implements a loop (using recursion) where in each iteration it calls `q` with the current value of `n`, and then performs a recursive call with the loop counter decremented by `m` (the number of iterations that `q` has performed). We are interested in estimating the amount of heap space (in terms of the number of objects) required to run the program without running out of memory, assuming that the GC frees all instances of class `A` after the call to `q` (Line 5), and all instances of class `B` after the recursive call (Line 7). The corresponding ACR program is depicted in Figure 5.8. Note the use of release(m_2) to model the effect of applying the GC. ■

If a program is analysed with an accumulative cost model, then a net-cost UBF is enough to safely estimate how many resources are required to execute the program. However, if the program is analysed with a non-accumulative cost model, or if its semantics contains non-terminating executions, we then need to estimate its **peak-cost**, which is the maximum quantity of resources held (i.e., acquired and not yet released) at any point of the execution. The

(a) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 = 1.$

(b) $p(\langle n_0\rangle, \langle\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad q(\langle n_0\rangle, \langle m_2\rangle),$
$\quad n_2 = n_0 - m_2,$
$\quad p(\langle n_2\rangle, \langle\rangle),$
$\quad \mathsf{release}(m_2).$

(c) $q(\langle n_0\rangle, \langle m_1\rangle) \leftarrow$
$\quad n_0 \geq 2,$
$\quad i_2 = n_0/2 - 1,$
$\quad \mathsf{acquire}(2),$
$\quad l(\langle i_2\rangle, \langle i_3\rangle),$
$\quad m_1 = n_0/2 - i_3,$
$\quad \mathsf{release}(m_1).$

(d) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 0,$
$\quad i_1 = i_0.$

(e) $l(\langle i_0\rangle, \langle i_1\rangle) \leftarrow$
$\quad i_0 \geq 1,$
$\quad \mathsf{acquire}(2),$
$\quad i_2 = i_0 - 1,$
$\quad l(\langle i_2\rangle, \langle i_1\rangle).$

Figure 5.8: ACR program for the Java program of Figure 5.7

peak cost of a trace t is defined as $\max\{\, \mathit{acrcost}(t') \mid t' \text{ is a prefix of } t\}$. A function p^+ is a UBF for the peak-cost of procedure p, if for any input $\bar{v}$, for any trace t that starts in $\langle \bar{x} = \bar{v}, p(\bar{x}, \bar{y})\rangle$, and for any prefix t' of t, we have that $p^+(\bar{v}) \geq \mathit{acrcost}(t')$. Note that this definition covers both terminating and non-terminating executions.

Example 5.8. Consider the ACR program of Figure 5.8. Any execution of `p(n)` creates $2n$ objects, but all of them are released before the execution ends, so the net-cost of this program is 0 (this is what the analysis of Section 5.2 infers). However, due to the GC, the program can hold up to n objects at any moment. Thus, the function $\widehat{p}(n) = n$ is a UBF on the peak-cost of `p`. ■

In order to handle non-accumulative resources, we complement the UBF on the net-cost with a UBF on the peak-cost. A UBF on the peak-cost of an ACR procedure is a function over its input parameters such that, for any concrete input, any trace has a peak-cost below the UBF. Note that peak-cost UBFs are defined in terms of the input parameters only. This is because inferring them will require examining partial executions, and, moreover, they will also handle non-terminating executions, for which it is not possible to express the cost in terms of the output. However, we will use the net-cost UBFs, which are defined in terms of both input and output parameters, in order to infer peak-cost UBFs.

As in the case of the net-cost, our approach for inferring peak-cost UBFs is developed in two steps: given an ACR program and a candidate set of peak-cost UBFs, we first derive a VC whose validity implies the validity of the candidate UBFs; then we use UBF templates and QE to turn this verification procedure to an inference procedure.

Let us first explain the intuition behind the peak-cost VC. Given an ACR rule $p(\langle x\rangle, \langle y\rangle) \leftarrow q_1(\langle x\rangle, \langle w\rangle), q_2(\langle w\rangle, \langle y\rangle)$, when executing p, its peak-cost can be reached either *a*) during the execution of q_1; or *b*) during the execution of q_2. Case a) imposes the condition that the peak-cost of p is at least as the peak-cost of q_1, and Case b) imposes the condition that the peak-cost of p is at least as the

$$
\begin{array}{lll}
\widehat{\phi}_a \equiv\ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widehat{p}(n_0) \geq 0 \\
\widehat{\phi}_b \equiv\ \forall \bar{w}_b : (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq 0 \quad) \\
\qquad \wedge\ (n_0 \geq 2 & \rightarrow & \widehat{p}(n_0) \geq \widehat{q}(n_0) \quad) \\
\qquad \wedge\ (n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widehat{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widehat{p}(n_2) \quad) \\
\widehat{\phi}_c \equiv\ \forall \bar{w}_c : (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 \quad) \\
\qquad \wedge\ (n_0 \geq 2 \wedge n_0 = 2i_2 + 2 & \rightarrow & \widehat{q}(n_0) \geq 2 + \widehat{l}(i_2) \quad) \\
\widehat{\phi}_d \equiv\ \forall \bar{w}_d : i_0 \geq 0 & \rightarrow & \widehat{l}(i_0) \geq 0 \\
\widehat{\phi}_e \equiv\ \forall \bar{w}_e : (i_0 \geq 1 & \rightarrow & \widehat{l}(i_0) \geq 2 \quad) \\
\qquad \wedge\ (i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widehat{l}(i_0) \geq 2 + \widehat{l}(i_2) \quad)
\end{array}
$$

$$
\begin{array}{lll}
\widetilde{\phi}_a \equiv\ \forall \bar{w}_a : n_0 = 1 & \rightarrow & \widetilde{p}(n_0) \geq 0 \\
\widetilde{\phi}_b \equiv\ \forall \bar{w}_b : n_0 \geq 2 \wedge n_2 = n_0 - m_2 & \rightarrow & \widetilde{p}(n_0) \geq \widetilde{q}(n_0, m_2) + \widetilde{p}(n_2) - m_2 \\
\widetilde{\phi}_c \equiv\ \forall \bar{w}_c : \left(\begin{array}{l} n_0 \geq 2 \wedge n_0 = 2i_2 + 2 \\ 2m_1 = n_0 - 2i_3 \end{array} \right) & \rightarrow & \widetilde{q}(n_0, m_1) \geq 2 + \widetilde{l}(i_2, i_3) - m_1 \\
\widetilde{\phi}_d \equiv\ \forall \bar{w}_d : i_0 \geq 0 \wedge i_1 = i_0 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 0 \\
\widetilde{\phi}_e \equiv\ \forall \bar{w}_e : i_0 \geq 1 \wedge i_2 = i_0 - 1 & \rightarrow & \widetilde{l}(i_0, i_1) \geq 2 + \widetilde{l}(i_2, i_1)
\end{array}
$$

Figure 5.9: Net Cost and Peak Cost VC for the ACR program in Figure 5.8

quantity of resources it holds upon exit from q_1, which coincides with the net-cost of q_1, plus the peak-cost of q_2. In addition, we require that the peak-cost is non-negative. Note the use of the net-cost in the second case. Let us now apply this intuition to our example.

Example 5.9. Using the ACR program of Figure 5.8 we derive the VC $\widehat{\Phi} = \widehat{\phi}_a \wedge \cdots \wedge \widehat{\phi}_e \wedge \widetilde{\phi}_a \wedge \cdots \wedge \widetilde{\phi}_e$, where $\widetilde{\phi}_i$ are clauses that correspond to the net-cost VC (derived as in Section 5.2) and $\widehat{\phi}_i$ are clauses for the peak-cost. As notation, $\bar{w}_i$ stands for the variables occurring at the formula indexed by i.

As explained above, the net-cost UBFs are required for formulating the peak-cost conditions. All clauses are depicted in Figure 5.9. Clause $\widehat{\phi}_i$ is generated from the ACR rule with label (i) in Figure 5.8, following the intuition described above. For example, clause $\widehat{\phi}_b$ includes three conjuncts: the first one states that the peak-cost of p is positive, the second one states that it is at least as that of q; and the third one states that it is at least the peak-cost of the recursive call to p plus the quantity of resources held before that call. This last quantity is exactly the net-cost $\widetilde{q}(n_0, m_2)$ of q. ■

To verify a candidate set of peak-cost UBFs, we have to substitute them in the VC and check the validity modulo theory of the VC. However, our interest is in inferring these UBFs rather than checking the validity of the given ones. This can be done using UBF templates and QE, as in the case of net-cost.

$$
\begin{array}{ll}
\widehat{\psi}_a \equiv & \widehat{p}_n + \widehat{p}_c \geq 0 \\
\widehat{\psi}_b \equiv & (\widehat{p}_n \geq 0 \wedge 2\widehat{p}_n + \widehat{p}_c \geq 0) \ \wedge \ (\widehat{p}_n \geq \widehat{q}_n \wedge 2\widehat{p}_n + \widehat{p}_c \geq 2\widehat{q}_n + \widehat{q}_c) \\
& \wedge \ (\widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \wedge \widehat{p}_n = \widetilde{q}_m) \\
\widehat{\psi}_c \equiv & (\widehat{q}_n \geq 0 \wedge 2\widehat{q}_n + \widehat{q}_c \geq 2) \ \wedge \ (2\widehat{q}_n \geq \widehat{l}_i \wedge 2\widehat{q}_n + \widehat{q}_c \geq \widehat{l}_c + 2) \\
\widehat{\psi}_d \equiv & \widehat{l}_i \geq 0 \wedge \widehat{l}_c \geq 0 \\
\widehat{\psi}_e \equiv & (\widehat{l}_i \geq 0 \wedge \widehat{l}_i + \widehat{l}_c \geq 2) \ \wedge \ (\widehat{l}_i \geq 2)
\end{array}
$$

$$
\begin{array}{ll}
\widetilde{\psi}_a \equiv & \widetilde{p}_c + \widetilde{p}_n \geq 0 \\
\widetilde{\psi}_b \equiv & \widetilde{p}_n = \widetilde{q}_m - 1 \wedge \widetilde{q}_n \leq 0 \wedge 2\widetilde{q}_n + \widetilde{q}_c \leq 0 \\
\widetilde{\psi}_c \equiv & 2\widetilde{q}_n \geq \widetilde{l}_i + \widetilde{l}_j \wedge \widetilde{q}_m + \widetilde{l}_j = -1 \wedge 2\widetilde{q}_n + \widetilde{q}_c \geq \widetilde{l}_j + \widetilde{l}_c + 2 \\
\widetilde{\psi}_d \equiv & \widetilde{l}_i + \widetilde{l}_j \geq 0 \wedge \widetilde{l}_c \geq 0 \\
\widetilde{\psi}_e \equiv & \widetilde{l}_i \geq 2
\end{array}
$$

Figure 5.10: Quantifier-Free clauses, obtained by substituting the template UBFs of Example 5.10 in the VC of Figure 5.9, and then applying real QE.

