FACULTAD DE INFORMTICA

UNIVERSIDAD POLITCNICA DE MADRID

Termination and Cost Analysis: Complexity and Precision Issues

PhD Thesis

Md. Abu Naser Masud

November 2012

PhD Thesis

Termination and Cost Analysis: Complexity and Precision Issues

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment of the degree of
PhD in Research into Technologies for the
Development of Complex Software Systems

PhD Candidate: **Md. Abu Naser Masud**

Computer Science Engineer
Master in Computational Logic
Technical University of Madrid, España
&
Dresden University of Technology, Germany

Advisor: **Germán Puebla**
Associate Professor
Technical University of Madrid

Advisor: **Samir Genaim**
Professor Contratado Doctor
Complutense University of Madrid

Madrid, November 2012

Sinopsis

La investigaciones llevadas a cabo en esta tesis se centran en el análisis estático de coste y el análisis de terminación. Mientras que el objetivo del análisis de coste es estimar la cantidad de recursos consumida por un programa durante su ejecución, el análisis de terminación se centra en garantizar que la ejecución de un programa terminará en un tiempo finito. Sin embargo, ambos análisis se encuentran estrechamente relacionados, de hecho, muchas de las técnicas utilizadas para el análisis de coste se basan en técnicas desarrolladas inicialmente para el análisis de terminación.

La precisión, la escalabilidad y la aplicabilidad son aspectos clave para cualquier análisis estático: un aumento de precisión mejora la calidad de la información inferida por el análisis; la escalabilidad del mismo hace referencia a la capacidad de analizar programas de mayor tamaño; y la aplicabilidad a la clase de programas que se pueden analizar satisfactoriamente (independientemente de la precisión y escalabilidad). Esta tesis aborda todos estos aspectos en el contexto del análisis de coste y de terminación, haciéndolo tanto desde una perspectiva teórica como práctica.

Con respecto al análisis de coste, esta tesis aborda el problema de, dado un sistema de relaciones de coste (una forma de relaciones de recurrencia), resolver estas relaciones y expresarlas en forma de funciones de coste en forma cerrada, permitiendo establecer tanto las cotas superiores como inferiores del consumo de recursos del programa. Este problema es crucial para la mayora de los analizadores de coste modernos, y en él radican muchas de las limitaciones de precisión y aplicabilidad de los análisis. En esta tesis se desarrollan y detallan los fundamentos teóricos de nuevas técnicas para la resolución de relaciones de coste, venciéndose las limitaciones de trabajos anteriores, y resultando en un aumento tanto de la precisión obtenida, como en una mejora en la escalabilidad de los análisis. Una caracterstica única de las técnicas descritas en esta tesis es la de poder inferir tanto cotas superiores como cotas inferiores, solo con invertir las nociones correspondientes en la teoría subyacente.

En lo que respecta al análisis de terminación, nuestro trabajo se centra en el

estudio de la dificultad de decidir sobre la terminación de cierto tipo de bucles sencillos que aparecen en el contexto del análisis de coste. Este estudio nos ayuda a esclarecer los límites teóricos de la aplicabilidad y de la precisión tanto de los análisis de coste como de los análisis de terminación.

Palabras clave: Análisis estático de programas, Análisis de Coste, Análisis de Terminación, Ecuaciones de Recurrencia, Complejidad Computacional.

Abstract

The research in this thesis is related to static cost and termination analysis. Cost analysis aims at estimating the amount of resources that a given program consumes during the execution, and termination analysis aims at proving that the execution of a given program will eventually terminate. These analyses are strongly related, indeed cost analysis techniques heavily rely on techniques developed for termination analysis. Precision, scalability, and applicability are essential in static analysis in general. Precision is related to the quality of the inferred results, scalability to the size of programs that can be analyzed, and applicability to the class of programs that can be handled by the analysis (independently from precision and scalability issues). This thesis addresses these aspects in the context of cost and termination analysis, from both practical and theoretical perspectives.

For cost analysis, we concentrate on the problem of solving cost relations (a form of recurrence relations) into closed-form upper and lower bounds, which is the heart of most modern cost analyzers, and also where most of the precision and applicability limitations can be found. We develop tools, and their underlying theoretical foundations, for solving cost relations that overcome the limitations of existing approaches, and demonstrate superiority in both precision and applicability. A unique feature of our techniques is the ability to smoothly handle both lower and upper bounds, by reversing the corresponding notions in the underlying theory. For termination analysis, we study the hardness of the problem of deciding termination for a specific form of simple loops that arise in the context of cost analysis. This study gives a better understanding of the (theoretical) limits of scalability and applicability for both termination and cost analysis.

Keywords: Static analysis, Cost analysis, Termination analysis, Recurrence equations, Complexity.

Acknowledgments

I would like to express my heartiest gratitude to those without whom this thesis would never be accomplished. First and foremost, I would like to thank my thesis director Samir Genaim whose continuous support and inspiration keep me motivated in continuing this research. Samir is one of the great teachers I have met in my life who teaches me the fundamental things about doing research. He would reshape my thinking, my way of looking into a problem and the directions of finding solutions. He teaches me the ethics in doing research. Overall, I think that all those teachings will have a great influence in my future life.

I would like to thank Germán Puebla who gave me the opportunity to enroll in the PhD program and initiated in developing my research plan. I am also very much thankful to both Elvira Albert and Germán Puebla who gave me supports not only in my PhD studies but also in my personal life in many respects throughout my studies. I would also like to thank Amir M. Ben-Amram with whom I have publications and part of my thesis is the results of joint-work with him.

I would like to thank Guillermo Román Díez who has been one of my good friends and helped me a lot in resolving lengthy spanish bureaucratic matters. I would also like to thank Diana Ramirez who gave me answers to questions on COSTA implementations with patience. I would like to thank the whole COSTA team.

I wish to thank my wife Syeda Shahrin Moudud who had very difficult situations but kept her patience and gave me inspirations and rooms for work. I would like to remember my one and half years old loveliest daughter Farha who gave me a lot of pleasure during this time. Finally, I have my sincere thanks to my parents who have devoted their lives for all the goods of my life. I also would like to thank all of my family members.

Last, but not least, I would like to acknowledge the financial support without which this thesis would have not existed. This work was funded in part by the Information & Communication Technologies program of the European Commission,

Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

!!!THANK YOU ALL!!!

Contents

List of Figures

Chapter 1

Introduction

Static program analysis aims at inferring runtime properties of a given program, written in some programming language, without actually executing it. This includes non-quantitative properties, e.g., the value of a given variable lies in the interval [1..10], or quantitative properties, e.g., the program does not consume more than 100 memory units. Such properties are mainly used to prove, statically, that programs do not reach erroneous states, or that they meet their corresponding specifications. The research in this thesis is related to statically analyzing the resource consumption (a.k.a. cost) and termination behavior of programs. Cost and termination analysis are strongly related topics, indeed much of the research in cost analysis in the last decade has benefited from techniques developed in the context of termination analysis.

The research presented in this thesis has both practical and theoretical aspects. On the one hand it develops techniques in order to achieve practical, precise and widely applicable resource usage analysis, and on the other hand it studies the theoretical complexity of some termination analysis problems that arise in the context of cost analysis. The rest of this chapter overviews the objectives (Section 1.1), contributions (Section 1.2), and organization (Section 1.3) of this thesis.

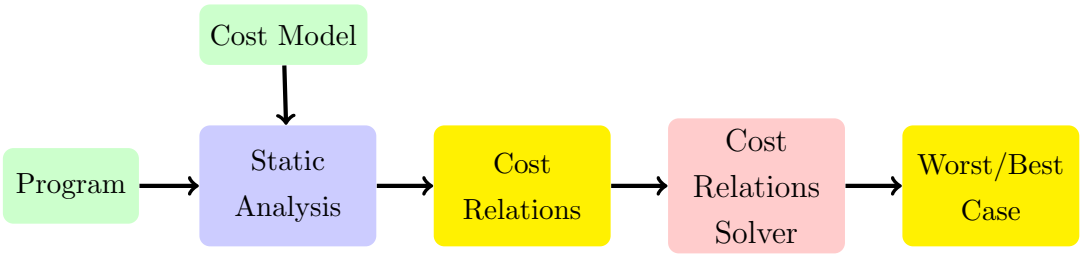


Figure 1.1: Phases of the classical approach to cost analysis.

1.1 Objectives

Having available information about the computational cost of programs execution, i.e., the amount of resources that the execution will require, is clearly useful for many different purposes, like for performance debugging, resource usage verification/certification and for program optimization (see, e.g., [8]). In general, reasoning about execution cost is difficult and error-prone. This is specially the case when programs contain *loops* (either as iterative constructs or recursions), since one needs to reason about the number of iterations that loops will perform and the cost of each of them.

Static cost analysis aims at automatically inferring the resource consumption (or cost) of executing a program as a function of its input data sizes. The classical approach to cost analysis by Wegbreit dates back to 1975 [80], which consists of two phases as depicted in Figure 1.1. In the *first phase*, given a program and a *cost model*, the analysis produces *cost relations*, i.e., a system of recursive equations, which capture the cost of the program in terms of (the size of) its input data. A set of recursive equations can be seen as a constraints logic program (over the integers domain), such that when executing it on a given (abstraction of the) input it computes the cost of executing the corresponding source program on that input. Thus, at this point, we still need some form of execution, although simpler, in order to estimate the cost of executing the source program on a given input. In order to get closer to fully static estimation of the program's cost, in a *second phase*, these recursive equations are solved into closed-form upper and lower bounds, i.e., to nonrecursive functions which can be evaluated on a given

(abstraction of the) input to estimate the cost of executing the source program on that input. Note that *evaluation* here does not involve any form of execution, it is just an evaluation of simple mathematical expressions. It is worth mentioning that the role of the *cost model* in Figure 1.1 is to specify the resource we are measuring; cost models widely used are the number of executed instructions, number of calls to selected methods, amount of memory allocated, etc.

In general, the first phase of cost analysis (i.e., the process of generating cost relations from the program) heavily depends on the programming language in which the program is written. In principle, it consists of applying several static analyses in order to understand the control and data flow in the program, e.g., how the data changes when a given loop goes from one iteration to another, which is later used to bound the number of iterations of the loop. Multiple analysis have been developed for different paradigms including for functional [80, 54, 70, 79, 72, 20, 58, 45], logic [33, 66], and imperative [1, 8] programming languages. Importantly, the resulting cost relations are a common target of cost analyzers, i.e., they abstract away the particular features of the original programming language and, at least conceptually, have the same form.

In the second phase of cost analysis, closed-form bounds, in terms of the input arguments, for the cost relations generated in first phase are computed. Computing such closed-form bound is challenging as the cost relations are usually recursive equations (because the source program contains loops and/or recursive methods) and hence computing such bounds require, for example, computing the number of iterations of the recursive equations. This becomes even more complicated in the presence of nondeterminism in cost relations (due to abstractions in the first phase), since we have to account for all possible combinations. Moreover, it is not always possible to compute an exact bound because of this nondeterministic behavior of cost relations and hence analyzers try to infer closed-form upper and lower bounds for such cost relation, which correspond, respectively, to the worst-case and best-case costs.

Needless to say, precision is fundamental for most applications of cost analysis. For instance, upper bounds are widely used to estimate the space and time requirements of programs execution and provide resource guarantees [32]. Lack of precision in such case can make the system fail to prove the resource usage

requirements imposed by the software client. For example, it even makes much difference inferring the upper bound $\frac{1}{2}n^2$ instead of n^2 for a given method (where n is an input integer value for example). With the latter, an execution with n=10 will be rejected if we have only 50 units of the corresponding resource, while with the former one it is accepted. Precision is also important for lower bounds, for example, when they are used for scheduling tasks in parallel execution in such a way that it is not worth parallelizing a task unless its lower-bound resource consumption is sufficiently large. Precision will be essential here to achieve a satisfactory scheduling.

Precision issues in cost analysis can be divided into two categories: (i) precision issues related to the static analyses applied in the first phase; and (ii) precision issues related to the process of solving cost relations into closed-form bounds in the second phase. The former category is not particular to cost analysis, but rather a well-know issue in static analysis in general. It is common to assume that precision issues in this category can, at least conceptually, be solved using more expressive abstract domains, which might also come with some performance overhead. The later category is very particular to cost analysis; existing techniques for solving cost relations are based on fundamentally different concepts, and thus the precision issues of each technique are affected by conceptually different parameters. In addition, in this category, the notion of applicability is also very important. This notion characterizes the set of cost relations that can be handled by such solvers, independently of how precise the inferred bounds are. The *first objective* of this thesis is to study precision and applicability limitations of existing techniques for solving cost relations into upper and lower bounds, and, develop new techniques that overcome these limitations and thus increase both precision and applicability of cost analysis in general.

Achieving the first objective of this thesis results in techniques that leads to precise and widely applicable cost analysis. However, it does not provide any insight on the theoretical (or practical) limits of cost analysis in general. In particular, one might be interested to know the degree of solvability of inferring resource bounds for some class of programs. This leads to a better understanding of the theoretical and practical limits of scalability and applicability when solving cost relations. Note that understanding the limits set by inherent undecidability

or intractability of problems yields more profound information than evaluating the performance of one particular algorithm.

The techniques developed in this thesis in order to achieve the first objective, as well as other related techniques [5], heavily rely on problems from the field of termination analysis. In particular, on the problem of bounding the number of iterations that a loop can make and thus proving its termination. In fact, proving termination of a given program can be seen as a special case of cost analysis, where in every iteration of the program's loops (or recursions) the contributed cost is 1, and non-iterative program instructions contribute cost 0. Due to this relation, the theoretical and practical limits of proving termination, for a given class of programs, give us some insights on the corresponding limits of cost analysis for the same class of programs. For example, the undecidability of inferring cost bounds can be reduced from the undecidability of the halting problem for Turing equivalent computer programs. Since cost analysis is an inherently complex task, in this thesis we restrict our interest to limitations that are inherited from the corresponding termination problems. In particular, the *second objective* of this thesis is to study the hardness of proving termination of some form of loops that arise in the context of solving cost relations.

1.2 Summary of Contributions

The contributions of this thesis can be divided into two categories, following the two objectives described in the previous section. As for the first objective, our main contributions are:

1. We developed novel techniques for inferring upper bounds for cost relations. These techniques demonstrate superiority on previous techniques with respect to precision, and, at the same time, they are still widely applicable. The techniques have been formalized and their soundness has been proven.

2. We extended our previous techniques for the case of inferring precise lower bounds as well. The techniques have been formalized, using dual arguments to those used for the upper bounds case, and their soundness has

been proven. Our techniques can obtain nontrivial lower bounds, which are also very precise, and still widely applicable when compared to previous approaches.

3. We have implemented our techniques in PUBS (Practical Upper Bound Solver) [5], which is also used as backend solver in COSTA (a COSt and Termination Analyzer for Java bytecode) [5]. We have experimentally evaluated them on cost relations obtained from a selected set of Java (bytecode) programs.

The above contributions have been published in the following research papers:

- Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer, January 2011.

- Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*. **To Appear**.

As for the second objective, we have investigated the complexity of deciding termination for some variations of simple integer loops. Our main contributions are the following:

1. For some variations we have proved that it is undecidable, despite of their simple form.

2. For some other variations we provided lower bounds on the complexity.

The above contributions have been published in the following research papers:

- Amir M. Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. In Viktor Kuncak and Andrey Rybalchenko,

editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, USA, January 25-27, 2012. Proceedings*, Lecture Notes in Computer Science. Springer, January 2012.

- Amir M. Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. *ACM Transactions on Programming Langauges and Systems*. **To Appear**.

1.3 Organization

Our research was performed between September 2009 and September 2012, and is presented in this thesis in three parts.

Part I

This part presents the research related to the first objective, which deals with developing sound and precise techniques for inferring lower and upper bounds for cost relations. This part is organized as follows

- In Chapter 2, we describe the problems and challenges for inferring lower and upper bounds for cost relations, provide an informal but enough intuitive details, and discuss the overview of the corresponding contributions.
- In Chapter 3, we define some mathematical notations that we use throughout Part I of this thesis.
- In Chapter 4, we develop our techniques for inferring precise upper bounds for cost relations.
- In Chapter 5, we generalize the techniques of Chapter 4 for the case of lower bounds.
- In Chapter 6, we describe our implementation and a corresponding experimental evaluation.

Part II

This part presents the research related to the second objective, which deals with deciding termination of some variations of integer loops. This part is organized as follows

- In Chapter 7, we describe the problems and challenges for deciding termination of integer loops, and provide an informal overview of the contributions.
- In Chapter 8, we define several variations of integer loops, whose termination we are interested in, and recall some definitions and mathematical background required in this part of the thesis.
- In Chapter 9, we study the complexity of deciding termination of several variations of integer loops.

Part III

This part includes an overview of related research, in Chapter 10, and conclusions and possible future extensions of our work, In Chapter 11.

Part I

Inference of Precise Upper and Lower Bounds for Cost Relations

Chapter 2

Overview of the Problems, Challenges, and Contributions

In this chapter we overview the problems and challenges for solving cost relations into closed form lower and upper bounds, informally present our proposed solutions, and detail the organization of part I of the thesis.