Example 5.10. Consider the following net- and peak-cost UBF templates for the ACR program of Figure 5.8:

$$
\begin{array}{lll}
\widehat{p}(n_0) = \widehat{p}_n n_0 + \widehat{p}_c & \widehat{q}(n_0) = \widehat{q}_n n_0 + \widehat{q}_c & \widetilde{q}(n_0, m_1) = \widetilde{q}_n n_0 + \widetilde{q}_m m_1 + \widetilde{q}_c \\
\widetilde{p}(n_0) = \widetilde{p}_n n_0 + \widetilde{p}_c & \widehat{l}(i_0) = \widehat{l}_i i_0 + \widehat{l}_c & \widetilde{l}(i_0, i_1) = \widetilde{l}_i i_0 + \widetilde{l}_j i_1 + \widetilde{l}_c
\end{array}
$$

Substituting these templates in the VC of Figure 5.9, and then eliminating the universal quantifiers results in the quantifier free formulas $\widehat{\Psi} = \widehat{\psi}_a \wedge \cdots \wedge \widehat{\psi}_e$ and $\widetilde{\Psi} = \widetilde{\psi}_a \wedge \cdots \wedge \widetilde{\psi}_e$ where $\widetilde{\psi}_i$ and $\widehat{\psi}_i$ are depicted in Figure 5.10. Each clause $\widehat{\psi}_i$ (resp. $\widetilde{\psi}_i$) is obtained by eliminating the universal quantifier from the clause $\widehat{\phi}_i$ (resp. $\widetilde{\phi}_i$). Every solution for $\widehat{\Psi} \wedge \widetilde{\Psi}$ defines instances of the templates that are valid UBFs. In particular, the following solution:

$$
\begin{array}{lllllllll}
\widehat{p}_n = 1 & \widehat{p}_c = 0 & & \widehat{q}_n = 1 & \widehat{q}_c = 0 & & \widetilde{q}_n = 0 & \widetilde{q}_m = 1 & \widetilde{q}_c = 0 \\
\widetilde{p}_n = 0 & \widetilde{p}_c = 0 & & \widehat{l}_i = 2 & \widehat{l}_c = 0 & & \widetilde{l}_i = 2 & \widetilde{l}_j = -2 & \widetilde{l}_c = 0
\end{array}
$$

defines the following UBFs on the peak and net-cost:

$$
\begin{array}{lll}
\widehat{p}(n_0) = n_0 & \widehat{q}(n_0) = n_0 & \widetilde{q}(n_0, m_1) = m_1 \\
\widetilde{p}(n_0) = 0 & \widehat{l}(i_0) = 2i_0 & \widetilde{l}(i_0, i_1) = 2i_0 - 2i_1
\end{array}
$$

Note that for procedure q we use a net-cost UBF that only depends on the output parameter m_1 of q. ■

Let us finish this section by commenting on the case of non-terminating programs.

Example 5.11. Consider a non-terminating ACR program defined by the rule

$$p(\langle n_0\rangle, \langle\rangle) \leftarrow n_0 \geq m \geq 0, \mathsf{acquire}(m), n_2 = n_0 - m, p(\langle n_2\rangle, \langle\rangle).$$

Procedure p receives a non-negative integer n_0, nondeterministically chooses a non-negative value $m \leq n_0$, acquires m resources, and then calls p recursively with $n_0 - m$. The peak-cost of this program is exactly n_0, since no trace can acquire more than n_0 resources, and there are some traces that acquire exactly n_0. This peak-cost UBF can be inferred using the approach described in this section. For more details, see Example 9 of the article. ■

4 Relation to Amortised Cost Analysis

Let us discuss now an interesting relation we observed between UBFs that are defined in terms of both input and output parameters, and the notion of potential functions used in amortised cost analysis. This offers a semantics-based explanation to why amortised analysis can obtain more precise UBFs.

In the context of an ACR program, a potential function maps a given state to a non-negative number, which is called the potential of the state. This potential can be interpreted as the quantity of resources available in the given state. An automatic amortised cost analysis [81, 75] assigns to each ACR procedure $p(\bar{x}, \bar{y})$ two potential functions: one for the input $P_p(\bar{x})$, and one for the output $Q_p(\bar{y})$. Intuitively, the input potential $P_p(\bar{x})$ must be large enough to pay for the cost of executing $p(\bar{x}, \bar{y})$, and, upon exit, leave at least $Q_p(\bar{y})$ units to pay the cost of the rest of the execution. Thus, if c is the net-cost of p, then $P_p(\bar{x}) \geq c + Q_p(\bar{y})$ must hold. This can be rewritten as $P_p(\bar{x}) - Q_p(\bar{y}) \geq c$, so the difference $P_p(\bar{x}) - Q_p(\bar{y})$ is a UBF on the net-cost of p, defined both over the input and output of p.

The above potential functions are in principle UBFs as described in Section 5.2, however, they are just a special case. We have observed some cases in which the net-cost UBF cannot be expressed as the difference of two functions as above, i.e., one over the input parameters and one over the output parameters, but rather it includes a monomial that combines input and output parameters. For more related details, see Section 6 in the article. It is worth noting, in addition, that net-cost UBFs are not limited to polynomial templates as the case of the potential functions used in [75].

5 Experimental Evaluation

Implementation. We have developed for our method a prototype implementation, called ACRP. This program takes less than 1500 lines of HASKELL code.

ACRP takes as input a text file that, in a Prolog-like syntax, contains an ACR program and a template UBF on the net and peak cost of each procedure. ACRP generates the VC for the net and peak cost of the ACR program. Then, ACRP invokes the QE methods provided by REDLOG [57] (an extension of the REDUCE computer algebra system [69]) to generate the constraints over the parameters of the template UBFs. ACRP uses the FOL theory of *real closed fields*, which allows us to use a wide range of template UBFs, such as multivariate polynomials and max and min operations. ACRP includes also an option to use the combination of REDLOG, QEPCAD and SLFQ of [129], to reduce the size of the constraints. ACRP outputs the constraint as a SMTLIB2 script [29], using the logic of *nonlinear real arithmetic* (`QF_NRA`). This script can then be passed to an SMT solver like Z3 [53], to obtain a solution of the parameters.

Precision Experiments. We have applied the analyser on small ACR programs that we wrote from examples in the literature. Some of these examples are the methods to manipulate a binary counter [45, §17], or some nested loops analysed in the articles of the SPEED project [66, 67, 65], For these examples we obtained the expected precise UBFs. In many of them, there was one procedure for which it was necessary to use a UBF on the net cost that depended on the output parameters. This indicates the presence of the output-cost codependency that we mentioned in many of these programs, and the effectiveness of our approach to cost analysis. However, we also found that some programs usually described in terms of amortised cost, like the methods for a dynamic array [45, §17] or for a queue implemented with two stacks [104], could be solved *without* using UBFs that depended on the output parameters. This indicated us that the output-cost codependency is not important for such programs, and that these could in principle be solved with precision using a more precise CR solver. This was an inspiration of our contribution described in Chapter 6.

Scalability Limitations. Unfortunately, due to the high computational costs of QE procedures for the theory of real closed fields, ACRP can only be applied to small examples and it does not scale for large programs. This limits the practicality of our approach, however, it does not degrade its theoretical importance. Besides, our approach is applicable to any set of template UBFs, so long as they are supported by the QE procedure. In this sense, the choice of a QE procedure and a FOL theory determines the scope and the scalability of the technique. In particular, in Chapter 6 we overcome these limitations by restricting ourselves to scalable QE procedures for linear real arithmetic.

6 Related Work

Automated Amortised Analysis. In Section 5.4, we have already commented on the connection between our approach and that of automated amortised analysis. This approach was introduced in the doctoral dissertation of Jost [81], and has later been extended in several directions [35, 82, 124, 78, 75].

Inductive Assertion Method. Our approach for inferring UBFs on the net- and peak-cost is based on viewing these UBFs as program specifications, and then use the inductive assertion method to verify and synthesise those specifications. This idea was already proposed by Wegbreit [136] for verifying average-cost estimates. QE over the real field has been previously used for inferring UBFs [19], and for verifying UBFs [52].

Simple Loops. The works of the SPEED project [65, 67, 141, 66], present analysis techniques for imperative programs that are able to handle a particular form of the output-cost codependency, which appears in some nested loops in which the control variable of an outer loop is modified inside an inner loop. Unlike ours, their approach cannot handle multiple recursion.

7 Further Reading

In this Chapter we have described the technical contributions of [18], which studies the limitations of the classical approach to cost analysis on which COSTA is based. We explained why the source of these limitations is related to the fact that CRSs ignore the output parameters of ACR procedures, and showed that output-cost codependency is crucial for inferring precise UBFs. In order to overcome these limitations, we introduced the notion of net-cost UBFs, which are UBFs defined in terms of both input and output parameters. We have also developed a novel technique for inferring such UBFs, and extended them to infer UBFs on the peak-cost of programs. We have observed a strong relation to amortised cost analysis [75], which provides an alternative semantic-based explanation to why amortised analysis (of ACR programs) can be more precise than the classical approach.

The article extends and formalises the contents of this chapter. It includes a formal syntax and semantics for the ACR language, as well as the definitions of net-cost and peak-cost. It also describes the derivation of the different VCs from the ACR program, and provides correctness statements together with their corresponding proofs. The article also contains the tables with the results of our experiments.

6 | Logical Resolution of CRS

In this chapter we develop a precise and scalable technique for solving CRSs into UBFs. This is done by exploring the gap between the scalable technique of PUBS, and the precise techniques of Chapter 5. Our technique first splits the input CRS into several atomic ones, uses precise local reasoning to infer the corresponding UBFs, and then combines these UBFs into a UBF for the original CRS. For the local reasoning we propose several methods that define the cost as a solution of a universally quantified FOL formula, similarly to what has been done in Chapter 5. We also rely on techniques used in PUBS. This contribution has been published in the following article:

> DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013

Overview

The techniques used in PUBS [5] to compute UBFs are based on assuming worst-case behaviour for all equations, both for the cost contributed by each equation and for the number of times each equation is applied. This approach is efficient and can handle a wide class of CRSs. However, it might lead to undesired imprecision, which might be even unrelated to output-cost codependency of Chapter 5. This imprecision becomes even more significant for CRSs that originate from divide and conquer algorithms. In such CRSs, the number of times each equation is applied and the cost contributed by each application are not independent, so taking the worst-case of each measure leads to imprecise AUBFs.

The imprecision of PUBS, among other issues, was addressed in Chapter 5 where precise and novel techniques for solving CRSs were proposed. They are based on defining the cost as a solution of a corresponding universally quantified FOL formula. This method, as expected, would obtain the most precise UBFs, however, it has two major limitations: (1) template UBFs have to be provided by the user; and (2) the use of QE renders the technique impractical, since QE is computationally expensive in the general case.

In this chapter we explore the gap between these two approaches, looking for solving techniques with an efficiency close to those of PUBS and a precision close to those of Chapter 5. Concretely, we develop a novel technique that splits the input CRS into atomic CRs of a simpler form, solves each of them separately, and then combines the resulting UBFs into UBFs for the original CRS. We call a CR atomic, if all equations contribute 0, except one equation that contributes a basic cost expression, i.e, of the form m, $\mathsf{nat}(l)$, $\log_m(\mathsf{nat}(n)+1)$, or $m^{\mathsf{nat}(n)}-1$. Our main observation is that it is enough to solve few atomic CRs precisely, using techniques similar to those of Chapter 5, while solving the others as in PUBS without affecting the overall precision.

In Sections 6.1 and 6.2 we propose two methods for precisely solving atomic CRs, which are based on the idea of specifying the cost using universally quantified FOL formulas as in Chapter 5. However, we do not require the user to provide any template (we always use linear templates), and, importantly, the generated VCs have almost a linear form for which QE can be done efficiently. In Section 6.3, we show how to handle the general case by reducing the problem of solving a given CRS into that of solving several atomic CRs. In Section 6.4 we discuss an experimental evaluation of the techniques. In Section 6.5 we overview related work, and in Section 6.6 we describe some further details that can be found in the article.

1 The Tree-Sum Method

In this section we describe the tree-sum method for solving atomic CRs. We first assume that the non-zero basic cost expression in the atomic CR is of the form $\mathsf{nat}(l)$, then, at the end of this section, we describe how to handle arbitrary basic cost expressions. We start with a motivating example for which the underlying techniques [5, 14] of PUBS infer an asymptotically imprecise UBF.

Example 6.1. Consider a CR C defined by the following equations:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = \mathsf{nat}(n) + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

Note that each application of the recursive equation contributes n (between 1 and m) units to the cost, and that the same quantity n is subtracted from the input m in the recursive call. Figure 6.1 depicts three evaluation trees for $C(m)$:

- In tree (a), the chain of recursive calls is the largest possible. In each node, the contributed cost is $n = 1$, which allows applying the recursive equation m times. Thus, the total cost is m;

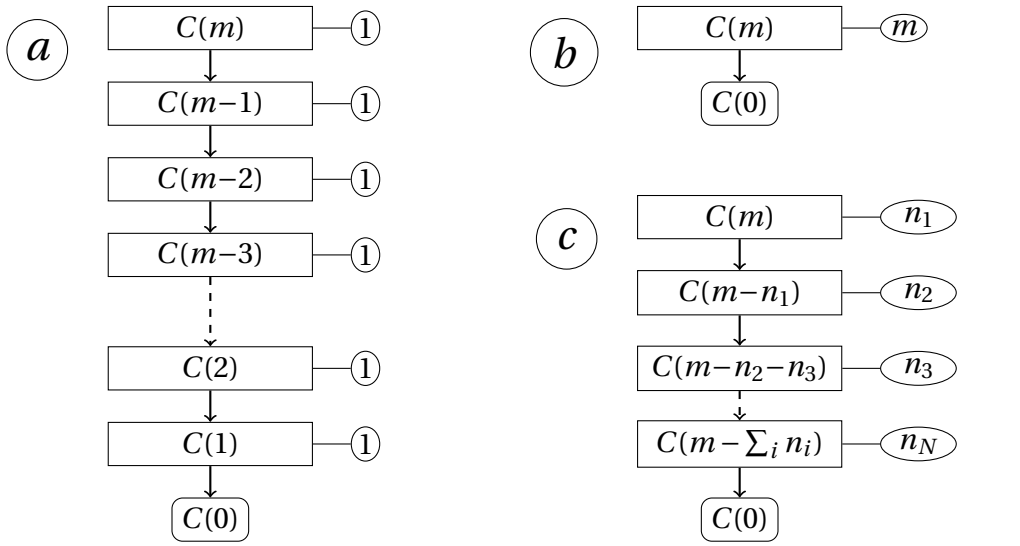

Figure 6.1: Evaluation trees for $C(m)$, with (a) maximum length, (b) maximum local cost, and (c) anything between them.

- In tree (b), the local cost of the first call is chosen as $n = m$, which prevents further applications of the recursive equation. The total cost is also m; and
- In tree (c), the recursive equation is applied N times, where $1 \leq N \leq m$. The i-th node, which corresponds to the i-th application, contributes n_i units to the cost such that $m = \sum_{i=1}^{N} n_i$. Still the total cost is also m.

From these trees, in particular tree (c), we can conclude that every evaluation of $C(m)$ has a total cost m. Thus, the most precise UBF is $C^+(m) = \mathsf{nat}(m)$. However, PUBS computes the UBF $C^+(m) = \mathsf{nat}(m)^2$ which is asymptotically less precise. This is because it multiplies the worst-case cost of all nodes, which is m as in Tree (b), by the maximum possible number of internal nodes, which is also m as in Tree (a). ■

In order to precisely handle examples like the one above, we should take into account the relation between the cost contributed in each node of an evaluation tree and the number of nodes in the same tree. This way, we can rule out (some) cases in which the worst-case node cost and worst-case number of nodes come from different trees. None of the techniques on which PUBS is based [5, 14] is able to take this relation into account. In what follows we describe an approach that is able to model this relation.

Our approach is based on the inductive assertion method for proving program correctness [34], like the the approach that we have described in Section 5.2. In particular, it is based on deriving a VC, from the CR this time, whose

validity implies the validity of some candidate UBFs. However, despite this similarity, in the case of atomic CRs solving the VC can be efficient as well.

Example 6.2. Given the CR of Example 6.1, and a candidate UBF $C^+(m)$, the corresponding VC is a conjunction of the following clauses:

$$\begin{array}{lllll} \forall m: & m = 0 & \rightarrow & C^+(m) \geq 0 \\ \forall m,n: & m \geq n \geq 1 & \rightarrow & C^+(m) \geq \mathsf{nat}(n) + C^+(m-n) \end{array}$$

The first clause corresponds to the first equation of the CR, and the second one to the second equation. Intuitively, each clause requires that the candidate UBF C^+ covers the local cost of the corresponding equation, and the cost of the recursive calls. The similarity to the net-cost formulas of Section 5.2 is clear. ■

To verify that a cost expression $f(m)$ is a valid UBF of $C(m)$, we replace $C^+(m)$ by $f(m)$ in the VC of Example 6.2 and then check its validity. However, as in Chapter 5, our interest is in synthesizing a UBF rather than checking the validity of a given one. This can be done as in Chapter 5 by using UBF templates, together with QE and SMT solving.

Example 6.3. Let $f(m) = \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}})$ be a UBF template for the CR of Example 6.1. Note that a_{m} and a_{c} are the template parameters. Substituting this template in the VC of Example 6.2 results in a FOL formula that is the conjunction of the following clauses:

$$\begin{array}{ll} \forall m: & m = 0 \quad \rightarrow \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}}) \geq 0 \\ \forall m,n: & m \geq n \geq 1 \quad \rightarrow \mathsf{nat}(a_{\mathrm{m}} m + a_{\mathrm{c}}) \geq \mathsf{nat}(n) + \mathsf{nat}(a_{\mathrm{m}}(m-n) + a_{\mathrm{c}}) \end{array}$$

Note that the parameters $a_{\mathrm{m}}, a_{\mathrm{c}}$ are free variables. Any satisfying assignment for this formula defines a template instance that is a UBF, e.g., $a_{\mathrm{m}} = 1, a_{\mathrm{c}} = 0$ and $a_{\mathrm{m}} = 2, a_{\mathrm{c}} = 1$ define $f(m) = \mathsf{nat}(m)$ and $f(m) = \mathsf{nat}(2m+1)$, respectively. As in Section 5.2, solving the VC means finding such an instance automatically. Again, we first apply a QE procedure to eliminate the variables n and m, which results in the equivalent quantifier-free constraint $a_{\mathrm{m}} \geq 1 \wedge a_{\mathrm{c}} \geq 0$. Clearly, the instances we mentioned above are solutions of this constraint. Next, we apply an SMT solver to obtain a particular solution of this constraint. ■

Note that the SMT may give a solution like $a_{\mathrm{m}} = 2 \wedge a_{\mathrm{c}} = 5$ that corresponds to a non-optimal UBF. In order to find an optimal assignment, one may use heuristics based on linear programing, or a greedy strategy like [66].

As we have discussed in Chapter 5, QE procedures can be computationally expensive in general. However, if we restrict ourselves to linear UBF templates, i.e., of the form $\mathsf{nat}(l)$ where l is a linear expression with parametric coefficients, as in Example 6.3, then the derived VC is of a particular form that is

almost linear. This is because we have restricted ourselves to atomic CRs with cost expressions of the form $\mathsf{nat}(l')$ as well. Fortunately, VCs of this form, unlike the general form used in Chapter 5, can be solved (or precisely approximated) using linear programming techniques that have polynomial-time complexity. This renders the overall approach practical. The article contains more details on how to solve such VCs.

Handling Arbitrary Basic Cost Expressions. Now we extend the above technique to handle basic cost expression b, i.e., $b = m$, $b = m^{\mathsf{nat}(l)} - 1$, or $b = \log_m(\mathsf{nat}(l) + 1)$, instead of only $b = \mathsf{nat}(l)$. Note, however, that we still require that only one equation has a non-zero cost expression. The case of $b = m$ is handled by considering it as $\mathsf{nat}(m)$, thus we concentrate on the other two cases. A possible solution is to generate the VC with these basic cost expressions, however, this will result in a VC that does not have the desired (almost linear) form, and thus solving it might be computationally expensive. Instead, our approach for solving such CRs is as follows: we first replace the basic cost expression ($m^{\mathsf{nat}(l)}-1$ or $\log_m(\mathsf{nat}(l)+1)$) by its sub-expression $\mathsf{nat}(l)$. This results in a new CR E that can be solved as described above to a UBF $E^+(\bar{x}) = \mathsf{nat}(L)$. Using this UBF, we then generate a UBF for C as follows:

$$C^+(\bar{x}) = \begin{cases} 1.5 * \mathsf{nat}(L) & \text{if } b = \log_m(\mathsf{nat}(l)+1) \\ m^{\mathsf{nat}(L)} - 1 & \text{if } b = m^{\mathsf{nat}(l)} - 1 \end{cases}$$

Example 6.4. Consider the CR C defined by the following equations:

$$\begin{array}{ll} C(m) = 0 & \{m = 0\} \\ C(m) = 2^{\mathsf{nat}(n)} - 1 + C(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

We replace $2^{\mathsf{nat}(n)} - 1$ by $\mathsf{nat}(n)$, resulting in the follwoing CR:

$$\begin{array}{ll} E(m) = 0 & \{m = 0\} \\ E(m) = \mathsf{nat}(n) + E(m-n) & \{m \geq n, n \geq 1\} \end{array}$$

We can solve this CR as described above to the UBF $E^+(m) = \mathsf{nat}(m)$. Then we define $C^+(m) = 2^{\mathsf{nat}(m)} - 1$ as a UBF for C. Note that applying PUBS on this CR we obtain the AUBF $C^+(m) = \mathsf{nat}(m) * 2^{\mathsf{nat}(m)}$, which is imprecise. ■

2 The Level-Sum Method

The technique of the previous section handles cases for which the given atomic CR admits a linear UBF. In this section, building on similar ideas, we develop