2.1 Problems and Challenges

The classical approach to cost analysis [80] consists of two phases. In the first phase, given a program and a *cost model*, the analysis produces *cost relations* (*CRs* for short), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. In the second phase, these *CRs* are solved into closed-form bounds. Part I of this thesis focuses on the second phase of cost analysis, i.e., in developing widely applicable techniques for precisely solving *CRs* into closed-form lower and upper bounds (LBs and UBs for short).

The following example illustrates informally how high-level programs are translated to *CRs*, and also the syntax and semantic of *CRs*.

EXAMPLE 2.1.1. *Let us consider the Java program depicted in Figure 2.1, which we will be using as a running example throughout the first part of the*

```
1 void fun_heapConsume(int q) {
2   List l = null;
3   int i=0;
4   while (i < q) {
5     int j=0;
6     while ( j < i ) {
7       for(int k=0; k < q+j; k++)
8         l=new List(i*k*j,l);
9       j=j+random() ? 1:3;
10    }
11    i=i+random() ? 2:4;
12  }
13 }
```

Figure 2.1: Running Example

thesis. It is sufficiently simple in order to explain the main technical parts, but still interesting to understand the challenges and our achievements. For this program and the memory consumption *cost model, the cost analysis of [8] generates the CR which appears in Figure 2.2. This cost model estimates the number of objects allocated in the memory. Note that in this thesis we ignore the effect that compiler optimizations might have on the resource consumption; handling this is out of the scope of this thesis. Observe that the structure of the Java program and its corresponding CR match. The equations for C correspond to the* `for` *loop, those of B to the inner* `while` *loop and those of A to the outer* `while` *loop. The recursive equation for C states that the memory consumption of executing the inner loop with $\langle k, j, q\rangle$ such that $k+1 \leq q+j$ is 1 (one object) plus that of executing the loop with $\langle k', j, q\rangle$ where $k' = k+1$. The recursive equation for B states that executing the loop with $\langle j, i, q\rangle$ costs as executing $C(0, j, q)$ plus executing the same loop with $\langle j', i, q\rangle$ where $j+1 \leq j' \leq j+3$. While, in the Java program, j' can be either $j+1$ or $j+3$, due to the static analysis, the case for $j+2$ is added in order to over approximate $j' = j+1 \vee j' = j+3$ by the polyhedron $j+1 \leq j' \leq j+3$ [31].*

Though *CRs* are simpler than the programs they originate from, in several

$F(q)$	$= A(0,q)$	$\{\}$
$A(i,q)$	$= 0$	$\{i \geq q\}$
$A(i,q)$	$= B(0,i,q) + A(i',q)$	$\{i+1 \leq q, i+2 \leq i' \leq i+4\}$
$B(j,i,q)$	$= 0$	$\{j \geq i\}$
$B(j,i,q)$	$= C(0,j,q) + B(j',i,q)$	$\{j+1 \leq i, j+1 \leq j' \leq j+3\}$
$C(k,j,q)$	$= 0$	$\{k \geq q+j\}$
$C(k,j,q)$	$= 1 + C(k',j,q)$	$\{k' = k+1, k+1 \leq q+j\}$

Figure 2.2: *CRs* for the program of Figure 2.1

respects they are not as static as one would expect from the result of a static analysis. One reason is that they are recursive and thus we may need to iterate for computing their value for concrete input values. Another reason is that even for deterministic programs, the loss of precision introduced by the abstraction of data structures and arrays may result in *CRs* which are nondeterministic. This is because arrays are abstracted to their length, and data structures to their depth. Thus, when generating *CRs*, instructions accessing array elements or elements of data structures are abstracted to *true* in most cases and hence making the corresponding equations not mutually exclusive. As an example, if we have the instruction "**if** (A[i]>A[j]) A **else** B" where A and B are sequences of instructions, its abstraction generates two not mutually exclusive equations, one that includes A and the other includes B. In our example the nondeterminism happens, e.g., because j' can be either $j+1$, $j+2$ or $j+3$, and they become nondeterministic choices when applying the second equation defining B. In general, for finding the worst-case and best-case cost we may need to compute and compare (infinitely) many results. For both reasons, it is clearly essential to compute *closed-form* bounds for the *CR*, i.e., bounds which are not in recursive form.

Two main approaches exist for automatically solving *CRs* into closed-form

bounds:

1. Since *CRs* are syntactically quite close to standard (linear) *recurrence relations* (*RRs* for short), most cost analysis frameworks rely on existing *Computer Algebra Systems* (*CAS* for short) for finding closed-form functions. The main problem of this approach is that *CAS* only accept as input a small subset of *CRs* which have a single argument, a single recursive equation, and a single base-case equation. This seldom happens. Thus, *CAS* are very precise when applicable, but handle only a restricted class of *CRs*.

2. Instead of the previous approach, specific UB solvers developed for *CRs* try to reason on the worst-case cost and obtain sound UBs of the resource consumption. This is the approach taken in [5]. As regards LBs, due in part to the difficulty of inferring under-approximations, general solvers for *CRs* which are able to obtain useful approximations of the best-case cost have not yet been developed.

Let us see the application of the above approaches to our running example. As regards 1, note that, for example, in the *CR* B, variable j' can increase by one, two or three at each iteration. Therefore, an exact cost function which captures the cost of any possible execution does not exist. Thus, we cannot use *CAS* since an exact solution does not exist. As regards 2, since the cost accumulated in *CR* B varies from one iteration to another (because the value of $C(0, j, q)$ depends on j), this approach assumes the *same* worst-case cost for all iterations which results in a lose of precision as we explain in the next section. Similar precision loss happens when solving *CR* A.

Our challenge is to develop novel techniques for solving *CRs* into closed-form bounds that overcome the limitations of the two approaches described above, but at the same time take advantage of the underlying theory developed in the corresponding contexts. Importantly, and unlike previous approaches, we want that the developed techniques work for inferring both UBs and LBs.

2.2 Informal Overview of the Contributions

In this section we discuss our contributions by explaining the basics of our approach for solving *CRs* into closed-form bounds (both UBs and LBs) and compare it to previous approaches. In particular, we compare to [5] since we heavily rely on some of their techniques. For this, we use a very simple *CR*, instead of the one in Figure 2.2 since it requires further knowledge that is not available to the reader yet.

Consider a *CR* in its simplest form with one base-case equation and one recursive equation with a single recursive call:

$$\langle C(x) = 0, \{x < 0\}\rangle$$
$$\langle C(x) = \|x\| + C(x'), \{x - 3 \leq x' \leq x - 1, x \geq 0\}\rangle$$

An evaluation for $C(x_0)$ (where x_0 denotes the initial value of x) might invoke (if $x_0 \geq 0$) a recursive call $C(x_1)$ with $x_0 - 3 \leq x_1 \leq x_0 - 1$, which in turn might invoke $C(x_2)$ with $x_1 - 3 \leq x_2 \leq x_1 - 1$, etc. Assume that the last recursive call is $C(x_\kappa)$ and that it is solved using the base-case equation (i.e., $x_\kappa < 0$). Note that the recursive equation is applied κ times. In each of these invocations, except the last one, the recursive equation is applied and we accumulate $e_i = \|x_{i-1}\|$ to the cost (here $\|v\| = \max(0, v)$, but it is not important at this point). Thus, the total cost is $e_1 + \cdots + e_\kappa$.

Clearly, $C(x_0)$ has many possible evaluations, depending on the choice of x' in each recursive call, which also determines the number of times we apply the recursive equation, and thus the number of e_i expressions and their values. This means that different evaluations might also have different costs. Our challenge is to accurately estimate the cost of C for any input, i.e., to infer a function $C^{ub}(x_0)$ (resp. $C^{lb}(x_0)$) such that $C^{ub}(x_0)$ is larger (resp. $C^{lb}(x_0)$ is smaller) than the cost $e_1 + \cdots + e_\kappa$ of any possible evaluation.

CAS aim at obtaining the exact cost function, and thus it is not possible to apply it to the above example since $C(x_0)$ has multiple solutions (one for each possible evaluation). Instead, the goal of static cost analysis is to infer approximations in terms of closed-form LBs and UBs for C. Our starting point is the general approximation for UBs proposed by [5] which is based on the following

two dimensions:

1. *Number of applications of the recursive case:* The first dimension is to infer an UB $\hat{\kappa}$ on the number of times the recursive equation can be applied (which, for loops, corresponds to the number of iterations); and

2. *Cost of applications:* The second dimension is to infer an UB $\hat{e}$ on the cost of all loop iterations, i.e., $\hat{e} \geq e_i$ for all i.

For the above example it infers $\hat{\kappa} = \|x_0 + 1\|$ and $\hat{e} = \|x_0\|$. Then, $C^{ub}(x_0) = \hat{\kappa} * \hat{e} = \|x_0\| * \|x_0 + 1\|$ is guaranteed to be an UB for C. Note that if the relation C had two recursive calls, then the UB would be an exponential function of the form $2^{\hat{\kappa}} * \hat{e}$. The most important point to notice is that the cost of all iterations e_i is approximated by the same worst-case cost $\hat{e}$, which is the source of imprecision of [5] that we will improve on. Technically, [5] solves the above two dimensions using programming language techniques (see Section 3.1), which makes it widely applicable in practice.

Our challenge is to improve the precision of [5] while still keeping a similar applicability for UBs and, besides, be able to apply our approach to infer useful LBs. The fundamental idea is to generate a sequence of non-negative elements $\langle u_1, \ldots, u_{\hat{\kappa}} \rangle$, with $\hat{\kappa} \geq \kappa$, such that for any concrete evaluation $\langle e_1, \ldots, e_\kappa \rangle$, each e_i has a corresponding *different* u_j satisfying $u_j \geq e_i$ (observe that the subindexes do not match as $\hat{\kappa} \geq \kappa$). This guarantees soundness since $u_1 + \cdots + u_{\hat{\kappa}}$ is an UB of $e_1 + \cdots + e_\kappa$. Moreover, it is potentially more precise than [5] since the u_i's are not required to be all equal. For the above example, we generate the sequence $\langle \|x_0\|, \|x_0 - 1\|, \ldots, 0 \rangle$. This allows inferring the UB $\frac{\|x_0\| * \|x_0+1\|}{2}$ which is more precise than that of [5] shown before.

Technically, we compute the approximation by transforming the *CR* into a (worst-case) *RR* whose closed-form solution is $u_1 + \cdots + u_{\hat{\kappa}}$. When e is a simple linear expression such as $e \equiv \|l\|$, the novel idea is to view $u_1, \cdots, u_{\hat{\kappa}}$ as an arithmetic sequence (can be geometric or any other sequence) that starts from $u_{\hat{\kappa}} \equiv \hat{e}$ and each time decreases by $\check{d}$ where $\check{d}$ is an under-approximation of all $d_i = e_{i+1} - e_i$, i.e., $u_i = u_{i-1} + \check{d}$. When e is a complex non-linear expression, e.g., $\|l\| * \|l'\|$, it cannot be precisely approximated using sequences. For such cases,

our novel contribution is a method for approximating e by approximating its $\|.\|$ sub-expressions (which are linear) separately.

An important advantage of our approach w.r.t. previous ones [5, 42, 45], is that the problem of inferring LBs is dual. In particular, we can infer a LB $\check{\kappa}$ on the length of chains of recursive calls, the minimum value $\check{e}$ to which e can be evaluated, and then sum the sequence $\langle \ell_1, \ldots, \ell_{\check{\kappa}} \rangle$ where $\ell_i = \ell_{i-1} + \check{d}$ and $\ell_1 = \check{e}$. For the above example, we have $\check{e} = 0$, $\check{d} = 1$ and $\check{\kappa} = \|\frac{x_0+1}{3}\|$ and thus the LB we infer is: $C^{lb}(x_0) = \frac{1}{2} * \|\frac{x_0+1}{3}\| * (\|\frac{x_0+1}{3}\| + 1)$. In addition, our techniques can be applied to cost expressions with any progression behavior that can be modeled using sequences, and not only a linear progression behavior.

In summary, the main achievement in this part of the thesis is a seamless and not-trivial integration of two approaches of solving cost relations, so that we get the best of both worlds: precision as the one based on solving recurrence relations, whenever possible, while applicability as close to the approach of [5]. Technically, the main contributions are:

- We propose an automatic transformation from a *CR* with multiple arguments and a single recursive equation, which possibly accumulates a non-constant cost at each application, into a worst-case/best-case single-argument *RR* that can be solved using *CAS*. Soundness of the transformation requires that we are able to infer the so-called *progression parameters*, which describe the relation between the contributions (to the total cost) of two consecutive applications of the recursive equations.

- As a further step, we consider *CRs* in which we have several recursive equations defining the same relation. We propose an automatic transformation of these *CRs* into a worst-case/best-case *RR* that can be solved using existing *CAS*.

- As another contribution, we present a technique for inferring LBs on the number of iterations, which has similarities with that of [51]. Then, the problem of inferring LBs on the cost becomes dual to the UBs, with some additional conditions for soundness.

- We report on a prototype implementation within the COSTA system [7]. Preliminary experiments on Java (bytecode) programs confirm the good balance between the accuracy and applicability of our analysis.

To the best of our knowledge, this is the first general approach to inferring LBs from *CRs* and, as regards UBs, the one that achieves a better precision vs. applicability balance.

2.3 Organization

The rest of part I of the thesis is organized as follows. Chapter 3 recalls some preliminary notions and introduces some more notations. It formalizes the notion of *cost relation* and *single-argument recurrence relation*. Chapter 4 presents the main technical details of this part of the thesis, which describes how to transform a *CR* into a *RR* for the sake of inferring UBs. Chapter 5 presents the dual problem of inferring LBs from the *CRs*. The main focus of this section is then on obtaining such LBs on loop iterations. Given such bounds, the techniques proposed in Chapter 4 dually apply to the automatic inference of LBs from *CRs*. Chapter 6 describes the implementation of our approach and evaluates it on a series of benchmarks programs that contain loops whose cost in not constant, e.g., sorting algorithms. In these cases, the fact that we accurately approximate the cost of each loop iterations is reflected in the more precise UB that we can obtain. Each chapter ends with some concluding remarks.

Chapter 3

Background on Cost and Recurrence Relations

In this chapter, we fix some notation and recall preliminary definitions. The sets of integer, rational, non-negative integer, and non-negative rational values are denoted respectively by $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_+$, and $\mathbb{Q}_+$. A *linear expression* over $\mathbb{Z}$ has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Q}$, and $x_1, \ldots, x_n$ are variables that range over $\mathbb{Z}$. A *linear constraint* over $\mathbb{Z}$ has the form $l_1 \leq l_2$, where l_1 and l_2 are linear expressions. We use $l_1 = l_2$ as an abbreviation for $l_1 \leq l_2 \wedge l_2 \leq l_1$. We use $\bar{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, and $vars(t)$ to refer to the set of variables that appear syntactically in an entity t. We use φ, ψ and Ψ (possibly subscripted and/or superscripted) to denote (conjunctions of) linear constraints. A set of linear constraints $\{\varphi_1, \ldots, \varphi_n\}$ denotes the conjunction $\varphi_1 \wedge \cdots \wedge \varphi_n$. A solution for φ is an assignment $\sigma : vars(\varphi) \mapsto \mathbb{Z}$ for which φ is satisfiable. The set of all solutions (assignments) of φ is denoted by $[\![\varphi]\!]$. We use $\varphi_1 \models \varphi_2$ to indicate that $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. We use $\sigma(t)$ or $t\sigma$ to bind each $x \in vars(t)$ to $\sigma(x)$, $\exists\bar{x}.\varphi$ for the elimination of the variables $\bar{x}$ from φ, and $\bar{\exists}\bar{x}.\varphi$ for the elimination of all variables but $\bar{x}$ from φ. We use $t[X/Y]$ to replace all occurrences of X by Y in a syntactic object t.

3.1 Cost Relations: The Common Target of Cost Analyzers

Let us now recall the general notion of *CRs* as defined in [5]. The basic building blocks of *CRs* are the so-called *cost expressions* which are generated using this grammar:

$$e ::= \ r \ \mid \ \|l\| \ \mid \ e + e \ \mid \ e * e \ \mid \ e^r \ \mid \ \log(\|l\| + 1) \ \mid \ n^{\|l\|} \ \mid \ \max(S)$$

where $r \in \mathbb{Q}_+$, $n \in \mathbb{Q}_+$ and $n \geq 1$, l is a linear expression over $\mathbb{Z}$. S is a nonempty set of cost expressions and $\|.\| : \mathbb{Q} \to \mathbb{Q}_+$ is defined as $\|v\| = \max(\{v, 0\})$. Note that $\|.\|$ is read as "`nat`" (for natural numbers) and $\|l\|$ as "`nat` of l". Importantly, linear expressions are always wrapped by $\|.\|$ in order to avoid negative evaluations. For instance, as we will see later, an UB for $C(k, j, q)$ in Figure 2.2 is $\|q + j - k\|$. Without the use of $\|.\|$, the evaluation of $C(5, 5, 11)$ results in the negative cost -1 which must be lifted to zero, since it corresponds to an execution in which the `for` loop is not entered (i.e., $k \geq q + j$). Moreover, $\|.\|$ expressions provide a compact representation for piecewise functions, in which each $\|l\|$ is represented by two cases for $l \geq 0$ and $l < 0$. Observe that cost expressions are monotonic in their $\|.\|$ sub-expressions, i.e., replacing $\|l\| \in e$ by $\|l'\|$ such that $l' \geq l$ results in a cost expression e' such that $e' \geq e$. This property is fundamental for the correctness of our approach.