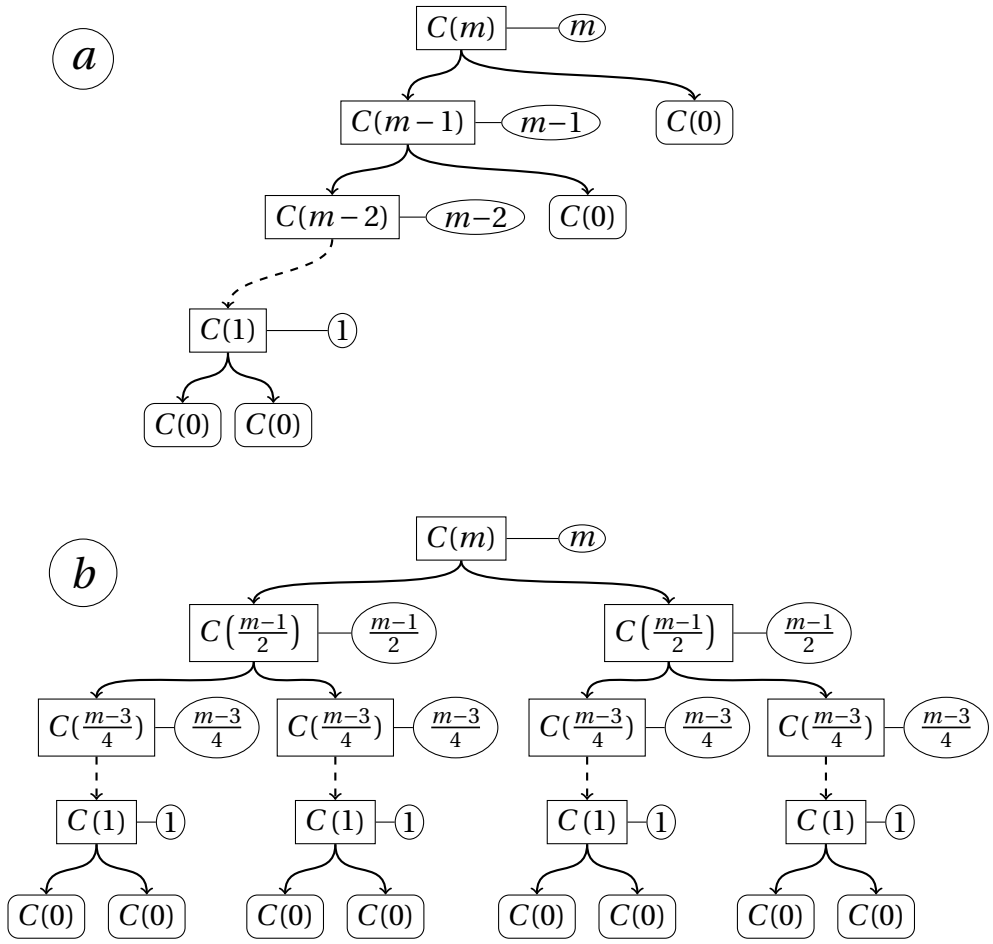

Figure 6.2: Evaluation trees of the CR $C(m)$.

an approach for solving atomic CRs that exhibit a divide and conquer like behaviour, which usually do not admit a linear UBF. Again, we first assume that the non-zero cost expression in the atomic CR is of the form $\mathsf{nat}(l)$, then, at the end of this section, we describe how to handle other basic cost expressions. We start with a motivating example that is used throughout this section.

Example 6.5. Consider a CR C that is defined by the following equations

$$\begin{array}{lll} C(m) = & 0 & \{m \geq 0\} \\ C(m) = & \mathsf{nat}(m) + C(m_1) + C(m_2) & \{m = m_1 + m_2 + 1, m_1 \geq 0, m_2 \geq 0\} \end{array}$$

Note that the cost contributed by the recursive equation, in each application, is equal to the input m, and that its recursive calls are applied on m_1 and m_2 such that $m = m_1 + m_2 + 1$. This models a typical divide and conquer behaviour.

Depending on the values chosen for m_1 and m_2 in each application of the recursive equation, a call $C(m)$ can be evaluated to (asymptotically) different values. For example, choosing $m_1 = m-1$ and $m_2 = 0$ in all applications results in the worst-case $\frac{m^2+m}{2}$, as depicted in Tree (a) of Figure 6.2. On the other hand, splitting $m-1$ equally between m_1 and m_2 results in the best-case which is in $\Theta(m \log_2 m)$, as depicted in Tree (b) of Figure 6.2. Applying PUBS on this example infers the AUBF $C^+(m) = \mathsf{nat}(m) * 2^{\mathsf{nat}(m)}$, which is imprecise. ■

Handling the CR of the above example using the tree-sum method requires a polynomial UBF template, otherwise it fails. This is because it has an evaluation tree with polynomial cost as Tree (a) of Figure 6.2. However, recall that we are interested in using only linear UBF templates, which then allows solving the corresponding VC efficiently. Thus, using the tree-sum method is ruled out. Instead, in what follows, we describe an alternative approach for handling such CRs that it is based on a different kind of VCs and uses only linear UBF templates.

The intuition behind our approach is that in divide and conquer examples, a cost expression $w(\bar{x}) * h(\bar{x})$ is a UBF for the corresponding CR if (i) $w(\bar{x})$ is a *cost expression* that bounds the total cost contributed by *each level* of any evaluation tree; and (ii) $h(\bar{x})$ is a *cost expression* that bounds the height of all evaluation trees. For example, for the CR of Example 6.5 we have $w(m) = \mathsf{nat}(m)$ and $h(m) = \mathsf{nat}(m)$, which results in the UBF $C^+(m) = \mathsf{nat}(m)^2$. Thus, we could aim first at inferring $w(\bar{x})$ and $h(\bar{x})$, and then combine them into a UBF for the corresponding CR. The advantage is that these bounds are typically linear, and thus can be synthesized efficiently. The next example describes the basics of this approach, by deriving appropriate VCs and synthesizing $w(m)$ and $h(m)$ for the CR of Example 6.5.

Example 6.6. Consider the CR of Example 6.5 and, for simplifying the presentation, let φ_2 refer to the constraints attached to the second equation. We say that a given cost expression $h(m)$ is a valid bound on the height of any evaluation tree of C if the following VC is valid:

$$\forall m, m_1, m_2 : \varphi_2 \quad \rightarrow \quad h(m) \geq 1 + h(m_1) \wedge h(m) \geq 1 + h(m_2)$$

Intuitively, since the height in this case corresponds to the number of consecutive applications of the recursive equation, the above VC requires that $h(m)$ covers one application of the recursive equation, and further ones that might arise through each recursive call. Note that each recursive call is considered separately (the clause has a conjunction inside), since they correspond to different paths in the evaluation tree. Observe that it is crucial that $h(m)$ is a cost expression, since this guarantees that it is always non-negative, otherwise the above VC is not correct.

Similarly, we say that a cost expression $w(m)$ is a valid bound on the cost of each level if the VC defined as a conjunction of the following clauses is valid:

$$\begin{array}{lcl} \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq \mathsf{nat}(m) \\ \forall m, m_1, m_2 : \varphi_2 & \rightarrow & w(m) \geq w(m_1) + w(m_2) \end{array}$$

Intuitively, the first clause requires that $w(m)$ covers the cost contributed by the recursive equation at any level and the second clause requires that it also covers the next level. Again, it is crucial that $w(m)$ is a cost expression, since this guarantees that it is always non-negative.

In order to automatically synthesize the cost expressions $h(m)$ and $w(m)$ for which the above VCs are valid, we use the UBF templates $h(m) = \mathsf{nat}(h_{\mathrm{m}} m + h_{\mathrm{c}})$ and $w(m) = \mathsf{nat}(w_{\mathrm{m}} m + w_{\mathrm{c}})$, and then QE to eliminate the universally quantified variables. This results in the constraints $h_{\mathrm{m}} \geq 1 \wedge h_{\mathrm{c}} + h_{\mathrm{m}} \geq 1$ and $w_{\mathrm{m}} \geq 1 \wedge w_{\mathrm{m}} \geq w_{\mathrm{c}} \wedge w_{\mathrm{m}} + w_{\mathrm{c}} \geq 1$, that admit the solutions $h_{\mathrm{m}} = 1, h_{\mathrm{c}} = 0$ and $w_{\mathrm{m}} = 1, w_{\mathrm{c}} = 0$, respectively. Thus, $h(m) = \mathsf{nat}(m)$ and $w(m) = \mathsf{nat}(m)$ are valid bounds on the height and the cost of each level. ■

For a general atomic CR, unlike the CR that we used above, there might be recursive equations that do not contribute to the cost, i.e., they have local cost zero. This means that some level of the evaluation tree will have cost 0, and it would be more precise to ignore these levels when computing $h(\bar{x})$. To achieve this, instead of bounding the height of an evaluation tree, we bound the number of nodes with non-zero cost in each path from the root to a leaf. Similarly, instead of considering the level of a node as its distance from the root, it would be more precise to define it as the number of non-zero nodes from the root to that node. This generalization is formalized in the article.

Handling Arbitrary Basic Cost Expressions. Now we discuss how the above technique is extended to handle an arbitrary basic cost expression b, i.e., $b = m$, $b = m^{\mathsf{nat}(l)} - 1$, or $b = \log_m(\mathsf{nat}(l) + 1)$, instead of only $b = \mathsf{nat}(l)$. Note, however, that we still require that only one equation has a non-zero basic cost expression. Recall that in the level-sum method we have two VCs, the one that corresponds to $h(\bar{x})$ remains the same since the height of an evaluation trees does not depend on b. Handling the VC that corresponds to $w(\bar{x})$ is done exactly as we described for the case of the tree-sum method.

3 Handling General CRSs

Without loss of generality, we assume that the input CRS is a stand-alone CR, i.e., it includes only one CR symbol C. Handling the general case is done exactly as in PUBS, by solving them one CR at a time as in Section 2.5.

The tree-sum and level-sum methods were designed to handle atomic CRs in which all equations contribute zero to the cost, except one equation that can contribute a basic cost expression. However, in practice, CRs have several equations with non-zero cost, and, moreover, they use arbitrary cost expressions not necessarily basic. In what follows we describe our approach for handling such general CRs. It is based on the idea of breaking the input CR into several atomic CRs, solving the atomic CRs into UBFs by using the tree-sum or level-sum methods, and finally combining these UBFs into a UBF for the original CR.

For simplicity, we may assume that cost expressions in the input CR do not use the max operator, for if they use it, we can replace each max operator by the sum of its parameters, just as we did in Section 3.2. Another alternative is to clone the corresponding equation into several cases, one for each parameter of the max operator. We also assume that cost expressions are given in normal form, namely, each cost expression is a sum of products of basic cost expressions. We present our approach in two steps: We first extend atomic CRs to allow using a product of basic cost expressions; and then we consider a general case in which we have several equations that contribute arbitrary non-zero cost expressions (given in a normal form).

Products. Assume a given CR C in which all equations contribute zero, except for one equation that contributes $e = b_1 * \cdots * b_n$ where b_i, $1 \leq i \leq n$, are basic cost expressions. Our method to solve C into a UBF proceeds as follows:

1. We pick b_i, for some $1 \leq i \leq n$, and replace e by this b_i. This results in a CR E that is atomic;
2. We solve E into a UBF $E^+(\bar{x})$ using the tree-sum or level-sum methods;
3. For any b_j, with $j \neq i$, we compute its maximisation, denoted by $\hat{b}_j$, as described in Section 2.5; and
4. We define $C^+(\bar{x}) = \hat{b}_1 * \cdots * \hat{b}_{i-1} * E^+(\bar{x}) * \hat{b}_{i+1} * \cdots * \hat{b}_n$ as a UBF for C.

The correctness of this method relies on the fact that $\hat{b}_j$ is larger than any instance of b_j, and, moreover, it is a constant expression since it is given in terms of the initial input parameters and not in terms of the CR variables. Therefore we can factorized it out of the CR. Note that the nondeterministic choice of i in the first step might lead to different UBFs.

Example 6.7. Consider a CR C that is defined by the following equations:

$$\begin{array}{ll} C(m,p) = 0 & ,\{m = 0\} \\ C(m,p) = \mathsf{nat}(q) * \mathsf{nat}(n) + C(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

Note that the cost contributed by the second equation is a product of two basic cost expressions. We replace this product by $\mathsf{nat}(n)$, which results in the follow-

ing atomic CR:

$$\begin{array}{ll} E(m,p) = 0 & ,\{m = 0\} \\ E(m,p) = \mathsf{nat}(n) + E(m-n,p') & ,\{m \geq n \geq 1, p \geq q, p \geq p' \geq 0\} \end{array}$$

Using the tree-sum method we solve E into the UBF $E^+(m,p) = \mathsf{nat}(m)$. Now we maximise the other basic cost expression $\mathsf{nat}(q)$ into $\mathsf{nat}(p)$, which is the maximum value to which $\mathsf{nat}(q)$ can be evaluated. Note that p here refers to the initial input value and not to the parameter of the CR E. Finally, we define $C^+(m,p) = \mathsf{nat}(p) * \mathsf{nat}(m)$ as a UBF for CR C. ■

The General Case. Assume a general CR C with k equations, such that the i-th equation contributes a cost expression $e_i = P_{i1} + \cdots + P_{in_i}$, where each addend P_{ij} is a product of basic cost expressions. Our method to solve C into a UBF proceeds as follows:

1. For each product P_{ij}, we define a CR E_{ij} that is derived from C by removing all other products from C;
2. We solve each E_{ij} into a UBF $E^+_{ij}(\bar{x})$, as above; and
3. We build the UBF $C^+(\bar{x})$ of $C(\bar{x})$ as $C^+(\bar{x}) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} E^+_{ij}(\bar{x})$.

The correctness of this approach relies on the fact that each CR E_{ij} models the maximum possible contribution of the addend P_{ij} to the total cost of C.

Example 6.8. Consider a CR C that is defined by the following equations:

$$\begin{array}{lll} \text{Eq.1} & C(m,n) = 0 & ,\varphi_1 \\ \text{Eq.2} & C(m,n) = \underbrace{\mathsf{nat}(n)}_{P_{21}} + C(m,n_1) & ,\varphi_2 \\ \text{Eq.3} & C(m,n) = \underbrace{\mathsf{nat}(m_0)}_{P_{31}} + \underbrace{\mathsf{nat}(n)}_{P_{32}} + C(m_1,n_1) + C(m_2,n_2) & ,\varphi_3 \end{array}$$

where $\varphi_1 \equiv \{m_0 = 0, n = 0\}$, $\varphi_2 \equiv \{n \geq 1, n = 1 + n_1\}$, and $\varphi_3 \equiv \{m \geq m_0 + m_1 + m_2, n \geq 1 + n_1 + n_2\}$. In this CR there are three products: $P_{21} = \mathsf{nat}(n)$, $P_{31} = \mathsf{nat}(m_0)$, and $P_{32} = \mathsf{nat}(n)$.

To solve the CR C, we first split it into the CRs E_{21}, E_{31}, and E_{32}, depicted in Figure 6.3. Solving these CRs we get the UBFs $E^+_{21}(m,n) = \mathsf{nat}(n)^2$, $E^+_{31}(m,n) = \mathsf{nat}(m)$, and $E^+_{32}(m,n) = \mathsf{nat}(n)^2$. Then, we define $C^+(m,n) = \mathsf{nat}(n)^2 + \mathsf{nat}(m) + \mathsf{nat}(n)^2$ as a UBF for $C(m,n)$. ■

4 Experimental Evalauation

We implemented our techniques as an extension of PUBS. The implementation took less than 750 lines of Prolog code.

$$
\begin{array}{lll}
E_{21}(m,n) = & 0 & ,\varphi_1 \\
E_{21}(m,n) = & \mathsf{nat}(n)+ \quad E_{21}(m,n_1) & ,\varphi_2 \\
E_{21}(m,n) = & 0+ \quad E_{21}(m_1,n_1)+E_{21}(m_2,n_2) & ,\varphi_3 \\
E_{31}(m,n) = & 0 & ,\varphi_1 \\
E_{31}(m,n) = & 0+ \quad E_{31}(m,n_1) & ,\varphi_2 \\
E_{31}(m,n) = & \mathsf{nat}(m_0)+ \quad E_{31}(m_1,n_1)+E_{31}(m_2,n_2) & ,\varphi_3 \\
E_{32}(m,n) = & 0 & ,\varphi_1 \\
E_{32}(m,n) = & 0+ \quad E_{32}(m,n_1) & ,\varphi_2 \\
E_{32}(m,n) = & \mathsf{nat}(n)+ \quad E_{32}(m_1,n_1)+E_{32}(m_2,n_2) & ,\varphi_3
\end{array}
$$

Figure 6.3: Decomposition of a CR in three smaller CRs in the general case

Precision. To evaluate the asymptotic precision of our techniques, we gathered from related literature some programs. Although this programs are short, they pose to cost analysis some challenge that is solved by our methods. In particular, for the tree-sum method we consider some programs usually described using the notions of amortised cost, such as the procedure to add many elements into a list implemented with a resizable array [45, §17], the procedure to add and remove many elements from a queue implemented with two stacks [104]. For the level-sum method we consider some *divide and conquer* algorithms, like QuickSort. In the article we show for each example the AUBF obtained by COSTA using the method to solve CRs of [5], and the one obtained using ours. In all examples, the methods of [5] obtain an imprecise AUBF, while our method obtains the precise AUBF of the program. Although PUBS already had a technique for solving divide-and-conquer CRs [5], it had a restricted applicability. For instance, it can not be applied to bind the cost of QuickSort.

Scalability Following [5, §10], we have composed the programs of the previous experiments to build a set of programs of increasing size as set of benchmarks. For each benchmark, we measured the time for solving the CRS using the methods of PUBS and using ours. In the experiments, we observed that the runtimes of our solving method is reasonable, and within a factor with respect to the runtime of the methods of [5].

We have also compared our approach to the ACRP prototype of [18]. For all benchmarks, ACRP failed to obtain an UBF in less than a minute. This is expected since ACRP is based on a general QE procedure for nonlinear real arithmetic, which is not scalable. Instead, our methods solved all programs in a few seconds. This is due to the use of a QE method based in linear programming.

In short, in our experiments we see that our method is more precise than the method of [10], and that its scalability is similar to that of that method.

5 Related Work

Automatic Solvers for Recurrence Relations. The first support for automatically solving RRs into closed form appeared in computer algebra systems like MACSYMA [80] and MATHLAB [42]. PURRS [27] is a relatively recent system that aims at solving recurrence relations as well, however, it tries to approximate the solution when it fails to find an exact one. All these systems assume deterministic recurrences, which cannot directly model the cost of nondeterministic (abstract) programs. The resolution of divide and conquer recurrences has been studied in [55].

PUBS. The closest works to ours are those developed in the context of PUBS, which solves CRSs into UBFs and LBFs. The syntax, semantics of CRSs, and differences from recurrence relations, are described in [5]. This article also discusses two methods for solving CRSs: (i) the node-count method, that we described in Section 2.5; and (ii) the level-count method, which aims at solving CRSs that originate from divide and conquer algorithms as our level-sum method, but it has restricted applicability.

Cost expressions were also introduced in [5], as a form of monotonic expressions that are generated from linear expressions. The key insight in cost expressions is that, since they are monotonic on their nat components, maximising a cost expression is done by maximising the linear expressions inside the nat expression. Thus, cost expressions are a leverage to reason about complex expressions using linear programming techniques. In the same way, our decomposition of a CR into atomic CRs is a leverage to solve complex CRs using QE tools for linear arithmetic.

PUBS also relies on the techniques of [14] which solve CRSs by first transforming then into RRs that approximate their worst-case, and then solve these RRs with computer algebra systems. This approach can obtain UBFs that are more precise than those of [5], and can be easily used to infer LBFs as well. However, its applicability is more restricted, and still obtains an asymptotically imprecise UBF for our examples.

Logical Reasoning and Quantifier Elimination. The tree-sum method and the level-sum method are inspired by the inductive assertion method for proving program correctness [34]. The use of QE methods for linear arithmetic has been used in other related works. It is the basis for automatic inference of linear ranking functions [26], and lexicographical ranking functions [15]. In [126] inductive assertions, combined with templates, are used for synthesis and verification of programs. Their tool handles a wider class of template predicates, that include disjunctions and conjunctions of linear constraints.

Reachability Bound. Our level-sum method bounds the height of the evaluation trees of a CRS, which can be seen as the number of *subsequent* visits to an equation, which is a *reachability bound*. Such a bound can be directly computed as a linear ranking function [26]. In systems with bounded nondeterminism, it can also be computed from a lexicographical (or multidimensional) linear ranking function [15]. In [67], the authors define this problem and present some tools for computing such bounds using an SMT solver. Their technique allows computing UBFs that are products or a max of two nat expressions. In [141], they combine this approach with the size-change abstraction. Some works have proposed inferring polynomial ranking functions: some by using QE methods for nonlinear real arithmetic [38], others by using the technique of Lagrangian relaxation to reduce it to a linear programming problem [47]. Other techniques have been proposed in [98].

6 Further Reading

In this Chapter we have described the technical contributions of [17]. We explained how to solve a CR by splitting it into several atomic CRs, how to solve these atomic CRs into corresponding UBFs, and how to combine these UBFs into a UBF for the original CR. We have also explained the tree-sum and level-sum methods for solving atomic CRs. These methods are based on the use of universally quantified FOL formulas as in Chapter 5, however, they can be solved efficiently using QE for linear arithmetic.

The article extends and formalises the content of this chapter. It includes a formal definition for the syntactic class of atomic CRs. It also formalises the tree-sum method and the level-sum methods, and provides correctness statements and their corresponding proofs. The article describes the QE procedure that we use to solve the different VCs, which is based on the use of Farkas Lemma, and how to adapt it to handle the nat expressions in the VCs. The article also contains the tables with the results of our experiments.

7 | Conclusions, and Future Work

In this chapter we draw the conclussions of this dissertation. In Section 7.1 we describe the achievements, impact, and practical applicability of our contributions. In Section 7.2 we give a extended discussion about our conclussions about the relation between the classical and the amortised approaches to cost analysis. Finally, in Section 7.3 we sketch some possible lines for future work.

1 Objectives, Achievements, and Impact

Our first objective was to adapt existing tools to compute asymptotic UBFs as well, which are a natural choice for specifying the expected cost of a program under development since they are (i) less sensitive to small changes in the program, and (ii) concise and easier to interpret for large programs. This objective has been achieved in Chapter 3 as follows:

- Initially, we have followed a transformational approach and developed automatic methods to transform UBFs, or more precisely cost expressions, to asymptotic form. The advantage of this approach is that it can be applied directly to UBFs obtained by any cost analyser, not necessarily COSTA. The disadvantage is that we still have to obtain non-asymptotic UBFs before applying the transformation.
- Afterwards, we developed a novel asymptotic CRS solver that directly solves CRSs into asymptotic UBFs. This was achieved by interleaving the different phases of PUBS with the transformation of cost expressions to asymptotic form. This solver can be used for any cost analysis that generates CRSs. Experimental evaluation confirmed the superiority of this last approach over the first one.

An important feature of our approach, unlike previous ones [128], is that it takes context information into account.