DEFINITION 3.1.1 (Cost Relation)**.** *A CR C is defined by a set of equations of the form $\mathcal{E} \equiv \langle C(\bar{x}) = e + \sum_{i=1}^{m} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$ where $m \geq 0$; $n \geq 0$; C and D_i are cost relation symbols with $D_i \neq C$; all variables $\bar{x}$, $\bar{y}_i$ and $\bar{z}_j$ are distinct; e is a cost expression; and φ is a set of linear constraints over vars$(\mathcal{E})$.*

W.l.o.g., we make two assumptions on the *CR* that we will be using in the rest of this part of the thesis:

1. *Direct recursion:* all recursions are *direct* (i.e., cycles in the call graph are of length one). Direct recursion can be automatically achieved by applying partial evaluation as described in [5]; and

2. *Standalone cost relations:* *CRs* do not depend on any other *CR*, i.e., the equations do not contain external calls, and thus have the form $\langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$.

The second assumption can be made because our approach is compositional. We start by computing bounds for the *CRs* which do not depend on any other *CRs*, e.g., C in Figure 2.2 is solved to the UB $\|q + j - k\|$. Then, we continue by substituting the computed bounds in the equations which call such relation, which in turn become standalone. For instance, substituting the above UB in the relation B results in the equation $\langle B(j, i, q) = \|q + j\| + B(j', i, q), \{j < i, j + 1 \leq j' \leq j + 3\} \rangle$. This operation is repeated until no more *CR* need to be solved. In what follows, *CR* refers to standalone *CRs* in direct recursive form, unless we explicitly state otherwise.

The evaluation of a *CR* C for a given valuation $\bar{v}$ (integer values), denoted $C(\bar{v})$, is based on the notion of evaluation trees [5], which is similar to SLD trees in the context of Logic Programming [53].

DEFINITION 3.1.2 (Evaluation Trees)**.** *The set of evaluation trees for* $C(\bar{v})$ *is defined as follows*

$$\mathcal{T}(C(\bar{v})) = \left\{ Tree(\sigma(e), [T_1, \ldots, T_n]) \;\middle\|\; \begin{array}{l} (1)\ \langle C(\bar{x}) = e + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle \in C \\ (2)\ \sigma \in [\![\bar{v} = \bar{x} \wedge \varphi]\!] \\ (3)\ T_j \in \mathcal{T}(C(\sigma(\bar{z}_j))) \end{array} \right\}$$

A possible evaluation tree for $C(\bar{v})$ is generated as follows: In (1) we chose a matching equation from those defining the *CR* C; In (2) we chose a solution for $\bar{v} = \bar{x} \wedge \varphi$, which indicates that the chosen equation is applicable; In (3) we recursively generate an evaluation tree T_j for each recursive call $C(\sigma(\bar{z}_j))$; and then we construct an evaluation tree $Tree(\sigma(e), [T_1, \ldots, T_n])$ for $C(\bar{v})$, which has $\sigma(e)$ as the root and $T_1, \ldots, T_n$ as sub-trees. Note that due to the non-deterministic choices in (1) and (2) we might have several evaluation trees for $C(\bar{v})$. Note also that trees might be infinite. The sum of all nodes of $T \in \mathcal{T}(C(\bar{v}))$ is denoted by $sum(T)$, and the set of answers for $C(\bar{v})$ is defined as $answ(C(\bar{v})) = \{sum(T) \mid T \in \mathcal{T}(C(\bar{v}))\}$. A closed-form function

$C^*(\bar{x}_0) = e$ is an UB (resp. LB) for C, if for any valuation $\bar{v}$ it holds that $C^*(\bar{v}) \geq \max(answ(C(\bar{v})))$ (resp. $C^*(\bar{v}) \leq \min(answ(C(\bar{v})))$). Note that even if the original program is deterministic, due to the abstractions performed during the generation of the *CR*, it might happen that several results can be obtained for a given $C(\bar{v})$. Correctness of the underlying analysis used to obtain the *CR* must ensure that the actual cost is one of such solutions [5]. This makes it possible to use *CRs* to infer both UBs and LBs from them.

EXAMPLE 3.1.3. *Let us evaluate $B(0,3,3)$ for the CR B of Figure 2.2. The only matching equation is the second one for B. We choose an assignment σ. Here we have a non-deterministic choice for selecting the value of j' which can be 1, 2 or 3. We evaluate the cost of $C(0,0,3)$. Finally, one of the recursive calls of $B(1,3,3)$, $B(2,3,3)$ or $B(3,3,3)$ will be made, depending on the chosen value for j'. If we continue executing all possible derivations until reaching the base-cases, the final result for $B(0,3,3)$ is any of $\{9,10,13,14,15,18\}$. The actual cost is guaranteed to be one of such values.*

Next we recall the essentials of the approach proposed in [5] for inferring closed-form UBs for a given standalone *CR* $C(\bar{x})$. We denote by m the maximum number of recursive calls in a single equation. This approach first infers the following information:

1. *Number of applications of the recursive case:* The first dimension is to infer an UB $\hat{\kappa}$ on the number of times the recursive equations can be applied (which, for loops, corresponds to the number of iterations). This bounds the depth of the corresponding evaluation trees; and

2. *Cost of applications:* The second dimension is to infer an UB $\hat{e}$ on the cost of all loop iterations, i.e., $\hat{e} \geq e_i$ for all i. This bounds the contribution of each node in the evaluation tree.

Then, the closed-form function $C^{ub}(\bar{x}) = m^{\hat{\kappa}} * \hat{e}$ is guaranteed to be an UB for C.

EXAMPLE 3.1.4. *Let us consider the following CR again*

$$\langle C(x) = 0, \{x < 0\}\rangle$$
$$\langle C(x) = \|x\| + C(x'), \{x - 3 \leq x' \leq x - 1, x \geq 0\}\rangle$$

The approach of [5] infers $\hat{\kappa} = \|x_0 + 1\|$ *and* $\hat{e} = \|x_0\|$ *for the above mentioned dimensions. Then, it produces* $C^{ub}(x_0) = \|x_0\| * \|x_0 + 1\|$ *as an UB for the CR* C*.*

Technically, [5] solves the above two dimensions by relying on program analysis techniques as follows:

1. The first dimension is solved by inferring a *ranking function* f, such that for any recursive equation $\langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_m), \varphi \rangle$ in the *CR*, it holds that $\varphi \models f(\bar{x}) \geq f(\bar{x}_i) + 1 \wedge f(\bar{x}) \geq 0$ for all $1 \leq i \leq m$. This guarantees that when evaluating $C(\bar{x}_0)$, where variables $\bar{x}_0$ denote the initial values, the length of any chain of calls to C cannot exceed $f(\bar{x}_0)$. Thus, f bounds the length of these chains, and thus the depth of all evaluation trees.

2. The second dimension is solved by first inferring an *invariant* $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \Psi \rangle$, where Ψ is a set of linear constraints, which describes the relation between the values that $\bar{x}$ can take in any call to C and the initial values $\bar{x}_0$. Then, it generates $\hat{e}$ as follows: each $\|l\| \in e$ is replaced by $\|\hat{l}\|$ where $\hat{l}$ is a linear expression (over $\bar{x}_0$) that satisfies $\hat{l} \geq l$. In practice, $\hat{l}$ is obtained by syntactically looking for an expression $\xi \leq \hat{l}$ in $\bar{\exists} \bar{x}_0 \cup \{\xi\}.\ \Psi \wedge \varphi_1 \wedge \xi = l$ where ξ is a new variable. Alternatively, we could use parametric integer linear programming [40] in order to maximize l w.r.t. Ψ and with $\bar{x}_0$ as parameters.

The use of the above automated techniques is what makes the tools of [5] widely applicable.

Our approach for inferring UBs (resp. LBs) for a *CR* C heavily relies on the above two dimensions, but uses them in a fundamentally different way. Thus, in the rest of this thesis, we use the techniques of [5] for automatically inferring ranking functions and maximizing linear expression.

3.2 Single-Argument Recurrence Relations

It is fundamental for our work to understand the differences between *CRs* and *RRs*. The following features have been identified in [5] as main differences, which in turn justify the need to develop specific solvers to bound *CRs*:

1. *CRs* often have *multiple arguments* that increase or decrease over the relation. The number of evaluation steps (i.e., recursive calls performed) is often a function of such several arguments (e.g., in A of Figure 2.2 it depends on i and q).

2. *CRs* often contain *inexact size relations*, e.g., variables range over an interval $[a, b]$ (e.g., variable j' in B of Figure 2.2). Thus, for a given input, we might have several solutions which perform a different number of evaluation steps.

3. Even if the original programs are deterministic, due to the loss of precision in the first stage of the static analysis, *CRs* often involve several *non-deterministic equations*.

As a consequence of 2 and 3, an exact solution often does not exist and hence *CAS* just cannot be used in such cases. But, even if a solution exists, due to the above features, *CAS* do not accept *CRs* as a valid input. Below, we define a class of *RRs* that *CAS* can handle.

DEFINITION 3.2.1 (Single-argument *RR*)**.** *A* single-argument *RR* P *is defined by at most one recursive equation* $\langle P(N) = E + m * P(N-1)\rangle$ *where* E *is a function on* N *(and might have constant symbols) and* $m \in \mathbb{Z}_+$ *refers to the number of recursive calls, and a base-case equation* $\langle P(0) = \lambda\rangle$ *where* λ *is a constant symbol representing the value of the base-case.*

A closed-form solution for $P(N)$, if exists, is an arithmetic expression that depends only on the variable N (more precisely on the initial value x_0), the base-case constant symbol λ, and might include constant symbols that appear in E. Depending on the number of recursive calls m in the recursive equation and the

expression E, such solution can be of different complexity classes (exponential, polynomial, etc.).

It is worth mentioning that computing the closed-form expression of a general RR is undecidable [73]. However, there are decision algorithms for computing closed-form solutions of *Gosper-summable* [41] and *C-finite* [39] RRs. Note that if $m = 1$ and E is hypergeometric, the RR in Definition 3.2.1 becomes *Gosper-summable*. Examples of hypergeometric sequences are polynomials with coefficients from $\mathbb{Q}$ or $\mathbb{Z}$; and products of factorial, binomial or exponential expressions over the recurrence variable N. If $E = 0$, the RR $P(N)$ belongs to the *C-finite* RRs and the closed-form solutions are called *C-finite* expressions. If $E \neq 0$ but a *C-finite* expression, $P(N)$ can be transformed into *C-finite* RR and hence computing its closed-form solution is decidable [52]. To summarize, closed-form solutions of single-argument RR obtained from static cost analysis are decidable in most cases.

It is easy to see that the notion of evaluation trees for CRs can be easily adapted for RRs. The only difference is that for RRs, the call $P(v)$ has only one evaluation tree which is also complete (i.e., all levels are complete), while for CRs, the call $C(\bar{v})$ might have multiple trees with any shape.

Chapter 4

Inference of Precise Upper Bounds

In this chapter, we present our approach to accurately infer UBs for *CRs* in the following steps:

1. In Section 4.1, we handle a subclass of *CRs* which are defined by a single recursive equation and accumulate a constant cost.
2. In Section 4.2, we handle *CRs* which are still defined by a single recursive equation but accumulate non-constant costs.
3. In Section 4.3, we treat *CRs* with multiple overlapping equations.
4. In sections 4.1, 4.2 and 4.3 we assume that base-case equations always contribute cost zero, and in Section 4.4 we explain how to handle non-zero base-case equations.
5. Finally, in Section 4.5, we finish with some concluding remarks.

4.1 Cost Relations with Constant Cost

We consider *CRs* defined by a single recursive equation as depicted in Figure 4.1, where e contributes a *constant cost*, i.e., it is a constant number. As explained in

$$\langle C(\bar{x}) = 0, \varphi_0 \rangle$$
$$\langle C(\bar{x}) = e + C(\bar{x}_1) + \cdots + C(\bar{x}_m), \varphi_1 \rangle$$

Figure 4.1: *CR* with single recursive equation.

Section 2.2, any chain of calls in C when starting from $C(\bar{x}_0)$ is at most of length $\hat{f}_C(\bar{x}_0)$. We aim at obtaining an UB for C by solving a *RR* P_C in which all chains of calls are of length $\hat{f}_C(\bar{x}_0)$. Intuitively, P_C can be seen as a special case of a *RR* such that its recursive equation has m recursive calls (as in C), where all chains of calls are of length N, and each application accumulates the constant cost e. Its solution can be then instantiated for the case of C by replacing N with $\hat{f}_C(\bar{x}_0)$.

DEFINITION 4.1.1. *The worst-case RR of the CR C of Figure 4.1, when e is constant cost, is* $\langle P_C(N)=e + m * P_C(N-1)\rangle$.

The main achievement of the above transformation is that, for *CRs* with constant cost expressions, we get rid of their problematic features 1 and 2 described in Section 3.2 which prevented us from relying on *CAS* to obtain a precise solution. The following theorem explains how the closed-form solution of the *RR* P_C can be transformed into an UB for the *CR* C.

THEOREM 4.1.2. *If E is a solution for* $P_C(N)$ *of Definition 4.1.1, then* $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ *is an UB for its corresponding CR C.*

Proof. For any initial values $\bar{x}_0$, the evaluation tree T_1 of $P_C(\hat{f}_C(\bar{x}_0))$ is a complete tree such that any path (from the root to a leaf) has *exactly* $\hat{f}_C(\bar{x}_0)$ internal nodes (i.e., all nodes but the leaf) with cost e. Any evaluation tree $T_2 \in \mathcal{T}(C(\bar{x}_0))$ is a (possibly not complete) tree such that any path (from the root to a leaf) has *at most* $\hat{f}_C(\bar{x}_0)$ internal nodes, and each internal node has cost e. Thus, since the recursive equations of P_C and C have m recursive calls each, it holds that $sum(T_1) \geq sum(T_2)$ and therefore $P_C(\hat{f}_C(\bar{x}_0)) \geq C(\bar{x}_0)$. □

EXAMPLE 4.1.3. *The worst-case RR of the CR C of Figure 2.2 is* $\langle P_C(N)=1+ P_C(N-1)\rangle$, *which is solved using CAS to* $P_C(N)=N$ *for any* $N \geq 0$. *The UB for C is obtained by replacing N by the corresponding ranking function* $\hat{f}_C(k_0, j_0, q_0) =$

$\|j_0 + q_0 - k_0\|$ *which results in* $C^{ub}(k_0, j_0, q_0) = \|j_0 + q_0 - k_0\|$. *Recall that the ranking function is automatically inferred using the techniques of [5].*

4.2 Cost Relations with Non-Constant Cost

During cost analysis, in many cases we obtain *CRs* like the one of Figure 4.1, but with a non-constant expression e which is evaluated to different values e_i in different applications of the recursive equation. The transformation in Definition 4.1.1 would not be correct since in these cases e must be appropriately related to N. In particular, the main difficulty is to simulate the accumulation of the non-constant expressions e_i at the level of the *RR*. In this section we formalize the ideas intuitively explained in Section 2.2 which are based on using sequences to simulate the behavior of e.

We distinguish two cases: *CRs* with linear (a.k.a., arithmetic) and *CRs* with geometric progression behavior. In general, the cost expression e has a complex form (e.g., exponential, polynomial, etc.). Therefore, even a simple cost expression like $\|x + y\| * \|x + y\|$ does not increase arithmetically or geometrically even if the sub-expression $x + y$ does. Therefore, limiting our approach to cases in which e has a linear or geometric progression behavior would narrow its applicability. Instead, a key observation in our approach is that, it is enough to reason on the behavior of its $\|.\|$ sub-expressions, i.e., we only need to understand how each $\|l\| \in e$ changes along a sequence of calls to C, which very often have a linear or geometric progression behavior since l is a linear expression.

4.2.1 Linear Progression Behavior

This section describes how to obtain an UB for the *CR* of Figure 4.1, when e includes $\|.\|$ sub-expressions with linear progression behavior, using a *RR* that simulates the behavior of each such $\|.\|$ sub-expression separately. We first define the notion of linear progression behavior of a $\|.\|$ expression.