Our second objective was to explore the precision gap between existing approaches to cost analysis. This included (i) understanding the reasons for which COSTA, and Wegbreit's classical approach in general, infers imprecise UBFs for some programming patterns; and (ii) developing novel techniques to close this precision gap. We have identified several sources for this imprecision, each one is related to a different phase of COSTA, and proposed novel techniques to overcome these problems:

- In Chapter 4 we have addressed the imprecision induced by the use of non-linear arithmetic operations, which cannot be modeled in COSTA since it

relies on the use of linear constraints. We have observed that in the presence of context information, one can model nonlinear operations with linear constraints only, and, moreover, this can be encoded directly in the ACR program which avoids significant changes to the underlying analyser.
- In Chapter 5 we have observed a surprising relation between the output of a procedure and its cost, which is the main reason for some well-known precision problems. This imprecision is related to the phase that transforms the ACR program to the CRS (since output parameters are removed). To solve this problem, we have developed novel techniques that are based on specifying the cost as a FOL formula over some UBF templates, and then using QE procedures and SMT solving to compute concrete UBFs. Our approach supports releasing resources as well, and thus gave rise to the notion of peak-cost.
- In Chapter 6, we combined the techniques developed in Chapter 5 with those of PUBS, to develop a precise and scalable method to solve a CR into a UBF. This was achieved by splitting the input CRS into several atomic ones, whose cost can be modeled using simpler FOL formulas that can be solved by an efficient QE procedure. Our experiments show that our new techniques are close to those of Chapter 5 in terms of precision, and to those of PUBS in terms of scalability.

Concluding the achievements related to the last objectives: apart from justifying, for the first time, the source of some imprecision problems in the classical approach to cost analysis, our major contribution is to apply *The Calculus of Computation* [34] to static cost analysis, by turning UBF inference and CRS solving into the verification of an abstract program using the current progress on SMT solving [91] and QE [83] procedures.

Our contributions have a fundamental impact on both practical and theoretical aspects of cost analysis. In the practical case, this is witnessed by the new features that were added to COSTA, and the improvements in its applicability, precision, and scalability.

- Our first contribution extends the features of COSTA with the possibility of inferring asymptotic bounds. As shown in Section 3.5, the asymptotic CRS resolution improves the scalability of COSTA and, indirectly, its practical applicability to the larger programs.
- Our second contribution has extended the applicability of COSTA to JBC programs whose cost depends on a nonlinear instruction. As shown in Section 4.2, this contribution allows COSTA to infer a stronger value relations, with which PUBS is able to infer more precise AUBFs. Moreover, our solution does not compromises the scalability of COSTA, not only because

uses a non-disjunctive abstract domains, but also because it does not increase the size of the generated CRS.

- The impact of our third contribution (Chapter 5) goes beyond the practical aspects: it sheds new light on the relation between the different approaches to cost analysis. On the practical side, as discussed in Section 5.5, we implemented our approach in a prototype called ACRP. ACRP is able to infer precise AUBFs for many programs for which COSTA fails, including some classical examples related to amortised analysis, and some nested loops. However, ACRP uses the SMT and QE methods for the FOL theory of real fields (real numbers), which are not scalable. As a result, this contribution has no practical applicability in the context of COSTA, which is why we have not integrated it in there.
- In our fourth contribution we have improved the precision of PUBS, by developing a new method to solve a CR into a UBF. As shown in Section 6.4, our technique achieves a better asymptotic precision than those of [5, 14]. By relying on a QE method for linear real arithmetic, our method does not damage the scalability of COSTA. Our method also extend the applicability of COSTA and PUBS to some programs for which there are no linear ranking functions, even to some non-terminating programs, since our method does not need to compute a linear ranking function.

Concluding the impact, apart from the evidence (see Sections 3.5, 4.2, 5.5, and 6.4) on the practical improvements to COSTA, we believe that the theoretical contributions of Chapter 5 will make it easier to transfer knowledge and implementation techniques from one approach of cost analysis to another.

Applicability of our Contributions

Our major contributions are developed for ACR programs and CRSs. Thus, they are generic in the source programming language, the cost model, or the size measure used. Therefore, they can be also applied to non-accumulative cost models like heap consumption [13], or concurrent tasks [11], or to any user-definable cost model [100, 99]. In the same way, they can be applied if the size measure is the path-length of heap-allocated data structures [125], or the introspective size measures for functional data structures used in RAML [75], or any user definable size measure [66]. With respect to programming languages, the abstract compilation of the RBR into the ACR can be adapted to consider such features as field-sensitive analysis over heap-allocated data [6]. Our contributions can also be applied to the COSTABS [3] analyser for the concurrent ABS language.

2 Amortised Cost Analysis

When we started our research, we found several programs for which COSTA inferred imprecise UBFs. Some of these were (functional) programs for which the methods for automated amortised analysis [81, 75] could infer precise UBFs. Some other programs consisted of a single loop for which COSTA inferred an imprecise UBF, while the methods of [65, 67] could infer the precise *amortised* complexity. Finally, there were some data structures for imperative languages [45, §17], that COSTA could not handle. Since all these examples were described in terms of amortised cost or amortised complexity, while seeking for the source of imprecision in COSTA we had amortised analysis in mind.

Unfortunately, we did not found a clear definition of amortised cost in the literature. The seminal paper of Tarjan [131] described two methods for manual cost analysis, the banker's method and the physicist's method, which he used to analyse programs that performed sequences of operations on some data-structures. He used the word *amortisation* to illustrate the change of focus from analysing one operation to analysing a sequence of them. Thus, the term *amortised cost* is a metaphor to describe the result of studying the *worst-case* cost of a program with certain methods. However, other works [104] use it to indicate a different property of program semantics.

We believe that our contributions help to clarify the notion of amortised cost, and exactly identify the components due to which COSTA was imprecise for the programs mentioned above:

- On one hand, the output-cost codependency described in Section 5.1 explains why COSTA infers an asymptotically imprecise UBF, for instance, for the nested loops in [65, 67], as well as in the classical binary counter data structure [45, §17], or in Tarjan's stack example [131].
- On the other hand, there are programs associated to the notion of amortised cost for which the output-cost codependency does not appear. In terms of COSTA, this means that corresponding CRSs accurately model the corresponding costs. This is the case of the Dynamic Array data structure [45, §17.4], or the implementation of a functional queue with two stacks [104, §3]. For such programs, the imprecision is related to the CRS resolution phase, in particular to the inability of PUBS to track dependencies between the costs of the different iterations of a loop, as in Chapter 6.

Our work is the first to make the above observations.

3 Future Work

There are several ways in which our contributions can be improved:

- In Chapter 6 we focused on improving the asymptotic precision of the computed UBFs, and improving the asymptotic one is left for future work. For this, we need a better way to handle max expressions, or a CR with many equations. For instance, we may use a greedy counter optimisation technique in the manner of SPEED [66].
- The techniques of Chapter 6 for solving atomic CRs can be extended to handle more CRs. This can be done by looking for patterns of evaluation trees that are typically used in complexity analysis [45].
- A we have discussed in Chapter 5, it easy to adapt our techniques to infer LBFs on the net-cost or peak-cost of terminating programs. As future direction could address this problem for non-terminating programs.
- The techniques of Chapter 5 required UBF templates to be provided by the user. A future research direction could concentrate on inferring such templates automatically, by syntactic analysis of the corresponding CRS.
- The use of QE and SMT solvers over linear arithmetic can be adapted for other problems in the context of solving CRSs. For example, we can use in the context of [14] to infer the arithmetic or geometric increasing/decreasing factors required for translating CRSs to RRs. We could also use it for building an alternative maximisation procedure that is not based on invariant generation, or for inferring nonlinear ranking functions.

All these improvements would have an impact both on the precision and on the applicability of cost analysis.

Bibliography

[1] ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, JESÚS CORREAS, ANTONIO FLORES-MONTOYA, SAMIR GENAIM, MIGUEL GÓMEZ-ZAMALLOA, ABU NASER MASUD, GERMAN PUEBLA, JOSÉ MIGUEL ROJAS, GUILLERMO ROMÁN-DÍEZ, AND DAMIANO ZANARDINI. Automatic Inference of Bounds on Resource Consumption. In Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue, editors of the Proceedings of *International Symposium Formal Methods for Components and Objects (FMCO)*, volume 7866 of *Lecture Notes in Computer Science*, pages 119–144. Springer, September 2013.

[2] ELVIRA ALBERT, DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, SAMIR GENAIM, AND GERMAN PUEBLA. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009.

[3] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, MIGUEL GÓMEZ-ZAMALLOA, AND GERMÁN PUEBLA. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors of the Proceedings of *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 151–154. ACM, January 2012.

[4] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, ISRAEL HERRAIZ, AND GERMAN PUEBLA. Comparing Cost Functions in Resource Analysis. In Marko C. J. D. van Eekelen and Olha Shkaravska, editors of the Proceedings of *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, volume 6324 of *Lecture Notes in Computer Science*, pages 1–17. Springer, November 2010.

[5] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, AND GERMÁN PUEBLA. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[6] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND GUILLERMO ROMÁN-DÍEZ. Conditional termination of loops over heap-allocated data. *Science of Computer Programming*, 92:2–24, 2014.

[7] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMAN PUEBLA, AND DAMIANO ZANARDINI. COSTA: Design and Implementation of a Cost and

Termination Analyzer for Java Bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors of the Proceedings of *International Symposium Formal Methods for Components and Objects (FMCO)*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, October 2008.

[8] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND DAMIANO ZANARDINI. Removing useless variables in cost analysis of Java bytecode. In Roger L. Wainwright and Hisham Haddad, editors of the Proceedings of *ACM Symposium on Applied computing (SAC)*, pages 368–375. ACM, March 2008.

[9] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, AND DAMIANO ZANARDINI. Resource Usage Analysis and Its Application to Resource Certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors of the Proceedings of *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 258–288. Springer, 2009.

[10] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMAN PUEBLA, AND DAMIANO ZANARDINI. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[11] ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, AND DAMIANO ZANARDINI. Task-level analysis for a language with async/finish parallelism. In Jan Vitek and Bjorn De Sutter, editors of the Proceedings of *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 21–30. ACM, 2011.

[12] ELVIRA ALBERT, JESÚS CORREAS, GERMÁN PUEBLA, AND GUILLERMO ROMÁN-DÍEZ. Quantified Abstractions of Distributed Systems. In Einar Broch Johnsen and Luigia Petre, editors of the Proceedings of *Integrated Formal Methods (iFM)*, volume 7940 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2013.

[13] ELVIRA ALBERT, SAMIR GENAIM, AND MIGUEL GÓMEZ-ZAMALLOA. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.

[14] ELVIRA ALBERT, SAMIR GENAIM, AND ABU NASER MASUD. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, August 2013.

[15] CHRISTOPHE ALIAS, ALAIN DARTE, PAUL FEAUTRIER, AND LAURE GONNORD. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In Radhia Cousot and Matthieu Martel, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, September 2010.

[16] DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Handling Non-linear Operations in the Value Analysis of COSTA. *Electronic Notes in Theoretical Computer Science*, 279(1):3–17, 2011. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE).

[17] DIEGO ESTEBAN ALONSO-BLAS, PURI ARENAS, AND SAMIR GENAIM. Precise Cost Analysis via Local Reasoning. In Dang Van Hung and Mizuhito Ogawa, editors of the Proceedings of *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, October 2013.