DEFINITION 4.2.1 ($\|.\|$ with linear progression behavior)**.** *Consider the CR* C *of Figure 4.1. We say that* $\|l\| \in e$ *has an increasing (resp. decreasing) linear*

progression behavior, if there exists a progression parameter $\check{d} > 0$*, such that for any two consecutive contributions of* e *during the evaluation of* $C(\bar{x}_0)$*, denoted* e' *and* e''*, it holds that* $l'' - l' \geq \check{d}$ *(resp.* $l' - l'' \geq \check{d}$*) where* $\|l'\| \in e'$ *and* $\|l''\| \in e''$ *are the instances of* $\|l\|$*.*

For the case of the *CR* of Figure 4.1, the two consecutive instances e' and e'' in the above definition refer to two consecutive nodes in the corresponding evaluation tree (a node e', and one of its children e''). Note that there might be several values for $\check{d}$ that satisfy the conditions of the above definition. For example, if a $\|.\|$ expression decreases at least by 2, then it also decreases at least by 1, and therefore both $\check{d} = 1$ and $\check{d} = 2$ satisfy the conditions of the above definition. Although taking $\check{d} = 1$ is sound, it results in a loss of precision. Therefore, our interest is in finding the *maximum* $\check{d}$ that satisfies the above definition. It is important to note that this maximum value for (the minimum decrease/increase) $\check{d}$ is different from the maximum decrease/increase. In practice, we compute such $\check{d}$ for a given $\|l\| \in e$ with an increasing (resp. decreasing) behavior as follows: Let $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_m),\ \varphi'_1 \rangle$ be a renamed apart instance of the recursive equation of C such that l' is the renaming of l, and for each $1 \leq i \leq m$ let $\check{d}_i$ be the result of minimizing the objective function $l' - l$ (resp. $l - l'$) with respect to $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y}$ using integer programming, then $\check{d} = \min(\check{d}_1, \ldots, \check{d}_m)$.

EXAMPLE 4.2.2. *Consider again the cost relation* B *of Figure 2.2. Replacing the call* $C(0, j, q)$ *by the UB* $\|q + j\|$ *computed in Example 4.1.3 results in* $\langle B(j, i, q) = \|q + j\| + B(j', i, q), \varphi_1 \rangle$ *where* $\varphi_1 = \{j < i, j + 1 \leq j' \leq j + 3\}$*. The following is a renamed apart instance of the equation:* $\langle B(j_r, i_r, q_r) = \|q_r + j_r\| + B(j'_r, i_r, q_r), \varphi'_1 \rangle$ *where* $\varphi'_1 = \{j_r < i_r, j_r + 1 \leq j'_r \leq j_r + 3\}$*. Minimizing the objective function* $(q_r + j_r) - (q + j)$ *with respect to* $\varphi_1 \wedge \varphi'_1 \wedge \{j' = j_r, i = i_r, q = q_r\}$ *results in* $\check{d}_1 = 1$*. Therefore,* $\|q + j\|$ *has an increasing linear progression behavior with a progression parameter* $\check{d} = 1$*.*

Intuitively, the goal is to use a linear sequence that starts from the maximum value that a given $\|l\| \in e$ can take, i.e., $\|\hat{l}\|$, and in each step decreases by the minimum distance $\check{d}$ between two consecutive instances of $\|l\|$. Let us explain how our method works by focusing on a single $\|l\| \in e$ within the relation C, assuming

that it has a decreasing linear progress behavior with a progression parameter $\check{d}$. Recall that during the evaluation of an initial query $C(\bar{x}_0)$, any chain of calls has a length $\kappa \leq \hat{f}_C(\bar{x}_0)$. Let $\|l_1\|, \ldots, \|l_\kappa\|$ be the instances of $\|l\|$ contributed in each call. Our aim is to generate a sequence of elements $a_1, \ldots, a_\kappa$ such that $a_i \geq \|l_i\|$. Then, each a_i will be used instead of $\|l_i\|$ in order to over-approximate the total cost contributed by the i-th call.

Since $l_i - l_{i+1} \geq \check{d}$, for the first κ elements of the sequence $\{a_1 = \|\hat{l}\|, a_i = a_{i-1} - \check{d}\}$ it holds that $a_1 \geq l_1, \ldots, a_\kappa \geq l_\kappa$. However, this does not imply that $a_i \geq \|l_i\|$ since when $l_i < 0$ we have $\|l_i\| = 0$ but the corresponding a_i might be negative. This mainly happens because κ is an over-approximation of the actual length of the chain of calls. Therefore, an imprecise (too large) $\hat{f}_C$ would lead to a too large decrease and the smallest element $\|\hat{l}\| - \check{d} * (\hat{f}_C(\bar{x}_0) - 1)$, and possibly other subsequent ones, could be negative and would provide an incorrect result.

We avoid this problem by viewing this sequence in a dual way: we start from the smallest value and in each step increase it by $\check{d}$. Since still the smallest values could be negative, assuming that $\hat{f}_C(\bar{x}_0) = \|l'\|$, we start from $\|\hat{l} - \check{d} * l'\| + \check{d}$ which is guaranteed to be positive and greater than or equal to $\|\hat{l}\| - \check{d} * (\hat{f}_C(\bar{x}_0) - 1)$. This mean that when the smallest value is negative, we shift the sequence and start from a positive smallest value $\check{d}$ until the biggest value $\|l'\| * \check{d} \geq \|\hat{l}\|$. Therefore, using $a_i = \|\hat{l} - \check{d} * l'\| + (\hat{f}_C(\bar{x}_0) - i + 1) * \check{d}$, it is guaranteed that $a_1 \geq \|l_1\|, \ldots, a_\kappa \geq \|l_\kappa\|$. Similar reasoning can be done for the case in which $\|l\| \in e$ is linearly increasing by $\check{d}$. The next definition, that generalizes Definition 4.1.1, uses this intuition to replace each $\|.\|$ by an expression that generates its corresponding sequence at the level of RR.

DEFINITION 4.2.3. *Consider the CR C of Figure 4.1, and let $\hat{f}_C(\bar{x}_0) = \|l'\|$. Its associated worst-case RR is $\langle P_C(N) = \hat{E}_e + m * P_C(N-1)\rangle$ where $\hat{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by l_{RR} such that $l_{RR} \equiv \|\hat{l} - \check{d} * l'\| + (\|l'\| - N + 1) * \check{d}$ (resp. $l_{RR} \equiv \|\hat{l} - \check{d} * l'\| + N * \check{d}$) if $\|l\|$ is linearly increasing (resp. decreasing) with a progression parameter $\check{d}$; otherwise $l_{RR} \equiv \|\hat{l}\|$.*

EXAMPLE 4.2.4. *Let us see how a given $\|l\| \in e$, which is linearly decreasing by $\check{d}$, is simulated in P_C. If we apply P_C on $N = \hat{f}_C(\bar{x}_0) = \|l'\|$, i.e., on the maximum depth of the evaluation tree, we get $\|\hat{l} - \check{d} * l'\| + \|l'\| * \check{d}$, then in the*

following iteration, when applying P_C on $N-1$, we get $\|\hat{l} - \check{d} * l'\| + (\|l'\| - 1) * \check{d}$ which is smaller than the first one, and so forth. If $\|l\|$ is linearly increasing by $\check{d}$, then in the first application of P_C we get $\|\hat{l} - \check{d} * l'\| + \check{d}$, in the second one we get $\|\hat{l} - \check{d} * l'\| + 2\check{d}$, and so forth.

Note that, in Definition 4.2.3, if $\|l\| \in e$ does not have a linear progression behavior then it is replaced by $\|\hat{l}\|$, exactly as in [5]. Note also that the distinction between the decreasing and increasing case is of great importance when the *CR* has more than one recursive call. This affects the number of times the element $\|\hat{l}\|$ is contributed: one time (in the root of the evaluation tree) in the decreasing case, and $2^{(N-1)}$ times (the last level of internal nodes in the evaluation tree) in the increasing case. The following theorem explains how the closed-form solution of the *RR* P_C can be transformed into an UB for the *CR* C.

THEOREM 4.2.5. *If E is a solution for $P_C(N)$ of Definition 4.2.3, then $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for its corresponding CR C.*

Proof. We prove the theorem for the case when e is linearly decreasing. The case of linearly increasing is dual. Let $T_1 \in \mathcal{T}(C(\bar{x}_0))$, and T_2 be the evaluation tree of $P_C(\hat{f}_C(\bar{x}_0))$. Observe that: (1) The leaves have cost 0; (2) The number of *internal nodes* in any path from the root to a leaf in T_1 is at most $\hat{f}_C(\bar{x}_0)$, and in T_2 is exactly $\hat{f}_C(\bar{x}_0)$; (3) The *RR* P_C and the *CR* C have the same number of recursive calls in their recursive equation; and (4) $\hat{E}_e$ and e are identical up to their $\|.\|$ components since $\hat{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by $\|\hat{l} - l' * \check{d}\| + N * \check{d}$. These observations, together with the fact that cost expressions are monotonic, implies that in order to prove the theorem, it is enough to prove that for any $\|l\| \in e$ and its corresponding $L = \|\hat{l} - l' * \check{d}\| + N * \check{d}$ in $\hat{E}_e$, if $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ in a given path in T_1, and $L_1, \ldots, L_{\hat{f}_C(\bar{x}_0)}$ are those of L in T_2, then $L_i \geq \|l_i\|$ for any $1 \leq i \leq \kappa$. Recall that $\kappa \leq \hat{f}_C(\bar{x}_0)$.

Base Case: L_1 is obtained when $N = \hat{f}_C(\bar{x}_0)$, therefore $L_1 = \|\hat{l} - l' * \check{d}\| + \hat{f}_C(\bar{x}_0) * \check{d} \geq \|\hat{l}\| - \|l'\| * \check{d} + \hat{f}_C(\bar{x}_0) * \check{d} = \|\hat{l}\| \geq \|l_1\|$. Recall that $\|l'\| = \hat{f}_C(\bar{x}_0)$.

Inductive Case: First, we assume that $L_i \geq \|l_i\|$. Next, we consider two cases: (1) if $l_i \geq \hat{d}$, then $\|l_{i+1}\| \leq \|l_i\| - \check{d} \leq L_i - \check{d} = \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) -$

$P_A(N) = \frac{1}{2} * (\|i_0 - 1\| + (\|\frac{q_0 - i_0}{2}\| - N + 1) * 2) * (2 * \|q_0 - 1\| + \|i_0 - 1\| + (\|\frac{q_0 - i_0}{2}\| - N + 1) * 2 + 1) + P_A(N-1)$
$P_B(N) = \|q_0 + j_0 - 1\| + (\|i_0 - j_0\| - N + 1) * 1 + P_B(N-1)$
$P_C(N) = 1 + P_C(N-1)$

Figure 4.2: Worst-Case *RRs* automatically obtained from *CRs* in Figure 2.2.

$(i-1)) * \check{d} - \check{d} = \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) - i) * \check{d} = L_{i+1}$; and (2) if $l_i < \hat{d}$, then $\|l_{i+1}\| = 0 \leq \|\hat{l} - l' * \check{d}\| + (\hat{f}_C(\bar{x}_0) - i) * \check{d} = L_{i+1}$. □

EXAMPLE 4.2.6. *Consider the standalone CR B of Example 4.2.2, and recall that $\|q + j\|$ increases linearly with a progression parameter $\check{d} = 1$. Function $\hat{f}_B(j_0, i_0, q_0) = \|i_0 - j_0\|$ is a ranking function for CR B. Maximizing $\|q + j\|$ results in $\|q_0 + i_0 - 1\|$. Then, using Definition 4.2.3 we generate the* worst-case *RR $P_B(N)$ depicted in Figure 4.2 whose solution (computed by CAS) is:*

$$P_B(N) = \|q_0 + j_0 - 1\| * N + \|i_0 - j_0\| * N + \frac{N}{2} - \frac{N^2}{2}$$

By Theorem 4.2.5, replacing N by $\hat{f}_B(j_0, i_0, q_0)$ results in:

$$B^{ub}(j_0, i_0, q_0) = \|q_0 + j_0 - 1\| * \|i_0 - j_0\| + \frac{\|i_0 - j_0\|}{2} * (\|i_0 - j_0\| + 1)$$

Substituting this UB in the cost relation A of Figure 2.2 results in the CR:

$$\langle A(i, q) = \|q - 1\| * \|i\| + \frac{\|i\|}{2} * (\|i\| + 1) + A(i', q), \{i + 1 \leq q, i + 2 \leq i' \leq i + 4\}\rangle$$

Note that in this CR the expression $\|q - 1\|$ always evaluates to the same value, while $\|i\|$ has an increasing linear progression behavior with progression parameter $\check{d} = 2$. Given that: (1) $\hat{f}_A(i_0, q_0) = \|\frac{q_0 - i_0}{2}\|$; (2) the maximization of $\|q - 1\|$ is $\|q_0 - 1\|$; and (3) the maximization of $\|i\|$ is $\|q_0 - 1\|$, by applying Definition 4.2.3, we generate the worst-case *RR $P_A(N)$ depicted in Figure 4.2, which is solved by CAS to:*

$$\begin{aligned} P_A(N) \;=\; & \tfrac{N}{6} * [4 * N^2 + 3 * \|i_0 - 1\| * (2 * N + \|i_0 - 1\| + 3) + \\ & 6 * \|q_0 - 1\| * (\|i_0 - 1\| + N + 1) + 9 * N + 5] \end{aligned}$$

By Theorem 4.2.5, replacing N by $\hat{f}_A(i_0, q_0)$ results in:

$$\begin{aligned} A^{ub}(i_0, q_0) &= \tfrac{1}{6} * \|\tfrac{q_0-i_0}{2}\| * (4 * \|\tfrac{q_0-i_0}{2}\| * \|\tfrac{q_0-i_0}{2}\| + 3 * \|i_0 - 1\| * (2 * \|\tfrac{q_0-i_0}{2}\| \\ &\quad + \|i_0 - 1\| + 3) + 6 * \|q_0 - 1\| * (\|i_0 - 1\| + \|\tfrac{q_0-i_0}{2}\| + 1) + 9 * \|\tfrac{q_0-i_0}{2}\| + 5) \end{aligned}$$

Finally, substituting $A^{ub}(0, q_0)$ in the CR F, we obtain the UB:

$$F^{ub}(q_0) = \frac{1}{6} * \|\frac{q_0}{2}\| * (4 * \|\frac{q_0}{2}\| * \|\frac{q_0}{2}\| + 6 * \|q_0 - 1\| * (\|\frac{q_0}{2}\| + 1) + 9 * \|\frac{q_0}{2}\| + 5)$$

whereas [5] obtains $2 * \|\frac{q_0+1}{2}\| * \|q_0 - 1\|^2$, which is much less precise.

4.2.2 Geometric Progression Behavior

The techniques of Sections 4.1 and 4.2.1 can solve a wide range of *CRs*. However, in practice, we find also *CRs* that do not have constant or linear progression behavior, but rather a geometric progression behavior. This is typical in programs that implement divide and conquer algorithms, where the problem (i.e., the input) is divided into sub-problems which are solved recursively.

EXAMPLE 4.2.7. *Consider the following implementation of the merge-sort algorithm:*

```
1 void msort(int a[], int low, int hi) {
2   if ( hi > low ) {
3     int mid=(hi+low)/2;
4     msort(a,low,mid);
5     msort(a,mid+1,hi);
6     merge(a,low,mid,hi);
7   }
8 }
```

where, for simplicity, we omit the code of merge *and assume that its cost, for example, is $10 * \|hi - low + 1\|$, when counting the number of executed (bytecode) instructions. Using this UB,* COSTA *[6] automatically generates the following CR for* msort*:*

$$\langle msort(a, low, hi) = 0, \varphi_1 \rangle$$
$$\langle msort(a, low, hi) = 20 + 10 * \|hi - low + 1\| + msort(a, low, mid) + msort(a, mid', hi), \varphi_2 \rangle$$

where

$$\varphi_1 = \{hi \geq 0, low \geq 0, hi \leq low\}$$
$$\varphi_2 = \{hi \geq 0, low \geq 0, hi \geq low{+}1, mid'{=}mid{+}1, low{+}hi{-}1 \leq 2{*}mid \leq low{+}hi\}$$

The constant 20 *corresponds to the cost of executing the comparison, the sum and division, and invoking the methods. The constraint* $low + hi - 1 \leq 2 * mid \leq low+hi$ *in* φ_2 *is used to model the behavior of the* integer division *mid=(low+hi)/2 with linear constraints. The progression behavior of* $\|hi - low + 1\|$ *is geometric, i.e., if* $\|l_i\|$ *and* $\|l_{i+1}\|$ *are two instances of* $\|hi - low + 1\|$ *in two consecutive calls, then* $l_i \geq 2 * l_{i+1} - 1$ *holds, which means that the value of* $\|hi - low + 1\|$ *is reduced* almost *by half at each iteration. It is not reduced exactly by half since* $l_i \geq 2 * l_{i+1}$ *does not hold when the input array is of odd size, in such case it is divided into two sub-problems with different (integer) sizes.*

The above example demonstrates that: (1) there is a practical need for handling *CRs* with geometric progression behavior; and (2) the geometric progression in programs that manipulate integers does not comply the standard definition $u_i = c * r^i$ of geometric series, but rather it should consider small *shifts* around those values in order to account for examples like divide-and-conquer algorithms. The following definition specifies when a $\|.\|$ expression has a geometric progression behavior.

DEFINITION 4.2.8 ($\|.\|$ with geometric progression behavior)**.** *Consider the CR C of Figure 4.1. We say that* $\|l\| \in e$ *has an increasing (resp. decreasing) geometric progression behavior, if there exist progression parameters* $\check{r} > 1$ *and* $\check{p} \in \mathbb{Q}$*, such that for any two consecutive contributions of e during the evaluation of* $C(\bar{x}_0)$*, denoted* e' *and* e''*, it holds that* $l'' \geq \check{r} * l' + \check{p}$ *(resp.* $l' \geq \check{r} * l'' + \check{p}$*) where* $\|l'\| \in e'$ *and* $\|l''\| \in e''$ *are the instances of* $\|l\|$*.*

Note that the above increasing and decreasing conditions could be equivalently written as $\frac{l''}{\check{r}} + \check{p} \geq l'$ and $l'' \leq \frac{l'}{\check{r}} + \check{p}$ respectively. This might be more common in the literature, however, it does not lead to a simpler formalism. Thus, we prefer to use those of the above definition to keep the notation simpler.

As in the case of $\check{d}$ in the linear progression behavior, we are interested in values for $\check{r}$ and $\check{p}$ that are as close as possible to the *minimal* progression of

$\|l\|$. This happens when $\check{r}$ is maximal, and for that maximal $\check{r}$, the value of $|\check{p}|$ is minimal. In practice, computing such $\check{r}$ and $\check{p}$ for a given $\|l\| \in e$ with an increasing (resp. decreasing) behavior is done as follows: Let $\langle C(\bar{y}) = e' + C(\bar{y}_1) + \cdots + C(\bar{y}_m),\ \varphi'_1 \rangle$ be a renamed apart instance of the recursive equation of C such that l' is the renaming of l, then we look for $\check{r}$ and $\check{p}$ such that for each $1 \leq i \leq m$ it holds that $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y} \models l' \geq \check{r} * l + \check{p}$ (resp. $\varphi_1 \wedge \varphi'_1 \wedge \bar{x}_i = \bar{y} \models l \geq \check{r} * l' + \check{p}$). This can be done using Farkas' Lemma [74], which provides a systematic way to derive all implied inequalities of a given set of linear constraints [71]. However, systematically checking the conditions taking the coefficients and the constants that appear in φ_1 as candidates for $\check{r}$ and $\check{p}$, respectively, works very well in practice.

EXAMPLE 4.2.9. *For the CR of Example 4.2.7, we have that $\|hi - low + 1\|$ is decreasing geometrically, with progression parameters $\check{r} = 2$ and $\check{p} = -1$. Note that 2 and -1 explicitly appear as coefficient and constant, respectively, in φ_1.*

Similarly to the case of linear progression behavior in Section 4.2.1, the progression parameters $\check{r}$ and $\check{p}$ are used in order to over-approximate the contributions of a given $\|l\| \in e$ expression along a chain of calls. For example, if $\|l\| \in e$ has a decreasing geometric progression behavior, and $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ along any chain of calls where $\kappa \leq \hat{f}_C(\bar{x}_0)$, then first κ elements of the sequence

$$u_i = \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|-\check{p}\| * \sum_{j=1}^{i-1} \frac{1}{\check{r}^j}$$

satisfy $u_i \geq \|l_i\|$. We use $\|-\check{p}\|$ in order to lift the negative value $-\check{p}$ (when $\check{p} > 0$) to zero and avoid that u_i goes into negative values. The following definition extends Definition 4.2.3, by handling the translation of $\|.\|$ expression with geometric behavior. First, to simplify the notation, let us denote the sum $\sum_{j=1}^{i} \frac{1}{\check{r}^j}$ by $\hat{S}(i)$, which is also equal to $\frac{1}{\check{r}^i} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}}$.

DEFINITION 4.2.10. *We extend Definition 4.2.3 for the geometric progression case as follow: if $\|l\| \in e$ has an increasing (resp. decreasing) geometric progression behavior, then its corresponding l_{RR} is defined as*

$$l_{RR} \equiv \frac{\|\hat{l}\|}{\check{r}^{(N-1)}} + \|-\check{p}\| * \hat{S}(N-1) \quad \left[\textit{resp. } l_{RR} \equiv \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-N)}} + \|-\check{p}\| * \hat{S}(f_C(\bar{x}_0) - N) \right]$$

Note that value of $\frac{\|\hat{l}\|}{\check{r}^{(N-1)}} + \|-\check{p}\| * \hat{S}(N-1)$ decreases along the iterations of P_C, i.e., when N decreases. Similarly, the value of $\frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-N)}} + \|-\check{p}\| * \hat{S}(f_C(\bar{x}_0) - N)$ increases. Note also that the distinction between the decreasing and the increasing cases is fundamental, and it is for the same reasons as in Definition 4.2.3. The following theorem explains how the closed-form solution of the *RR* P_C can be transformed into an UB for the *CR* C.

THEOREM 4.2.11. *If E is a solution for $P_C(N)$ of Definition 4.2.3, together with the extension of Definition 4.2.10, then $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for its corresponding CR C.*

Proof. The proof is similar to the one of Theorem 4.2.5. We prove the theorem for the case when e is geometrically decreasing. The case of geometrically increasing is dual. Let $T_1 \in \mathcal{T}(C(\bar{x}_0)$ and T_2 be the evaluation tree of $P_C(\hat{f}_C(\bar{x}_0))$. The observations about T_1, T_2 and the monotonicity property in the proof of Theorem 4.2.5 also hold for this case, and therefore it is enough to prove that for any $\|l\| \in e$ and its corresponding $L = \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-N)}} + \|-\check{p}\| * \hat{S}(\hat{f}_C(\bar{x}_0) - N)$ in $\hat{E}_e$, if $\|l_1\|, \ldots, \|l_\kappa\|$ are instances of $\|l\|$ in a given path in T_1, and $L_1, \ldots, L_{\hat{f}_C(\bar{x}_0)}$ are those of L in T_2, then $L_i \geq \|l_i\|$ for any $1 \leq i \leq \kappa$.

Base Case: L_1 is obtained when $N = \hat{f}_C(\bar{x}_0)$, therefore $L_1 = \frac{\|\hat{l}\|}{\check{r}^{(\hat{f}_C(\bar{x}_0)-\hat{f}_C(\bar{x}_0))}} + \|-\check{p}\| * \hat{S}(\hat{f}_C(\bar{x}_0) - \hat{f}_C(\bar{x}_0)) = \|\hat{l}\| \geq \|l_1\|$ since $\hat{S}(0) = 0$.

Inductive Case: We assume that $L_i \geq \|l_i\|$ and will prove that $L_{i+1} \geq \|l_{i+1}\|$. We have $\hat{S}(i) = \frac{1}{\check{r}^{(i)}} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}} = \frac{1}{\check{r}} * (\frac{1}{\check{r}^{(i-1)}} * \frac{1}{1-\check{r}} - \frac{1}{1-\check{r}}) + \frac{1}{\check{r}*(1-\check{r})} - \frac{1}{(1-\check{r})} = \frac{1}{\check{r}} * \hat{S}(i-1) + \frac{1}{\check{r}}$. We also have $L_i = \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|-\check{p}\| * \hat{S}(i-1)$. Then, the following equations hold:

$$
\begin{array}{rcl}
L_{i+1} & = & \frac{\|\hat{l}\|}{\check{r}^{i}} + \|-\check{p}\| * \hat{S}(i) \\
& = & \frac{1}{\check{r}} * \frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|-\check{p}\| * (\frac{1}{\check{r}} * \hat{S}(i-1) + \frac{1}{\check{r}}) \\
& = & \frac{1}{\check{r}} * (\frac{\|\hat{l}\|}{\check{r}^{i-1}} + \|-\check{p}\| * \hat{S}(i-1)) + \|-\check{p}\| * \frac{1}{\check{r}} \\
& = & \frac{1}{\check{r}} * (L_i + \|-\check{p}\|) \\
& \geq & \frac{1}{\check{r}} * (\|l_i\| + \|-\check{p}\|) \\
& \geq & \|l_{i+1}\| \quad [\text{Since } \|-\check{p}\| \geq -\check{p} \text{ and } \varphi_1 \models l_i \geq \check{r} * l_{i+1} + \check{p}]
\end{array}
$$

□

Algorithm 1: compute_UB

Input: Statndalone *CR* C, as in Figure 4.1

Output: Close-form UB $C^{ub}(\bar{x}_0)$ for C

Compute a loop bound $\hat{f}_C(\bar{x}_0)$;
Compute an invariant $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \Psi \rangle$;
$\hat{E}_e = e$;
foreach $\|l\| \in e$ **do**
    Let $\hat{l}$ be the result of maximizing l w.r.t $\Psi \wedge \varphi_1$ and the parameter $\bar{x}_0$;
    Compute the progeression parameters $\check{d}$ or $\langle \check{r}, \check{p} \rangle$ for l;
    Compute l_{RR} as in definitions 4.2.3 and 4.2.10;
    $\hat{E}_e = \hat{E}_e[\|l\|/l_{RR}]$;
end
Solve $\langle P_C(N) = \hat{E}_e + m * P_C(N-1) \rangle$ into a closed-form expression E using *CAS*;
$C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$;

EXAMPLE 4.2.12. *Consider the CR of Example 4.2.7, and recall that the expression* $\|hi - low + 1\|$ *decreases geometrically with progression parameters* $\check{r} = 2$ *and* $\check{p} = -1$ *(see Example 4.2.9). Moreover, the ranking function for the CR msort is* $\hat{f}_{msort}(a_0, low_0, hi_0) = \log_2(\|hi_0 - low_0\| + 1) + 1$*, and maximization of* $\|hi - low + 1\|$ *results in* $\|hi_0 + 1\|$*. According to Definition 4.2.10, the associated worst-case RR (after simplifying* $\hat{E}_e$ *for clarity) is:*

$$P_{msort}(N) = 30 + 10 * \frac{\|hi_0 + 1\| - 1}{2^{(\log_2(\|hi_0 - low_0\|+1)+1-N)}} + 2 * P_{msort}(N-1)$$

Obtaining a closed-form solution for $P_{msort}(N)$ *using CAS, and then replacing* N *by* $\hat{f}_{msort}(a_0, hi_0, low_0)$ *results in the following UB for the CR msort:*

$$msort^{ub}(a_0, low_0, hi_0) = 30 + 60 * \|hi_0 - low_0\| + 10 * (\log_2(\|hi_0 - low_0\| + 1) + 1) * (\|hi_0 + 1\| - 1)\,.$$

Algorithm 1 summarizes the process of solving standalone *CR* of the form given in Figure 4.1, as described in sections 4.1 and 4.2. As we have explained in Section 3.1, non-standalone *CRs* are solved in a modular way. We first apply

$$\langle C(\bar{x}) = 0, \varphi_0 \rangle$$
$$\langle C(\bar{x}) = e_1 + C(\bar{x}_1) + \cdots + C(\bar{x}_{m_1}), \varphi_1 \rangle$$
$$\vdots$$
$$\langle C(\bar{x}) = e_h + C(\bar{x}_1) + \cdots + C(\bar{x}_{m_h}), \varphi_h \rangle$$

Figure 4.3: *CRs* with multiple recursive equations.

Algorithm 1 to the *CR* that does not call any other *CRs* (i.e., it is standalone), then we continue by substituting the computed bounds in the equations that call such relation, which in turn become standalone, and thus can be solved using Algorithm 1. This process is applied until all *CRs* are solved.

4.3 Non-constant Cost Relations with Multiple Equations

Any approach for solving *CRs* that aims at being practical has to consider *CRs* with several recursive equations as the one depicted in Figure 4.3. This kind of *CRs* is very common during cost analysis, and they mainly come from conditional statements inside loops. For instance, the instruction **if** (x[i]>0) *A* **else** B, may lead to two nondeterministic equations which accumulate the costs of A and B. This is because arrays are typically abstracted to their length and, hence, the condition x[i]>0 is abstracted to *true*, i.e., we do not keep this information in the corresponding *CR*. Hence, $\varphi_1, \ldots, \varphi_h$ are not necessarily mutually exclusive. In what follows, w.l.o.g., we assume that $m_1 \geq \cdots \geq m_h$, i.e., the first (resp. last) recursive equation has the maximum (resp. minimum) number of recursive calls among all equations.

As a first solution to the problem of inferring an UB for the *CR* of Figure 4.3, we simulate its *worst-case* behavior using another single-recursive *CR* $\hat{C}$ whenever possible. We refer to $\hat{C}$ as the *worst-case CR* of C. Namely, we generate the following *CR*

$$\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \ldots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$$

such that the evaluation trees of $\hat{C}(\bar{x}_0)$ up to depth $\hat{f}_C(\bar{x}_0)$ over-approximate the evaluation trees of $C(\bar{x}_0)$. Then, we will infer an UB on the evaluation trees of $\hat{C}(\bar{x}_0)$ up to such depth, by generating a corresponding RR, which is then guaranteed to be an UB for $C(\bar{x}_0)$. The process of constructing $\hat{C}$ will be discussed later in this section. Let us start by formalizing the conditions that $\hat{C}$ should satisfy, and how we approximate its evaluation trees up to a given depth.

DEFINITION 4.3.1. *We say that $\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \cdots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$ is a worst-case CR for the CR C of Figure 4.3, if for any valuation $\bar{v}$ it holds that*

$$max(\{sum(T, \hat{f}_C(\bar{v})) \mid T \in \mathcal{T}(\hat{C}(\bar{v}))\}) \geq max(answ(C(\bar{v})))$$

where $sum(T, \hat{f}_C(\bar{v}))$ denotes the sum of all nodes in T up to depth $\hat{f}_C(\bar{v})$.

Intuitively, we require that when evaluating $\hat{C}(\bar{v})$ until the maximum depth of the trees of $C(\bar{v})$, i.e., until depth $\hat{f}_C(\bar{v})$, we already get a larger cost than when evaluating $C(\bar{v})$. Note that we do not require the evaluation trees of $\hat{C}(\bar{v})$ to be finite, and indeed in some cases they are not, i.e., the loops of $\hat{C}$ are possibly non-terminating. This is because, when generating $\hat{C}$, we usually generalize $\varphi_1, \ldots, \varphi_h$ into φ which might affect the termination behavior. The following definition explains how to construct a worst-case RR for $\hat{C}$, that we use to approximate its cost up to depth $\hat{f}_C(\bar{v})$.

DEFINITION 4.3.2. *Given the CR C of Figure 4.3, a corresponding worst-case CR $\hat{C}$ as in Definition 4.3.1, and a ranking function $\hat{f}_C(\bar{x}_0)$ for C. The worst-case RR of $\hat{C}$ is defined as $\langle P_{\hat{C}}(N) = \hat{E}_e + m_1 * P_{\hat{C}}(N-1) \rangle$, where $\hat{E}_e$ is generated as in definitions 4.2.3 and 4.2.10, with the only difference of using $\hat{f}_C(\bar{x}_0)$ instead of $\hat{f}_{\hat{C}}(\bar{x}_0)$.*

Let us clarify how we compute $\hat{E}_e$ from e in the above definition. In principle, it is computed as in definitions 4.2.3 and 4.2.10 but using $\hat{f}_C(\bar{x}_0) = \|l'\|$, and not $\hat{f}_{\hat{C}}(\bar{x}_0) = \|l'\|$, in order to account only for paths of at most length $\hat{f}_C(\bar{x}_0)$. Apart from this difference, it is important to note that when computing l_{RR} for $\|l\| \in e$: (1) the progression parameters are computed using $\hat{C}$ (i.e., using the constraints φ); and (2) $\|\hat{l}\|$ is computed by considering an invariant of $\hat{C}$, i.e., $\langle \hat{C}(\bar{x}_0) \rightsquigarrow \hat{C}(\bar{x}), \Psi \rangle$.

In what follows, we describe how to construct a worst-case *CR* $\hat{C}$. The set of constraints φ is simply constructed as the convex-hull of $\varphi_1, \ldots, \varphi_h$, taking into account that each φ_i might include local variables that do not occur in other φ_j. Next we describe how to compute e. Observe that any cost expression (that does not include max) can be normalized to the form $\Sigma_{i=1}^{n}\Pi_{j=1}^{n_i} b_{ij}$ (i.e., sum of multiplications) where each b_{ij} is a *basic cost expression* of the form $\{r, \|l\|, m^{\|l\|}, \log(\|l\|+1)\}$. This normal form allows constructing e by considering the basic components of $e_1, \ldots, e_h$. For simplicity, we assume that $e_1, \ldots, e_h$ are given in this normal form, otherwise they could be normalized first. The following definition introduces the notion of a *generalization operator* for basic cost expressions. W.l.o.g., we consider that $e_1, \ldots, e_h$ have the same number of multiplicands n, and that all multiplicands have the same number of basic cost expressions m. This is not a restriction since otherwise, we just add 1 in multiplication and 0 in sum to achieve this form.

DEFINITION 4.3.3 (generalization of cost expressions)**.** *A generalization operator* $\sqcup$ *is a mapping from pairs of basic cost expressions to cost expressions such that it satisfies* $a \sqcup b \geq a$ *and* $a \sqcup b \geq b$*. The* $\sqcup$*-generalization of two CR expressions* $e_1 = \Sigma_{i=1}^{n}\Pi_{j=1}^{m} a_{ij}$ *and* $e_2 = \Sigma_{i=1}^{n}\Pi_{j=1}^{m} b_{ij}$ *is defined as* $e_1 \sqcup e_2 = \Sigma_{i=1}^{n}\Pi_{j=1}^{m}(a_{ij} \sqcup b_{ij})$.