[18] DIEGO ESTEBAN ALONSO-BLAS AND SAMIR GENAIM. On the Limits of the Classical Approach to Cost Analysis. In Antoine Miné and David Schmidt, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012.

[19] HUGH ANDERSON, SIAU-CHENG KHOO, STEFAN ANDREI, AND BEATRICE LUCA. Calculating Polynomial Runtime Properties. In Kwangkeun Yi, editor of the Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 230–246. Springer, November 2005.

[20] JASON ANSEL AND CY P. CHAN. PetaBricks. *Crossroads*, 17(1):32–37, 2010.

[21] ANDREW W. APPEL. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[22] MICHAEL ARMBRUST, ARMANDO FOX, REAN GRIFFITH, ANTHONY D. JOSEPH, RANDY KATZ, ANDY KONWINSKI, GUNHO LEE, DAVID PATTERSON, ARIEL RABKIN, ION STOICA, AND MATEI ZAHARIA. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[23] ROBERT ATKEY. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, 7(2), 2011.

[24] ROBERTO BAGNARA, PATRICIA M. HILL, AND ENEA ZAFFANELLA. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[25] ROBERTO BAGNARA, PATRICIA M. HILL, AND ENEA ZAFFANELLA. Weakly-relational shapes for numeric abstractions: improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[26] ROBERTO BAGNARA, FRED MESNARD, ANDREA PESCETTI, AND ENEA ZAFFANELLA. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215:47–67, 2012.

[27] ROBERTO BAGNARA, ANDREA PESCETTI, ALESSANDRO ZACCAGNINI, AND ENEA ZAFFANELLA. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR*, abs/cs/0512056, 2005.

[28] ROBERTO BAGNARA, ENRIC RODRÍGUEZ-CARBONELL, AND ENEA ZAFFANELLA. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In Chris Hankin and Igor Siveroni, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 19–34. Springer, September 2005.

[29] CLARK BARRETT, AARON STUMP, AND CESARE TINELLI. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors of the Proceedings of *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[30] FLORENCE BENOY AND ANDY KING. Inferring Argument Size Relationships with CLP(R). In John P. Gallagher, editor of the Proceedings of *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer, August 1997.

[31] RALPH BENZINGER. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

[32] RALPH BENZINGER. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.

[33] DOMINIQUE BOUCHER AND MARC FEELEY. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In Tibor Gyimóthy, editor

of the Proceedings of *International Conference on Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 192–207. Springer, April 1996.

[34] AARON R. BRADLEY AND ZOHAR MANNA. *The Calculus of Computation - Decision Procedures with Applications to Verification.* Springer, 2007.

[35] BRIAN CAMPBELL. Amortised Memory Analysis Using the Depth of Data Structures. In Giuseppe Castagna, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 190–204. Springer, March 2009.

[36] YONGJAE CHA, MARK VAN HOEIJ, AND GILES LEVY. Solving recurrence relations using local invariants. In Wolfram Koepf, editor of the Proceedings of *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 303–309. ACM, July 2010.

[37] LIQIAN CHEN, ANTOINE MINÉ, JI WANG, AND PATRICK COUSOT. Linear Absolute Value Relation Analysis. In Gilles Barthe, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 6602 of *Lecture Notes in Computer Science*, pages 156–175. Springer, March 2011.

[38] YINGHUA CHEN, BICAN XIA, LU YANG, AND NAIJUN ZHAN. Generating Polynomial Invariants with DISCOVERER and QEPCAD. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors of the Proceedings of *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 67–82. Springer, September 2007.

[39] THOMAS CLUZEAU AND MARK VAN HOEIJ. Computing Hypergeometric Solutions of Linear Recurrence Equations. *Appl. Algebra Eng. Commun. Comput.*, 17(2):83–115, 2006.

[40] MICHAEL CODISH. Efficient Goal Directed Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[41] JACQUES COHEN. Computer-Assisted Microanalysis of Programs. *Communications of the ACM*, 25(10):724–733, 1982.

[42] JACQUES COHEN AND JOEL KATCOFF. Symbolic Solution of Finite-Difference Equations. *ACM Transactions on Mathematical Software*, 3(3):261–271, 1977.

[43] JACQUES COHEN AND ALINE WEITZMAN. Software Tools for Microanalysis of Programs. *Software - Practice and Experience*, 22(9):777–808, 1992.

[44] JACQUES COHEN AND CARL ZUCKERMAN. Two Languages for Estimating Program Efficiency. *Communications of the ACM*, 17(6):301–308, 1974.

[45] THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[46] PATRICK COUSOT. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, March 1978.

[47] PATRICK COUSOT. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In Radhia Cousot, editor of the Proceedings of *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, June 2005.

[48] PATRICK COUSOT. Topics in Abstract Interpretation. `http://www.di.ens.fr/~cousot/AI/`, August 2008.

[49] PATRICK COUSOT, RADHIA COUSOT, JÉRÔME FERET, LAURENT MAUBORGNE, ANTOINE MINÉ, AND XAVIER RIVAL. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[50] KARL CRARY AND STEPHANIE WEIRICH. Resource Bound Certification. In Mark N. Wegman and Thomas W. Reps, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–198. ACM, January 2000.

[51] NORMAN DANNER, JENNIFER PAYKIN, AND JAMES S. ROYER. A static cost analysis for a higher-order language. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors of the Proceedings of *ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, pages 25–34. ACM, January 2013.

[52] JAVIER DE DIOS AND RICARDO PEÑA. Certification of Safe Polynomial Memory Bounds. In Michael Butler and Wolfram Schulte, editors of the Proceedings of *International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 184–199. Springer, June 2011.

[53] LEONARDO MENDONÇA DE MOURA AND NIKOLAJ BJØRNER. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors of the Proceedings of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, April 2008.

[54] SAUMYA K. DEBRAY AND NAI-WEI LIN. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[55] SAUMYA K. DEBRAY, PEDRO LÓPEZ-GARCÍA, MANUEL V. HERMENEGILDO, AND NAI-WEI LIN. Lower Bound Cost Estimation for Logic Programs. In Jan Maluszynski, editor of the Proceedings of *International Logic Programming Symposium (ILPS)*, pages 291–305. MIT Press, October 1997.

[56] XIAO YAN DENG, GREG MICHAELSON, AND PHILIP W. TRINDER. Cost-driven autonomous mobility. *Computer Languages, Systems & Structures*, 36(1):34–59, 2010.

[57] ANDREAS DOLZMANN, ANDREAS SEIDL, AND THOMAS STURM. *Redlog User Manual, Edition 3.1*, November 2006.

[58] PHILIPPE FLAJOLET, BRUNO SALVY, AND PAUL ZIMMERMANN. Automatic Average-Case Analysis of Algorithm. *Theoretical Computer Science*, 79(1):37–109, 1991.

[59] PHILIPPE FLAJOLET AND ROBERT SEDGEWICK. *Analytic Combinatorics*. Cambridge University Press, 2009.

[60] JÜRGEN GIESL, THOMAS STRÖDER, PETER SCHNEIDER-KAMP, FABIAN EMMES, AND CARSTEN FUHS. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In Danny De Schreye, Gerda Janssens, and Andy King, editors of the Proceedings of *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 1–12. ACM, September 2012.

[61] JÜRGEN GIESL, RENÉ THIEMANN, PETER SCHNEIDER-KAMP, AND STEPHAN FALKE. Automated Termination Proofs with AProVE. In Vincent van Oostrom, editor of the Proceedings of *International Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, June 2004.

[62] DANIEL H. GREENE AND DONALD E. KNUTH. *Mathematics for the Analysis of Algorithms*. Birkhäuser, 2008.

[63] BERND GROBAUER. Cost Recurrences for DML Programs. In Benjamin C. Pierce, editor of the Proceedings of *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 253–264. ACM, September 2001.

[64] BHARGAV S. GULAVANI AND SUMIT GULWANI. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In Aarti Gupta and Sharad Malik, editors of the Proceedings of *International Conference on Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, July 2008.

[65] SUMIT GULWANI, SAGAR JAIN, AND ERIC KOSKINEN. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors of the Proceedings of *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 375–385. ACM, June 2009.

[66] SUMIT GULWANI, KRISHNA K. MEHRA, AND TRISHUL M. CHILIMBI. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 127–139. ACM, January 2009.

[67] SUMIT GULWANI AND FLORIAN ZULEGER. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors of the Proceedings of *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 292–304. ACM, June 2010.

[68] NICOLAS HALBWACHS. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Ph.D. Thesis, Institut National Polytechnique de Grenoble, March 1978.

[69] ANTHONY C. HEARN. REDUCE *User's Manual, Version 3.8*, February 2004.

[70] MANUEL V. HERMENEGILDO, ELVIRA ALBERT, PEDRO LÓPEZ-GARCÍA, AND GERMÁN PUEBLA. Abstraction carrying code and resource-awareness. In Pedro Barahona and Amy P. Felty, editors of the Proceedings of *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 1–11. ACM, July 2005.

[71] MANUEL V. HERMENEGILDO, FRANCISCO BUENO, MANUEL CARRO, PEDRO LÓPEZ-GARCÍA, EDISON MERA, JOSÉ F. MORALES, AND GERMÁN

PUEBLA. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[72] MANUEL V. HERMENEGILDO, RICHARD WARREN, AND SAUMYA K. DEBRAY. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

[73] TIMOTHY J. HICKEY AND JACQUES COHEN. Automating program analysis. *Journal of the ACM*, 35(1):185–220, 1988.

[74] C. A. R. HOARE. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[75] JAN HOFFMANN, KLAUS AEHLIG, AND MARTIN HOFMANN. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14, 2012.

[76] JAN HOFFMANN AND ZHONG SHAO. Type-Based Amortized Resource Analysis with Integers and Arrays. In Michael Codish and Eijiro Sumii, editors of the Proceedings of *International Symposium on Functional and Logic Programming (FLOPS)*, volume 8475 of *Lecture Notes in Computer Science*, pages 152–168. Springer, June 2014.

[77] MARTIN HOFMANN AND STEFFEN JOST. Static prediction of heap space usage for first-order functional programs. In Alex Aiken and Greg Morrisett, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[78] MARTIN HOFMANN AND DULMA RODRIGUEZ. Automatic Type Inference for Amortised Heap-Space Analysis. In Matthias Felleisen and Philippa Gardner, editors of the Proceedings of *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 593–613. Springer, March 2013.

[79] RODNEY R. HOWELL. On Asymptotic Notations with Multiple Variables. Technical Report 2007-4, Department of Computing and Information Sciences, Kansas State University, Manhatan, Kansas, January 2008.

[80] JOHN IVIE. Some MACSYMA Programs for Solving Recurrence Relations. *ACM Transactions on Mathematical Software*, 4(1):24–33, 1978.

[81] STEFFEN JOST. Automated Amortised Analysis. Ph.D. Thesis, Ludwig Maximillians Universitat Munich, August 2010.

[82] STEFFEN JOST, KEVIN HAMMOND, HANS-WOLFGANG LOIDL, AND MARTIN HOFMANN. Static determination of quantitative resource usage for higher-order programs. In Manuel V. Hermenegildo and Jens Palsberg, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 223–236. ACM, January 2010.