The above definition does not provide an algorithm for generalizing two cost expressions, but rather a general method which is parametrized in: (1) the actual generalization operator $\sqcup$; and (2) the order of the multiplicands and the order of their basic cost expressions (since we generalize basic cost expressions with the same indexes). It is important to notice that there is no best-solution for these points, and that in practice heuristic-based solutions should be used. Below we describe such solution.

As regards (1), any generalization operator should try first to prove that $a_{ij} \geq b_{ij}$ or $a_{ij} \leq b_{ij}$, and take the bigger one as the result. Such comparison is feasible due to the simple forms of the basic cost expressions, which are also known a priori. This means that one could generate a set of rules that specify conditions under which it is guaranteed that one cost expression is bigger than another one. E.g., $\|l_1\| \geq \|l_2\|$ if $l_1 \geq l_2$. In [4], such rules are defined for comparing cost expressions in general. When the comparison fails, a possible

Algorithm 2: compute_UB_MulEqn_1

Input: *CR* C with multiple equations, as in Figure 4.3

Output: Close-form upper bound $C^{ub}(\bar{x}_0)$

Compute a loop bound $\hat{f}_C(\bar{x}_0)$;
Compute $e = e_1 \sqcup \cdots \sqcup e_h$;
Generalize $\varphi_1, \cdots, \varphi_h$ into φ;
Let $\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \ldots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$ be a wrost-case *CR* for C;
Compute an invariant $\langle \hat{C}(\bar{x}_0) \rightsquigarrow \hat{C}(\bar{x}), \Psi \rangle$;
$\hat{E}_e = e$;
foreach $\|l\| \in e$ **do**
    Let $\hat{l}$ be the result of maximizing l w.r.t $\Psi \wedge \varphi$ and the parameter $\bar{x}_0$;
    Compute the progeression parameters $\check{d}$ or $\langle \check{r}, \check{p} \rangle$ for l using $\hat{C}$;
    Compute l_{RR} as in definitions 4.2.3 and 4.2.3, using $\hat{f}_C(\bar{x}_0)$;
    $\hat{E}_e = \hat{E}_e[\|l\|/l_{RR}]$;
end
Solve $\langle P_{\hat{C}}(N) = \hat{E}_e + m_1 * P_{\hat{C}}(N-1) \rangle$ into a closed-form expression E using *CAS*;
$C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$;

sound solution is to take $a_{ij} + b_{ij}$. However, this might often results in too imprecise generalization. Again, the simple structure of such expressions makes it possible to build a set of generalization rules that obtain precise results. E.g., $\|2 * y_0 + z_0\|$ and $\|y_0 + 2 * z_0\|$ can be generalized into $\|2 * y_0 + 2 * z_0\|$, by taking the maximum of the coefficients that correspond to the same variables.

THEOREM 4.3.4. *Let E be a solution for $P_{\hat{C}}(N)$ of Definition 4.3.2. Then, $C^{ub}(\bar{x}_0) = E[N/\hat{f}_C(\bar{x}_0)]$ is an UB for the CR C.*

Proof. Let us consider an evaluation tree $T_1 \in \mathcal{T}(C(\bar{x}_0))$. Since $\hat{C}(\bar{x}_0)$ is the worst-case *CR* of C, for each evaluation tree T_1 of C, there exists an evaluation tree $T_2 \in \mathcal{T}(\hat{C}(\bar{x}_0))$ such that $sum(T_2, \hat{f}_C(\bar{x}_0)) \geq sum(T_1)$ holds according to definition 4.3.1. The theorem is not applicable if this fundamental property does not hold. Now the *CR* $\langle \hat{C}(\bar{x}) = e + \hat{C}(\bar{x}_1) + \cdots + \hat{C}(\bar{x}_{m_1}), \varphi \rangle$ where $e = e_1 \sqcup \cdots \sqcup e_h$

is similar to the *CR* in figure 4.1. Its corresponding worst-case RR is $\langle P_{\hat{C}}(N) = \hat{E}_e + m_1 * P_{\hat{C}}(N-1)\rangle$. Now observe that: (1) We consider the evaluation tree T_2 until the depth $\hat{f}_C(\bar{x}_0)$ and the number of *internal nodes* in any path from the root to a leaf in the evaluation tree T_3 of $P_{\hat{C}}(\hat{f}_C(\bar{x}_0))$ is exactly $\hat{f}_C(\bar{x}_0)$; and (2) The number of child nodes of any *internal node* in both T_2 and T_3 are exactly m_1. By considering these observations and according to theorems 4.2.5 and 4.2.11, we have that if E is closed-form solution of $P_{\hat{C}}(N)$, then $E[N/\hat{f}_C(\bar{x}_0)]$ is the UB of $\hat{C}(\bar{x}_0)$. So, for any $T_2 \in \mathcal{T}(\hat{C}(\bar{x}_0))$, $E[N/\hat{f}_C(\bar{x}_0)] \geq sum(T_2, \hat{f}_C(\bar{x}_0))$. Since $sum(T_2, \hat{f}_C(\bar{x}_0)) \geq sum(T_1)$ and this holds for any $T_1 \in \mathcal{T}(C(\bar{x}_0))$, $E[N/\hat{f}_C(\bar{x}_0)]$ is also the UB of $C(\bar{x}_0)$. □

Algorithms 2 summarizes the approach that we have discussed so far for solving *CRs* with multiple equations. Let us now apply it to a concrete example.

EXAMPLE 4.3.5. *Let us add the following recursive equation to the CR B:*

$$B(j,i,q) = \|j\|^2 + B(j',i,q)\ \{j+1 \leq i, j' = j+1\}$$

Note that B has a nondeterministic choice for accumulating either $e_1 = \|q+j\|$ *or* $e_2 = \|j\|^2$*, and that both* $\|q+j\|$ *and* $\|j\|$ *have increasing linear progression behavior with* $\check{d} = 1$*. Next, we compute* $e = e_1 \sqcup e_2 = \|q+j\| * \|j\|$ *and the worst-case CR* $\hat{B}$ *is*

$$\hat{B}(j,i,q) = \|q+j\| * \|j\| + \hat{B}(j',i,q)\ \{j+1 \leq i, j+1 \leq j' \leq j+3\}$$

Next we compute $\hat{E}_e = (\|q_0+j_0-1\| + (\|i_0-j_0\| - N + 1)) * ((\|j_0-1\| + (\|i_0-j_0\| - N + 1)))$*. Now we generate*

$$\langle P_{\hat{B}}(N) = (\|j_0-1\| + \|i_0-j_0\| - N + 1) * (\|q_0+j_0-1\| + \|i_0-j_0\| - N + 1) + P_{\hat{B}}(N-1)\rangle$$

which is solved by CAS to

$$\begin{aligned} P_{\hat{B}}(N) \;=\; & \tfrac{N}{6} * (2 * N^2 + 6 * \|i_0-j_0\|^2 - 6 * N * \|i_0-j_0\| + 3 * \|j_0-1\| + 6* \\ & \|i_0-j_0\| * (\|j_0-1\|+1) + 3 * \|q_0+j_0-1\| * (2 * \|j_0-1\| + 2* \\ & \|i_0-j_0\| + 1) - 3 * N * (\|j_0-1\| + \|q_0+j_0-1\| + 1) + 1) \end{aligned}$$

and finally instantiating N with $\|i_0-j_0\|$ *gives (with simplification for clarity):*

$$\begin{aligned} B^{ub}(j_0,i_0,q_0) \;=\; & \tfrac{1}{6} * \|i_0-j_0\| * [2 * \|i_0-j_0\| * \|i_0-j_0\| + 3 * \|i_0-j_0\| * \|j_0-1\| \\ & +3 * \|i_0-j_0\| * \|q_0+j_0-1\| + 3 * \|i_0-j_0\| + 3 * \|q_0+j_0-1\| \\ & +6 * \|q_0+j_0-1\| * \|j_0-1\| + 3 * \|j_0-1\| + 1] \end{aligned}$$

The above approach works very well in practice, since in many cases the cost expressions contributed by the different equations have very similar structure, and they differ only in constant expressions. However, there are some cases where this approach fails to precisely generalize expressions $e_1, \ldots, e_h$, and thus might infer imprecise UBs.

EXAMPLE 4.3.6. *Consider the following CR*

$$\begin{array}{ll} C(z,y) = 0 & \{z < 1, y < 1\} \\ C(z,y) = \|z\| + C(z',y) & \{z' = z-1, z > 0\} \\ C(z,y) = \|y\| + C(z,y') & \{y' = y-1, y > 0\} \end{array}$$

Generalizing $\|z\|$ and $\|y\|$ results in $\|z+y\|$. This leads to inferring the UB $C^{ub}(z_0, y_0) = \|z_0 + y_0\| * \|z_0 + y_0\|$, which is not precise enough.

In what follows we present an alternative approach for solving (some cases of) *CRs* with multiple equations, which is able to handle the one of the above example. The main idea is to concentrate on the contribution of each equation, independently from the rest. We start by defining the projection of a *CR* C on its i-th equation, which is used later to compute an UB on the contributions of the i-th equation.

DEFINITION 4.3.7. *Given the CR C of Figure 4.3, we denote by C_i the CR obtained by replacing each e_j when $j \neq i$ by 0.*

Clearly, if $C_i^{ub}(\bar{x}_0)$ is an UB for *CR* C_i, then $C^{ub}(\bar{x}_0) = \sum_{i=1}^{h} C_i^{ub}(\bar{x}_0)$ is an UB for *CR* C. The challenge is to compute a precise UB for each C_i. Of course one can use the generalization-based approach to solve each C_i, however this does not lead to precise UB. E.g., for the *CR* of Example 4.3.6 we obtain $\|y_0\| * \|z_0 + y_0\| + \|z_0\| * \|z_0 + y_0\|$.

Let us consider a path in an arbitrary evaluation tree of $C_i(\bar{x}_0)$, and concentrate on the contributions of a single $\|l\| \in e_i$ in this path. As we have done so far, we aim at simulating these contributions using a corresponding arithmetic or geometric sequence, and then use this sequence to generate a corresponding *RR* whose solution can be transformed into an UB for C_i. There are two important issues that should be taken into account: (1) a ranking function for C (which is

Algorithm 3: compute_UB_MulEqn_2

Input: *CR* C of Figure 4.3 with $m_1 = 1$

Output: Close-form upper bound $C^{ub}(\bar{x}_0)$

```
1  Compute an invariant ⟨C(x̄0) ⇝ C(x̄), Ψ⟩;
2  for i = 1 → h do
3      Generate CR C_i as in Definition. 4.3.7;
4      Compute a bound f̂_{C_i}(x̄0) on the number of visits to the i-the equation;
5      E_{e_i} = e_i;
6      foreach ‖l‖ ∈ e_i do
7          Let l̂ be the result of maximizing l w.r.t Ψ ∧ φ_i and the parameter
           x̄0;
8          Compute ď or ⟨ř, p̌⟩ for l (considering subsequent visits to the i-th
           equation);
9          Compute l_RR as in definitions 4.2.3 and 4.2.3, using f̂_{C_i}(x̄0);
10         Ê_{e_i} = Ê_{e_i}[‖l‖/l_RR];
11     end
12     Solve ⟨P_{Ĉ_i}(N) = Ê_{e_i} + P_{Ĉ_i}(N − 1)⟩ into a closed-form expression E_i
       using CAS;
13     C_i^{ub}(x̄0) = E_i[N/f̂_{C_i}(x̄0)] ;
14 end
15 C^{ub}(x̄0) = C_1^{ub}(x̄0) + ··· + C_h^{ub}(x̄0) ;
```

also valid for C_i) does not precisely bound the number of instances of $\|l\| \in e_i$, since it also accounts for visits to other equations; and (2) when computing the progression parameters of $\|l\| \in e_i$, it is not safe to consider only consecutive applications of the i-th equation, since between two applications of the i-th equation we might apply any other equations.

The above two issues can be solved as follows: (1) instead of using the ranking function $\hat{f}_C(\bar{x}_0)$, we use a function $\hat{f}_{C_i}(\bar{x}_0)$ which approximates the number of applications of the i-th equation only. Inferring such function can be done by instrumenting the *CR* with a counter that counts the number of visits to the i-th equation, and then infer an invariant that relates this counter to $\bar{x}_0$; and (2) when

inferring the progression parameters $\check{d}$ or $\langle \hat{r}, \hat{p} \rangle$, we consider the increase/decrease in two subsequent applications of the i-th equation (rather than of two consecutive ones). Again, this can be inferred by means of an appropriate invariant.

Now let us see how to use $\hat{f}_{C_i}(\bar{x}_0)$ and the progression parameters (computed as in (2) above) in order to compute a precise UB for C_i, assuming that it has at most one recursive call, i.e., $m_1 = 1$ (later we discuss this restriction): (i) we generate a worst-case *RR* $P_{C_i}(N) = E_{e_i} + P_{C_i}(N-1)$ where E_{e_i} is computed as in definitions 4.2.3 and 4.2.10, but using $\hat{f}_{C_i}(\bar{x}_0)$ instead of $\hat{f}_C(\bar{x}_0)$, and by computing the progression parameters as in point (2) above; (ii) we solve $P_{C_i}(N)$ into a closed-form solution E using *CAS*; and (iii) $C_i^{ub}(\bar{x}_0) = E[N/\hat{f}_{C_i}(\bar{x}_0)]$ is guaranteed to be a correct UB for C_i. Algorithm 3 summarizes this approach. Let us apply it to the *CR* of Example 4.3.6.

EXAMPLE 4.3.8. *Consider the following CR*

$$\begin{array}{ll} C(z,y) = 0 & \{z < 1, y < 1\} \\ C(z,y) = \|z\| + C(z',y) & \{z' = z-1, z > 0\} \\ C(z,y) = \|y\| + C(z,y') & \{y' = y-1, y > 0\} \end{array}$$

Generalizing $\|z\|$ *and* $\|y\|$ *results in* $\|z+y\|$*, which in turn leads to inferring the imprecise UB* $C^{ub}(z_0, y_0) = \|z_0 + y_0\| * \|z_0 + y_0\|$*. Using the above approach, we generate* C_1 *and* C_2 *as follows*

$$\begin{array}{ll} C_1(z,y) = 0 & \{z < 1, y < 1\} \\ C_1(z,y) = \|z\| + C_1(z',y) & \{z' = z-1, z > 0\} \\ C_1(z,y) = 0 + C_1(z,y') & \{y' = y-1, y > 0\} \end{array}$$

$$\begin{array}{ll} C_2(z,y) = 0 & \{z < 1, y < 1\} \\ C_2(z,y) = 0 + C_2(z',y) & \{z' = z-1, z > 0\} \\ C_2(z,y) = \|y\| + C_2(z,y') & \{y' = y-1, y > 0\} \end{array}$$

Observe that (1) $\|z\|$ *and* $\|y\|$ *are linearly decreasing with a progression parameter* $\check{d} = 1$*; (2) the maximization of* $\|z\|$ *and* $\|y\|$ *are* $\|z_0\|$ *and* $\|y_0\|$ *respectively; and (3) the number of applications of the first (resp. second) recursive equations of* C_1 *(resp.* C_2*) is* $\hat{f}_{C_1}(z_0, y_0) = \|z_0\|$ *(resp.* $\hat{f}_{C_2}(z_0, y_0) = \|y_0\|$*). We generate the*

RR for C_1 according to Definition 4.2.3 as follows:

$$\langle P_{C_1}(N) = \|z_0 - z_0 * 1\| + N + P_{C_1}(N-1)\rangle$$

The solution of $P_{C_1}(N)$ obtained by CAS is $P_{C_1}(N) = \frac{1}{2} * N^2 + \frac{1}{2} * N$ and the UB of C_1 according to Theorem 4.2.5 is

$${C_1}^{ub}(z_0, y_0) = \frac{1}{2} * \|z_0\| * \|z_0\| + \frac{1}{2} * \|z_0\|$$

Similarly, the RR for C_2 according to Definition 4.2.3 is as follows

$$\langle P_{C_2}(N) = \|y_0 - y_0 * 1\| + N + P_{C_2}(N-1)\rangle$$

The solution of $P_{C_2}(N)$ obtained from CAS is $P_{C_2}(N) = \frac{1}{2} * N^2 + \frac{1}{2} * N$ and the UB of C_2 according to Theorem 4.2.5 is

$${C_2}^{ub}(z_0, y_0) = \frac{1}{2} * \|y_0\| * \|y_0\| + \frac{1}{2} * \|y_0\|$$

So, the computed UB of C here is

$$C^{ub}(z_0, y_0) = \frac{1}{2} * \|z_0\| * \|z_0\| + \frac{1}{2} * \|z_0\| + \frac{1}{2} * \|y_0\| * \|y_0\| + \frac{1}{2} * \|y_0\|$$

which is more precise than $\|z_0 + y_0\| * \|z_0 + y_0\|$.

It is important to note that this last approach is correct only when $m_1 = 1$, it might infer incorrect UBs if $m_1 > 1$. Let us intuitively see why. Suppose we change the *RR* P_{C_i} such that it has $m_1 > 1$ recursive calls, then an evaluation tree for $P_{C_i}(\hat{f}_{C_i}(\bar{v}_0))$ might include less nodes than those contributed by the i-th equation in a corresponding evaluation tree for $C(\bar{v}_0)$. This is because deeper levels in an evaluation tree has more nodes, and since we have shortened the depth, by using $\hat{f}_{C_i}(\bar{x}_0)$ instead of $\hat{f}_C(\bar{x}_0)$, we might also reduce the number of such nodes. However, this approach is still practical since with $m_1 = 1$ we can handle all programs with iterative constructs and/or a single recursive call (per method).

4.4 Non-zero Base-case Cost

So far, we have considered *CRs* with only one base-case equation, and moreover, we have assumed that its contributed cost is always 0. In practice, many *CRs* that originate from real programs have several non-zero base-case equations and, besides, the cost contributed by such equations is not necessarily constant. In this section, we describe how to handle such *CRs*.

Consider the *CR* C of Figure 4.3 and assume that, instead of one base-case equation, it has n base-case equations, where the i-th equation is defined by $\langle C(\bar{x}) = e'_i, \varphi_0^i \rangle$. In order to account for these base-case equations, we first extend the worst-case *RR* P_C of Definition 4.1.1, 4.2.3 and 4.2.10 to include a generic base-case equation $\langle P_C(0) = \lambda \rangle$. Due to this extension, any solution E for P_C must involve the base-case symbol λ to account for all applications of the base-case equation.

In a second step, the base-case symbol λ in E is replaced by a cost expression e_λ that involves only $\bar{x}_0$ (i.e., it does not involve the parameter of P_C), and is greater than or equal to any value to which any $\hat{e}'_i$ is evaluated to during the evaluation of $C(\bar{x}_0)$. The cost expression e_λ is simply defined as $e_\lambda = \hat{e}'_1 \sqcup \ldots, \sqcup \hat{e}'_n$, where $\hat{e}'_i$ is the maximization of e'_i as defined in Section 2.2, and $\sqcup$ is a generalization operator of cost expressions like the one of *RR* expressions in Section 4.3.

EXAMPLE 4.4.1. *Let us replace the base-case equation* $\langle B(j,i,q) = 0, \{j \geq i\} \rangle$ *of Figure 2.2 by the equations* $\langle B(j,i,q) = \|j\|, \{j \geq i\} \rangle$ *and* $\langle B(j,i,q) = \|i\|, \{j \geq i\} \rangle$. *Maximizations of such base-case costs are, respectively,* $\hat{e}'_1 = \|i_0 + 2\|$, $\hat{e}'_2 = \|i_0\|$ *and thus their generalization is* $e_\lambda = \|i_0 + 2\|$. *Solving* P_B *of Example 4.2.6, together with a base-case equation* $P_B(0) = \lambda$, *results in:*

$$P_B(N) = \|q_0 + j_0 - 1\| * N + \|i_0 - j_0\| * N + \frac{N}{2} - \frac{N^2}{2} + \lambda$$

Then, replacing N *by the ranking function* $\|i_0 - j_0\|$ *and* λ *by* e_λ *we get*

$$B^{ub}(j_0, i_0, q_0) = \|q_0 + j_0 - 1\| * \|i_0 - j_0\| + \frac{\|i_0 - j_0\|}{2} * (\|i_0 - j_0\| + 1) + \|i_0 + 2\| \,.$$

4.5 Concluding Remarks

We have presented a practical and precise approach for inferring UBs on *CRs*. When considering *CRs* with a single recursive equation, in practice, our approach achieves an optimal precision. As regards *CRs* with multiple recursive equations, we have presented a solution which is effective in practice. Note that, although we have concentrated on arithmetic and geometric behavior of $\|.\|$ expression, our techniques are not limited to such behavior, and can be adapted to any behavior that can be modeled with sequences.

It is important to point out that in some cases the output of *CAS*, when solving a *RR*, might not comply with the grammar of cost expressions as specified in Section 3.1. Concretely, after normalization, it might include sub-expressions of the form $-e$ where e is a multiplication of basic cost expression. Converting them to valid cost expressions can be simply done by removing such negative parts and obviously still have a sound UB. In practice, these negative parts are asymptotically negligible when compared to the other parts of the UB, and thus, removing them does not significantly affect the precision. In addition, in many cases, the negative parts can be rewritten in order to *push* the minus sign inside a $\|.\|$ expression, e.g., $\|l_1\| - \|l_2\|$ is over-approximated by $\|l_1 - l_2\|$.

Chapter 5

Inference of Precise Lower Bounds

In this chapter we aim at applying the approach from Chapter 4 in order to infer *lower bounds*, i.e., under-approximations of the best-case cost. In addition to the traditional applications for performance debugging, program optimization and verification, such LBs are useful in granularity analysis to decide if tasks should be executed in parallel. This is because the parallel execution of a task incurs various overheads, and therefore the LB cost of the task can be useful to decide if it is worth executing it concurrently as a separate task. Due in part to the difficulty of inferring under-approximations, a general framework for inferring LBs from *CR* does not exist. When trying to adapt the UB framework of [5] to LB, we only obtain trivial bounds. This is because the minimization of the cost expression accumulated along the execution is in most cases zero and, hence, by assuming it for all executions we would obtain a trivial (zero) LB. In our framework, even if the minimal cost could be zero, since we do not assume it for all iterations, but rather only for the first one, the resulting LB is not trivial. In what follows, in Section 5.1 we develop our method for inferring LBs for *CRs* with single recursive equation as the one of Figure 4.1, and, in Section 5.2 we handle *CRs* with multiple recursive equations as the one of Figure 4.3. Finally, in Section 5.3 we finish with some concluding remarks.

5.1 Cost Relations with Single Recursive Equation

The basic ideas for inferring LBs are dual to those described in Section 4 for inferring UBs, i.e., they are based on simulating the behavior of $\|.\|$ expressions with corresponding linear or geometric sequences. For example, if a given $\|l\| \in e$ is linearly increasing with a progression parameter $\check{d} \geq 0$, then it is simulated with an arithmetic sequence that starts from the minimum value to which $\|l\|$ can be evaluated, and increases in each step by $\check{d}$. In addition, the number of elements that we consider in such sequence is an under-approximation of the length of any chain of calls when evaluating $C(\bar{x}_0)$. In what follows, we develop our approach for inferring LBs on the *CR* of Figure 4.1 as follows: we first describe how to infer a lower-bound on the length of any chain of calls; then we describe how to infer the minimum value to which a $\|l\|$ expression can be evaluated; and finally we use this information in order to build a best-case *RR* that under-approximates the best-case cost of the *CR* C.

The following definition provides a practical algorithm for inferring an under-approximation on the length of any chain of calls when evaluating $C(\bar{x}_0)$ using the *CR* of Figure 4.3, which is also applicable for the *CR* of Figure 4.1.

DEFINITION 5.1.1. *Given the CR of Figure 4.3, a lower-bound on the length of any chain of calls during the evaluation of* $C(\bar{x}_0)$ *denoted as* $\check{f}_C(\bar{x}_0)$ *is computed as follows:*

1. Instrumentation*: Replace each head* $C(\bar{x})$ *by* $C(\bar{x}, lb)$*, each recursive call* $C(\bar{x}_j)$ *by* $C(\bar{x}_j, lb')$*, and add* $\{lb' = lb + 1\}$ *to each* φ_i*;*

2. Invariant*: Infer an invariant* $\langle C(\bar{x}_0, 0) \rightsquigarrow C(\bar{x}, lb), \Psi \rangle$ *for the new CR, such that the linear constraints* Ψ *hold between (the variables of) the initial call* $C(\bar{x}_0, 0)$ *and any recursive call* $C(\bar{x}, lb)$*; and*

3. Synthesis*: compute* l *as the result of minimizing* lb *w.r.t* $\Psi \wedge \varphi_0$ *and the parameters* $\bar{x}_0$*, using parametric integer programming; or alternatively, compute* l *by syntactically looking for* $lb \geq l$ *in* $\bar{\exists}\bar{x}_0 \cup \{lb\}.\ \Psi \wedge \varphi_0$*.*

Then, $\check{f}_C(\bar{x}_0) = \|l\|$.

Let us explain intuitively the different steps of the above definition. In step 1, the *CR* C is instrumented with an extra argument lb which computes the length of the corresponding chain of calls, when starting the evaluation from $C(\bar{x}_0, 0)$. This instrumentation reduces the problem of finding a lower-bound on the length of any chain of calls to the problem of finding a (symbolic) minimum value for lb for which the base-case equation is applicable (i.e., the chain of calls terminates). This is exactly what steps 2 and 3 do. In 2, we infer an invariant Ψ on the arguments of any call $C(\bar{x}, lb)$ encountered during the evaluation of $C(\bar{x}_0, 0)$. This is done exactly as for the invariant described in Section 3.1 when maximizing cost expressions. In 3, from all states described by Ψ, we are interested only in those in which the base-case equation is applicable, i.e., in $\Psi \wedge \varphi_0$. Then, within this set of states, we take the minimum value l (in terms $\bar{x}_0$) of lb. Such l is the lower-bound we are interested in.

COROLLARY 5.1.2. *Function* $\check{f}_C(\bar{x}_0)$ *of Definition 5.1.1 is a* lower-bound on the length *of any chain of calls during the evaluation of* $C(\bar{x}_0)$.

EXAMPLE 5.1.3. *Applying step 1 of Definition 5.1.1 on the CR B of Example 4.2.2 results in*

$$\begin{array}{lcll} \langle B(j,i,q,lb) & = & 0 & \{j \geq i\}\rangle \\ \langle B(j,i,q,lb) & = & \|q{+}j\|{+}B(j',i,q,lb') & \{j < i, j{+}1 \leq j' \leq j{+}3, lb' = lb+1\}\rangle \end{array}$$

The invariant for this CR is $\Psi = \{j - j_0 - lb \geq 0, j_0 + 3{*}lb - j \geq 0, i = i_0, q = q_0\}$. *Projecting* $\Psi \wedge \{j \geq i\}$ *on* $\langle j_0, i_0, q_0, lb\rangle$ *results in* $\{j_0 + 3{*}lb - i_0 \geq 0\}$ *which implies* $lb \geq \frac{(i_0 - j_0)}{3}$, *from which we can synthesize* $\check{f}_B(j_0, i_0, q_0) = \|\frac{i_0 - j_0}{3}\|$. *Similarly, for CRs C and A of Figure 2.2 we obtain* $\check{f}_C(k_0, j_0, q_0) = \|q_0 + j_0 - k_0\|$ *and* $\check{f}_A(i_0, q_0) = \|\frac{q_0 - i_0}{4}\|$.

Inferring the minimum value to which $\|l\| \in e$ can be evaluated is done in a dual way to that of inferring the maximum value to which it can be evaluated (see Section 2.2). Namely, using the invariant Ψ of Definition 5.1.1, we syntactically look for an expression $\xi \geq \check{l}$ in $\bar{\exists}\bar{x}_0 \cup \{\xi\}$. $\Psi \wedge \varphi_1 \wedge \xi = l$ where ξ is a new variable. As in the case of maximization, the advantage of this approach is that

it can be implemented using any tool for manipulation of linear constraints such as PPL [11]. Alternatively, we can also use parametric integer programming [40] in order to minimize ξ w.r.t. $\Psi \wedge \varphi_1 \wedge \xi = l$ and the parameters $\bar{x}_0$.

Now that we have all ingredients for under-approximating the behavior of a given $\|l\| \in e$. In the following definition, we generate the best-case *RR* P_C of *CR* C. Let us first explain the idea intuitively. Let $\|l_1\|, \ldots, \|l_\kappa\|$ be the first $\kappa \leq \check{f}_C(\bar{x}_0)$ elements contributed by a given $\|l\| \in e$ along a chain of calls, and assume that $l_i \geq 0$ for all $1 \leq i \leq \kappa$. If $\|l\|$ is linearly increasing (resp. decreasing) with a progression parameter $\check{d} > 0$, then the elements of the sequence $\{\ell_1 = \|\check{l}\|, \ell_i = \ell_{i-1} + \check{d}\}$ satisfy $\ell_i \leq \|l_i\|$ (resp. $\ell_i \leq \|l_{\kappa-i+1}\|$). Similarly, if $\|l\|$ is geometrically increasing (resp. decreasing) with progression parameters $\check{r}$ and $\check{p}$, then the elements of the sequence $\{\ell_1 = \|\check{l}\|, \ell_i = \check{r} * \ell_{i-1} + \check{p}\}$ satisfy $\ell_i \leq \|l_i\|$ (resp. $\ell_i \leq \|l_{\kappa-i+1}\|$). The following definition uses these sequences in order to under-approximate the behavior of $\|l\|$. Note that the condition $l_i \geq 0$ is essential, otherwise, the sequences ℓ_i is not a sound under-approximation.

DEFINITION 5.1.4. *Let C be the CR of Figure 4.1, and $\check{f}_C(\bar{x}_0)$ a lower-bound function on the length of any chain of calls generated during the evaluation of $C(\bar{x}_0)$. Then, the best-case RR of C is $P_C(N) = \check{E}_e + m * P_C(N-1)$ where $\check{E}_e$ is obtained from e by replacing each $\|l\| \in e$ by l_{RR} such that $\check{l} \geq 0$ where:*

1. $l_{RR} \equiv \|\check{l}\| + (\check{f}_C(\bar{x}_0) - N) * \check{d}$, *if it is linearly increasing;*
2. $l_{RR} \equiv \|\check{l}\| + (N-1) * \check{d}$, *if it is linearly decreasing;*
3. $l_{RR} \equiv \check{r}^{(\check{f}_C(\bar{x}_0)-N)} * \|\check{l}\| + \check{p} * \check{S}(\check{f}_C(\bar{x}_0) - N)$, *if it is geometrically increasing;*
4. $l_{RR} \equiv \check{r}^{(N-1)} * \|\check{l}\| + \check{p} * \check{S}(N-1)$, *if it is geometrically decreasing;*
5. $l_{RR} \equiv \|\check{l}\|$, *otherwise.*

where $\check{S}(i) = \frac{\check{r}^i - 1}{\check{r} - 1}$

THEOREM 5.1.5. *If E is a solution for $P_C(N)$ of Definition 5.1.4, then $C^{lb}(\bar{x}_0) = E[N/\check{f}_C(\bar{x}_0)]$ is a LB for $C(\bar{x}_0)$.*

Proof. In order to prove the above theorem, it is enough to prove that if the costs contributed by $C(\bar{x}_0)$ and $P_C(N)$ along any corresponding chain of calls are $e_1, \cdots, e_{\kappa_0}$ and $\check{E}_{e_1}, \cdots, \check{E}_{e_\kappa}$ respectively, it holds that $\check{E}_{e_i} \leq e_i$ for all $1 \leq i \leq \kappa$ and $\kappa \leq \kappa_0$. Since N is instantiated by $\check{f}_C(\bar{x}_0)$ to the solution E of $P_C(N)$, any chain of calls of $P_C(N)$ is exactly $\check{f}_C(\bar{x}_0)$ (i.e. $\kappa = \check{f}_C(\bar{x}_0)$). According to Corollary 5.1.2, $\check{f}_C(\bar{x}_0)$ is the *lower-bound* on the length of any chain of $C(\bar{x}_0)$ and hence $\kappa \leq \kappa_0$ holds in general. Again, since cost expression e (and hence its corresponding RR expression $\check{E}_e$) follows the monotonicity property, in order to prove $\check{E}_{e_i} \leq e_i$, it is enough to prove the relation for their $\|.\|$ sub-component. That means, if $\|l_1\|, \cdots, \|l_{\kappa_0}\|$ are instances of $\|l\| \in e$ in the chain of calls $e_1, \cdots, e_{\kappa_0}$ and $L_1, \cdots, L_{\check{f}_C(\bar{x}_0)}$ are the instances of the replacements of $\|l\|$ in $\check{E}_e$ according to Definition 5.1.4 along the chain of calls $\check{E}_{e_1}, \cdots, \check{E}_{e_\kappa}$, it is enough to prove that $L_i \leq \|l_i\|$ for all $1 \leq i \leq \check{f}_C(\bar{x}_0)$.

Base Case: The comparison of L_1 and $\|l_1\|$ are done by the following case analysis as done for the replacement of $\|l\|$ in Definition 5.1.4.

1. We obtain L_1 when $N = \check{f}_C(\bar{x}_0)$ since $\|l\|$ is linearly increasing. $L_1 = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0)) * \check{d} = \|\check{l}\| \leq \|l_1\|$.

2. In this case $N = 1$ since $\|l\|$ is linearly decreasing and $L_1 = \|\check{l}\| + (1-1) * \check{d} = \|\check{l}\| \leq \|l_1\|$.

3. Here, $N = \check{f}_C(\bar{x}_0)$ and $L_1 = \check{r}^{(\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0))} * \|\check{l}\| + \check{p} * \check{S}(\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0)) = \|\check{l}\| \leq \|l_1\|$ since $\check{S}(0) = 0$.

4. Here, $N = 1$ and $L_1 = \check{r}^{(1-1)} * \|\check{l}\| + \check{p} * \check{S}(1-1) = \|\check{l}\| \leq \|l_1\|$.

5. $\|\check{l}\| \leq \|l_1\|$.