[83] DEEPAK KAPUR. Automatically Generating Loop Invariants Using Quantifier Elimination. In Franz Baader, Peter Baumgartner, Robert Nieuwenhuis, and Andrei Voronkov, editors of the Proceedings of *Deduction and Applications*, volume 05431 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, October 2006.

[84] RICHARD M. KARP. Probabilistic Recurrence Relations. *Journal of the ACM*, 41(6):1136–1150, 1994.

[85] RICHARD M. KARP, RAYMOND E. MILLER, AND SHMUEL WINOGRAD. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM*, 14(3):563–590, 1967.

[86] TAKUMI KASAI AND AKEO ADACHI. A Characterization of Time Complexity by Simple Loop Programs. *Journal of Computer and System Sciences (JCSS)*, 20(1):1–17, 1980.

[87] OWEN KASER, C. R. RAMAKRISHNAN, AND SHAUNAK PAWAGI. On the Conversion of Indirect to Direct Recursion. *ACM Letters on Programming Languages and Systems*, 2(1-4):151–164, 1993.

[88] STEVE KERRISON, UMER LIQAT, KYRIAKOS GEORGIOU, ALEJANDRO SERRANO, NEVILLE GRECH, PEDRO LÓPEZ-GARCÍA, KERSTIN EDER, AND MANUEL V. HERMENEGILDO. Energy Consumption Analysis of Programs based on XMOS ISA-Level Models. In Gopal Gupta and Ricardo Peña, editors of the Proceedings of *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, September 2013.

[89] DONALD E. KNUTH. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[90] DONALD E. KNUTH. Big Omicron and Big Omega and Big Theta. *SIGACT News*, 8(2):18–24, April 1976.

[91] DANIEL KROENING AND OFER STRICHMAN. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008.

[92] UGO DAL LAGO AND BARBARA PETIT. The geometry of types. In Roberto Giacobazzi and Radhia Cousot, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 167–178. ACM, January 2013.

[93] SENLIN LIANG. Non-termination Analysis and Cost-Based Query Optimization of Logic Programs. In Markus Krötzsch and Umberto Straccia, editors of the Proceedings of *International Conference on Web Reasoning and Rule Systems (RR)*, volume 7497 of *Lecture Notes in Computer Science*, pages 284–290. Springer, September 2012.

[94] TIM LINDHOLM, FRANK YELLIN, GILAD BRACHA, AND ALEX BUCKLEY. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[95] PEDRO LÓPEZ-GARCÍA, LUTHFI DARMAWAN, AND FRANCISCO BUENO. A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. In Manuel V. Hermenegildo and Torsten Schaub, editors of the Proceedings of *International Conference on Logic Programming (ICLP) (Technical Communications)*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, July 2010.

[96] PEDRO LÓPEZ-GARCÍA, MANUEL V. HERMENEGILDO, AND SAUMYA K. DEBRAY. A Methodology for Granularity-Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation*, 21(4):715–734, 1996.

[97] DANIEL LE MÉTAYER. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[98] MANUEL MONTENEGRO, OLHA SHKARAVSKA, MARKO C. J. D. VAN EEKELEN, AND RICARDO PEÑA. Interpolation-Based Height Analysis for Improving a Recurrence Solver. In Ricardo Peña, Marko C. J. D. van Eekelen, and Olha Shkaravska, editors of the Proceedings of *International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, volume 7177 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2012.

[99] JORGE A. NAVAS, MARIO MÉNDEZ-LOJO, AND MANUEL V. HERMENEGILDO. User-Definable Resource Usage Bounds Analysis for Java Bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5):65–82,

2009. Proceedings of Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE).

[100] JORGE A. NAVAS, EDISON MERA, PEDRO LÓPEZ-GARCÍA, AND MANUEL V. HERMENEGILDO. User-Definable Resource Bounds Analysis for Logic Programs. In Verónica Dahl and Ilkka Niemelä, editors of the Proceedings of *International Conference on Logic Programming (ICLP)*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363. Springer, September 2007.

[101] FLEMMING NIELSON, HANNE RIIS NIELSON, AND CHRIS HANKIN. *Principles of Program Analysis*. Springer, 2005.

[102] FLEMMING NIELSON, HANNE RIIS NIELSON, AND HELMUT SEIDL. Automatic Complexity Analysis. In Daniel Le Métayer, editor of the Proceedings of *European Symposium on Programming (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer, April 2002.

[103] LARS NOSCHINSKI, FABIAN EMMES, AND JÜRGEN GIESL. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.

[104] CHRIS OKASAKI. *Purely functional data structures*. Cambridge University Press, 1999.

[105] MARKO PETKOVSEK. Hypergeometric Solutions of Linear Recurrences with Polynomial Coefficents. *Journal of Symbolic Computation*, 14(2/3):243–264, 1992.

[106] MARKO PETKOVSEK, HERBERT WILF, AND DORON ZEILBERGER. *A=B*. A K Peters Ltd., 1996.

[107] LYLE HAROLD RAMSHAW. Formalizing the analysis of algorithms. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1979.

[108] JOHN C. REYNOLDS. An Overview of Separation Logic. In Bertrand Meyer and Jim Woodcock, editors of the Proceedings of *Verified Software: Theories, Tools, and Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer, October 2005.

[109] XAVIER RIVAL AND LAURENT MAUBORGNE. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[110] DULMA RODRIGUEZ. Amortised Resource Analysis for Object-Oriented Programs. Ph.D. Thesis, Ludwig Maximillians Universitat Munich, October 2012.

[111] ENRIC RODRÍGUEZ-CARBONELL AND DEEPAK KAPUR. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[112] OREN PATASHNIK RONALD L. GRAHAM, DONALD E. KNUTH. *Concrete Mathematics*. Addison-Weasley, 2nd edition, 1994.

[113] MADS ROSENDAHL. Automatic Complexity Analysis. In Proceedings of *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 144–156, 1989.

[114] BRUNO SALVY AND PAUL ZIMMERMANN. GFUN: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

[115] SRIRAM SANKARANARAYANAN, FRANJO IVANCIC, ILYA SHLYAKHTER, AND AARTI GUPTA. Static Analysis in Disjunctive Numerical Domains. In Kwangkeun Yi, editor of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, August 2006.

[116] SRIRAM SANKARANARAYANAN, HENNY SIPMA, AND ZOHAR MANNA. Non-linear loop invariant generation using Gröbner bases. In Neil D. Jones and Xavier Leroy, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 318–329. ACM, January 2004.

[117] SRIRAM SANKARANARAYANAN, HENNY B. SIPMA, AND ZOHAR MANNA. Constructing invariants for hybrid systems. *Formal Methods in System Design*, 32(1):25–55, 2008.

[118] GABRIEL SCHERER AND JAN HOFFMANN. Tracking Data-Flow with Open Closure Types. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors of the Proceedings of *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *Lecture Notes in Computer Science*, pages 710–726. Springer, December 2013.

[119] ROBERT SEDGEWICK AND PHILIPPE FLAJOLET. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2nd edition, January 2013.

[120] ALEJANDRO SERRANO, PEDRO LÓPEZ-GARCÍA, FRANCISCO BUENO, AND MANUEL V. HERMENEGILDO. Sized Type Analysis for Logic Programs. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013.

[121] ALEJANDRO SERRANO, PEDRO LÓPEZ-GARCÍA, AND MANUEL V. HERMENEGILDO. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming*, 14(4-5):739–754, 2014.

[122] OLHA SHKARAVSKA AND MARKO C. J. D. VAN EEKELEN. Univariate polynomial solutions of algebraic difference equations. *Journal of Symbolic Computation*, 60:15–28, 2014.

[123] OLHA SHKARAVSKA, MARKO C. J. D. VAN EEKELEN, AND RON VAN KESTEREN. Polynomial Size Analysis of First-Order Shapely Functions. *Logical Methods in Computer Science*, 5(2), 2009.

[124] HUGO R. SIMÕES, PEDRO B. VASCONCELOS, MÁRIO FLORIDO, STEFFEN JOST, AND KEVIN HAMMOND. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In Peter Thiemann and Robby Bruce Findler, editors of the Proceedings of *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 165–176. ACM, September 2012.

[125] FAUSTO SPOTO, FRED MESNARD, AND ÉTIENNE PAYET. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[126] SAURABH SRIVASTAVA, SUMIT GULWANI, AND JEFFREY S. FOSTER. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.

[127] BJARNE STEENSGAARD. Points-to Analysis in Almost Linear Time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41. ACM, January 1996.

[128] DAVID R. STOUTEMYER. Automatic Asymptotic and Big-O Calculations Via Computer Algebra. *SIAM Journal on Computing*, 8(3):287–299, 1979.

[129] THOMAS STURM AND ASHISH TIWARI. Verification and synthesis using real quantifier elimination. In Éric Schost and Ioannis Z. Emiris, editors

of the Proceedings of *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 329–336. ACM, June 2011.

[130] ANKUR TALY, SUMIT GULWANI, AND ASHISH TIWARI. Synthesizing switching logic using constraint solving. *International Journal on Software Tools for Technology Transfer*, 13(6):519–535, 2011.

[131] ROBERT ENDRE TARJAN. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[132] PEDRO VASCONCELOS. Space Cost Analysis using Sized Types. Ph.D. Thesis, University of St Andrews, August 2008.

[133] PEDRO B. VASCONCELOS AND KEVIN HAMMOND. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In Philip W. Trinder, Greg Michaelson, and Ricardo Peña, editors of the Proceedings of *Implementation of Functional Languages (IFL)*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, September 2004.

[134] PHILIP WADLER. Strictness Analysis Aids Time Analysis. In Jeanne Ferrante and P. Mager, editors of the Proceedings of *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM, January 1988.

[135] BEN WEGBREIT. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[136] BEN WEGBREIT. Verifying Program Performance. *Journal of the ACM*, 23(4):691–699, 1976.

[137] BEN WEGBREIT. Constructive Methods in Program Verification. *IEEE Transactions on Software Engineering (TSE)*, 3(3):193–209, 1977.

[138] TAO WEI, JIAN MAO, WEI ZOU, AND YU CHEN. A New Algorithm for Identifying Loops in Decompilation. In Hanne Riis Nielson and Gilberto Filé, editors of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer, August 2007.

[139] REINHARD WILHELM, JAKOB ENGBLOM, ANDREAS ERMEDAHL, NIKLAS HOLSTI, STEPHAN THESING, DAVID B. WHALLEY, GUILLEM BERNAT, CHRISTIAN FERDINAND, REINHOLD HECKMANN, TULIKA MITRA, FRANK MUELLER, ISABELLE PUAUT, PETER P. PUSCHNER, JAN STASCHULAT, AND

PER STENSTRÖM. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3), 2008.

[140] TING YU AND OWEN KASER. A Note on "On the Conversion of Indirect to Direct Recursion". *ACM Transactions on Programming Languages and Systems*, 19(6):1085–1087, 1997.

[141] FLORIAN ZULEGER, SUMIT GULWANI, MORITZ SINN, AND HELMUT VEITH. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In Eran Yahav, editor of the Proceedings of *International Static Analysis Symposium (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer, September 2011.

This document was compiled on Thursday 4th September, 2014, using version 3.1415926-2.5-1.40.14 of TEX, and the packages of the *TeX Live 2013/Debian* LATEX distribution.

The main text is formatted using the Utopia serif typeface, and `Inconsolata` was used for code listings.