Inductive Case: Here we assume that $L_i \leq \|l_i\|$ and will prove that $L_{i+1} \leq \|l_{i+1}\|$. We do the similar case analysis.

1. For L_i, we have $N = \check{f}_C(\bar{x}_0) - i + 1$. So, $L_i = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - (\check{f}_C(\bar{x}_0) - i + 1)) * \check{d} = \|\check{l}\| + (i-1) * \check{d}$. Then $L_{i+1} = \|\check{l}\| + (\check{f}_C(\bar{x}_0) - \check{f}_C(\bar{x}_0) + i) * \check{d} =$

$P_A(N) = \frac{1}{2} * (\|\frac{i_0}{3}\| + (\|\frac{q_0 - i_0}{4}\| - N) * \frac{2}{3}) * (\|\frac{i_0}{3}\| + (\|\frac{q_0 - i_0}{4}\| - N) * \frac{2}{3} + 2 * \|q_0 - \frac{1}{2}\|) + P_A(N-1)$
$P_B(N) = \|q_0 + j_0\| + (\|\frac{i_0 - j_0}{3}\| - N) * 1 + P_B(N-1)$
$P_C(N) = 1 + P_C(N-1)$

Figure 5.1: Best-Case *RRs* automatically obtained from *CRs* in Fig. 2.2.

$\|\check{l}\| + i * \check{d} = \|\check{l}\| + (i-1) * \check{d} + \check{d} = L_i + \check{d} \leq \|l_i\| + \check{d} \leq \|l_{i+1}\|$ since $\check{d}$ is the minimum distance of $\|l\|$ and $\check{l} \geq 0$.

2. Here, we have $N = i$ for L_i and $N = i + 1$ for L_{i+1}. Then the proof is similar to the proof of case (1).

3. For L_i and L_{i+1}, $N = \check{f}_C(\bar{x}_0) - i + 1$ and $N = \check{f}_C(\bar{x}_0) - i$ respectively. Thus we obtain $L_i = \check{r}^{(i-1)} * \|\check{l}\| + \check{p} * \check{S}(i-1)$ and $L_{i+1} = \check{r}^i * \|\check{l}\| + \check{p} * \check{S}(i)$. We also have $\check{S}(i) = \frac{\check{r}^i - 1}{\check{r} - 1} = \check{r} * \frac{\check{r}^{i-1} - 1}{\check{r} - 1} + 1 = \check{r} * \check{S}(i-1) + 1$. Then $L_{i+1} = \check{r}^i * \|\check{l}\| + \check{p} * \check{S}(i) = \check{r} * (\check{r}^{(i-1)} * \|\check{l}\| + \check{p} * \check{S}(i-1)) + p = \check{r} * L_i + \check{p} \leq \check{r} * \|l_i\| + \check{p} \leq \|l_{i+1}\|$ [since $\check{l} \geq 0$ and $\|l\|$ has the geometric progression behavior as defined in Definition 4.2.8].

4. For L_i and L_{i+1}, $N = i$ and $N = i + 1$ respectively and the proof is similar to the proof of case (3).

5. $\|\check{l}\| \leq \|l_{i+1}\|$.

□

An algorithm that summarizes the above approach can be derived in a very similar way to Algorithm 1, simply by considering the dual notions to $\hat{l}$ and $\hat{f}_C(\bar{x}_0)$.

EXAMPLE 5.1.6. *Consider again the LBs on the length of chains of calls as described in Example 5.1.3. Since $C(k_0, j_0, q_0)$ accumulates a constant cost 1, its LB cost is $\|q_0 + j_0 - k_0\|$. We now replace the call $C(0, j, q)$ in B by its LB $\|q + j\|$ and obtain the following recursive equation:*

$$\langle B(j, i, q) = \|q + j\| + B(j', i, q), \{j + 1 \leq i, j + 1 \leq j' \leq j + 3\} \rangle$$

Notice the need of the soundness requirement in Definition 5.1.4, i.e., $q_0 + j_0 \geq 0$ *where* $q_0 + j_0$ *is the minimization of* $q + j$ *for any call to* $B(j_0, i_0, q_0)$*. For example, when evaluating* $B(-5, 5, 0)$ *the first* 5 *instances of* $\|q + j\|$ *are zero since they correspond to* $\|-5\|, \ldots, \|-1\|$*. Therefore, it would be incorrect to start accumulating from 0 with a difference 1 at each iteration. However, in the context of the CRs of Figure 2.2, it is guaranteed that* $q_0 + j_0 \geq 0$ *(since it is always called with* $j \geq 0$ *and* $q \geq 0$*). Using Definition 5.1.4, we generate the* best-case *RR* P_B *depicted in Figure 5.1 which is solved by CAS to*

$$P_B(N) = \|q_0 + j_0\| * N + \|\frac{i_0 - j_0}{3}\| * N - \frac{N^2}{2} - \frac{N}{2}$$

Then, according to Theorem 5.1.5

$$B^{lb}(j_0, i_0, q_0) = \frac{1}{2} * \|\frac{i_0 - j_0}{3}\| * (\|\frac{i_0 - j_0}{3}\| + 2 * \|q_0 + j_0 - \frac{1}{2}\|)$$

Substituting this LB in the CR A of Figure 2.2 results in the CR

$$\langle A(i, q) = \frac{1}{2} * \|\frac{i}{3}\| * (\|\frac{i}{3}\| + 2 * \|q - \frac{1}{2}\|) + A(i', q), \{i + 1 \leq q, i + 2 \leq i' \leq i + 4\}\rangle$$

In this CR, the expression $2 * \|q - \frac{1}{2}\|$ *is constant, while* $\|\frac{i}{3}\|$ *has an increasing linear progression behavior with* $\check{d} = \frac{2}{3}$*. According to Definition 5.1.4, the generated* best-case *RR* P_A *is depicted in Figure 5.1 which is solved using CAS to*

$$\begin{aligned} P_A(N) \;=\; & \tfrac{N}{54} * (4 * N^2 + 6 * N + 18 * \|\tfrac{i_0}{3}\| * (N - 1) + 18 * \|q_0 - \tfrac{1}{2}\| * (N - 1) + \\ & 27 * \|\tfrac{i_0}{3}\| * \|\tfrac{i_0}{3}\| + 54 * \|\tfrac{i_0}{3}\| * \|q_0 - \tfrac{1}{2}\| - 12 * \|\tfrac{q_0 - i_0}{4}\| + 2) \end{aligned}$$

Then, according to Theorem 5.1.5, i.e., substituting N *by* $\|\frac{q_0 - i_0}{4}\|$*, we obtain*

$$\begin{aligned} A^{lb}(i_0, q_0) \;=\; & \tfrac{1}{54} * \|\tfrac{q_0 - i_0}{4}\| * (4 * \|\tfrac{q_0 - i_0}{4}\| * \|\tfrac{q_0 - i_0}{4}\| + 6 * \|\tfrac{q_0 - i_0}{4}\| + 18 * \|\tfrac{i_0}{3}\| \\ & * (\|\tfrac{q_0 - i_0}{4}\| - 1) + 18 * \|q_0 - \tfrac{1}{2}\| * (\|\tfrac{q_0 - i_0}{4}\| - 1) + 27 * \|\tfrac{i_0}{3}\| * \|\tfrac{i_0}{3}\| \\ & + 54 * \|\tfrac{i_0}{3}\| * \|q_0 - \tfrac{1}{2}\| - 12 * \|\tfrac{q_0 - i_0}{4}\| + 2) \end{aligned}$$

Finally, the LB of $F(q_0)$ *is*

$$\begin{aligned} F^{lb}(q_0) \;=\; & \tfrac{1}{54} * \|\tfrac{q_0}{4}\| * (4 * \|\tfrac{q_0}{4}\| * \|\tfrac{q_0}{4}\| + 6 * \|\tfrac{q_0}{4}\| + 18 * \|q_0 - \tfrac{1}{2}\| * \\ & (\|\tfrac{q_0}{4}\| - 1) - 12 * \|\tfrac{q_0}{4}\| + 2)\,. \end{aligned}$$

5.2 Cost Relations with Multiple Recursive Equations

We infer LBs for *CRs* with multiple recursive equations in a dual way to the inference of UBs, namely: we first try to generate a best-case *CR* $\check{C}$, for the multiple recursive *CRs* C in Figure 4.3, in a similar way to the worst-case *CR* $\hat{C}$. If this is not possible (or not precise enough) and $m_1 = 1$ (i.e. we have at most one recursive call) then we can use the second approach, in which we compute a LB for each C_i (the projection of C on the i-th equation), and then sum all these LBs into a sound LB for C.

DEFINITION 5.2.1. *We say that* $\langle \check{C}(\bar{x}) = e + \check{C}(\bar{x}_1) + \cdots + \check{C}(\bar{x}_{m_1}), \varphi \rangle$ *is a best-case CR for the CR* C *of Figure 4.3, if for any valuation* $\bar{v}$ *it holds that*

$$min(\{sum(T, \check{f}_C(\bar{v})) \mid T \in \mathcal{T}(\check{C}(\bar{v}))\} \leq min(answ(C(\bar{v})))$$

where $sum(T, \check{f}_C(\bar{v}))$ *denotes the sum of all nodes in* T *up to depth* $\check{f}_C(\bar{v})$.

CR $\check{C}$ is generated in a similar way to $\hat{C}$. The only difference is that in order to generate the cost expression e, we use a *reduction operator* $\sqcap$ instead of $\sqcup$ that appear in Definition 4.3.3. Such operator guarantees that $a \sqcap b \leq a$ and $a \sqcap b \leq b$. In practice, the reduction operator $\sqcap$ is implemented by syntactically analyzing the input cost expressions, in a similar way to the case of $\sqcup$.

THEOREM 5.2.2. *Given the CR* C *of Figure 4.3, a corresponding* $\check{f}_C(\bar{x}_0)$ *as defined in Definition 5.1.1, a best-case CR* $\check{C}$ *for* C*, and a solution* E *for the RR* $\langle P_{\check{C}}(N) = \check{E}_e + m_h * P_{\check{C}}(N-1) \rangle$*. Then* $C^{lb}(\bar{x}_0) = E[N/\check{f}_C(\bar{x}_0)]$ *is a LB for the CR* C*.*

Proof. Intuitively, since the evaluation trees of $\check{C}$ up to depth $\check{f}_C(\bar{x}_0)$ under-approximate those of C, then, by the construction of $\check{E}_e$, it is guaranteed that the evaluation tree of $\langle P_C(N) = \check{E}_e + m_h * P_C(N-1) \rangle$ up to depth $\check{f}_C(\bar{x}_0)$ under-approximates C. Therefore, if E is a solution for P_C then $E[N/\check{f}_C(\bar{x}_0)]$ is a LB for $C(\bar{x}_0)$. □

EXAMPLE 5.2.3. *Consider the CR B in Example 4.3.5. We simulate its best-case behavior by the following single recursive equation*

$$\langle \check{B}(j,i,q) = \|j\| + \check{B}(j',i,q), \{j+1 \leq i, j+1 \leq j' \leq j+3\}\rangle$$

Note that (1) $\|j\|$ under-approximates both e_1 and e_2; and (2) $\|j\|$ has an increasing linear progression behavior with progression parameter $\check{d} = 1$. Using Definition 5.1.4, we generate the following best-case RR $P_{\check{B}}$ for $\check{B}$

$$\langle P_{\check{B}}(N) = \|j_0\| + (\|\frac{i_0 - j_0}{3}\| - N) * 1 + P_{\check{B}}(N-1)\rangle$$

which is solved by CAS to

$$P_{\check{B}}(N) = N * \|\frac{i_0 - j_0}{3}\| + N * \|j_0\| - \frac{1}{2} * N^2 - \frac{1}{2} * N$$

According to Theorem 5.2.2, replacing N by $\check{f}_B(j_0, i_0, q_0) = \|\frac{i_0 - j_0}{3}\|$ results in (after simplification) the following LB for B

$$B^{lb}(j_0, i_0, q_0) = \frac{1}{2} * \|\frac{i_0 - j_0}{3}\| * \|\frac{i_0 - j_0}{3} - 1\| + \|\frac{i_0 - j_0}{3}\| * \|j_0\|$$

When the best-case *CR* approach leads to imprecise bounds, which happens when the reduction operator obtains trivial reductions (i.e., 0), we can apply the alternative method that is based on analyzing each C_i separately. Namely, we infer a LB ${C_i}^{lb}(\bar{x}_0)$ for each *CR* C_i, and then $C^{lb}(\bar{x}_0) = \sum_{i=1}^{h} {C_i}^{lb}(\bar{x}_0)$ is clearly a sound LB for C. The technical details for solving each ${C_i}^{lb}(\bar{x}_0)$ are identical to those of the UB case: (1) instead of using $\check{f}_C(\bar{x}_0)$, we should use $\check{f}_{C_i}(\bar{x}_0)$ which under-approximates the number of applications of the i-th equation. This is done by modifying Definition 5.1.1 such that it counts only the applications of the i-th equation instead of all equations; and (2) the progression parameter $\check{d}$, or $\langle \check{r}, \check{p}\rangle$ are the same as in the case of UB, i.e., we consider subsequent, rather than consecutive, applications of the i-th equation. It is important to note that this approach can be applied only when $m_1 = 1$. Algorithms that summarize the above approaches can be derived in a very similar way to algorithms 2 and 3, simply by considering the dual notions.

5.3 Concluding Remarks

We have presented a practical and precise approach for inferring LBs on *CRs*. When considering *CRs* with a single recursive equation, in practice, our approach achieves an optimal precision. As regards *CRs* with multiple recursive equations, we have presented a solution which is effective in many cases, however, it is less effective than its UB counterpart. This is expected, as indeed, the problem of inferring LBs is far more complicated than inferring UBs. It is important to note that this is the first work that attempts to automatically infer LBs for *CRs* that originate from real programs. Our approach for inferring LBs is not limited to $\|.\|$ expressions with linear and geometric behavior, but can be adapted to any behavior that can be modeled with sequences.

As in the case of UBs, the output of *CAS*, when solving a best-case *RR*, might not comply with the grammar of cost expressions as specified in Section 3. Concretely, after normalization, it might include sub-expressions of the form $-e$ where e is a multiplication of basic cost expressions. Unlike the case of UBs, for LBs it is not sound to remove such expressions as it results in a bigger one. Removing them requires changing other subexpressions in order to compensate on $-e$. E.g., $\|x\|^2 - \|x\|$ can be rewritten to $\|x - 1\| * \|x\|$.

Chapter 6

Implementation and Experiments

In this chapter we describe an implementation of the techniques developed in chapters 4 and 5, and its evaluation on some selected benchmarks. In Section 6.1 we discuss implementation issues. In Section 6.2 we describe the selected benchmarks, their respective challenges, and the UBs and LBs that we obtain and compare them to results of other available systems. We finish in Section 6.3 with some concluding remarks.

6.1 Implementation of the Cost Relations Solver

We have implemented the techniques developed in chapters 4 and 5 as a component in PUBS (Practical Upper Bound Solver) [5], which is also used as backend solver in COSTA (a COSt and Termination Analyzer for Java bytecode) [5]. This means that our solver can be used to solve (i) *CRs* that are automatically generated by COSTA from Java (bytecode) programs; or (ii) *CRs* provided by the user. In our experiments we apply it on Java programs via COSTA.

The solver is written in Prolog, and can be compiled both in CIAO Prolog [44] and SWI Prolog [81] on a LINUX based operating systems. The solver consists of the following major components:

1. A component for computing the progression behaviour of a given (possibly nonlinear) symbolic cost expression according to definitions 4.2.1 and 4.2.8;

2. A component for maximizing and minimizing (possibly nonlinear) symbolic cost expressions;

3. A component for computing the minimum number of chains of recursive calls in *CRs* as described in Definition 5.1.1;

4. A component for generating the worst-case and best-case *RRs* for computing UBs and LBs respectively; and

5. A component for communicating with external *RRs* solvers in order to solve the corresponding worst-case and best-case *RRs*, in particular with MAXIMA [60] and PURRS [13].

In addition, the solver relies on PUBS [5] for computing linear ranking functions and invariants, and uses the Parma Polyhedra Library (PPL) [11] for manipulating linear constraints – such as checking for consistency, eliminating variables, and solving linear programming problems.

The tool has both a command-line and a web interface. Using it from a command-line is done as follow

pubs_shell -file ProgramFile -computebound {ubseries,lbseries}

where Programfile is an ASCII text file that includes the corresponding *CRs*, and the options ubseries and lbseries indicate if the user wants to compute UBs or LBs respectively. The web interface is available at `http://costa.ls.fi.upm.es/pubs`. The user can provide a *CRs*, select an appropriate setting, and ask the solver to compute UBs or LBs for each *CR*. A screenshot is provided in Figure 6.1. Alternatively, the solver can be used via the web interface of COSTA which is available at `http://costa.ls.fi.upm.es/costa`. In this case the user is asked to provide a Java (bytecode) program, then COSTA automatically generates the corresponding *CRs* and passes them to the solver.

6.2 Experiments

As benchmarks, we use classical challenging examples from complexity analysis and numerical methods. We avoid examples in which all iterations of a loop

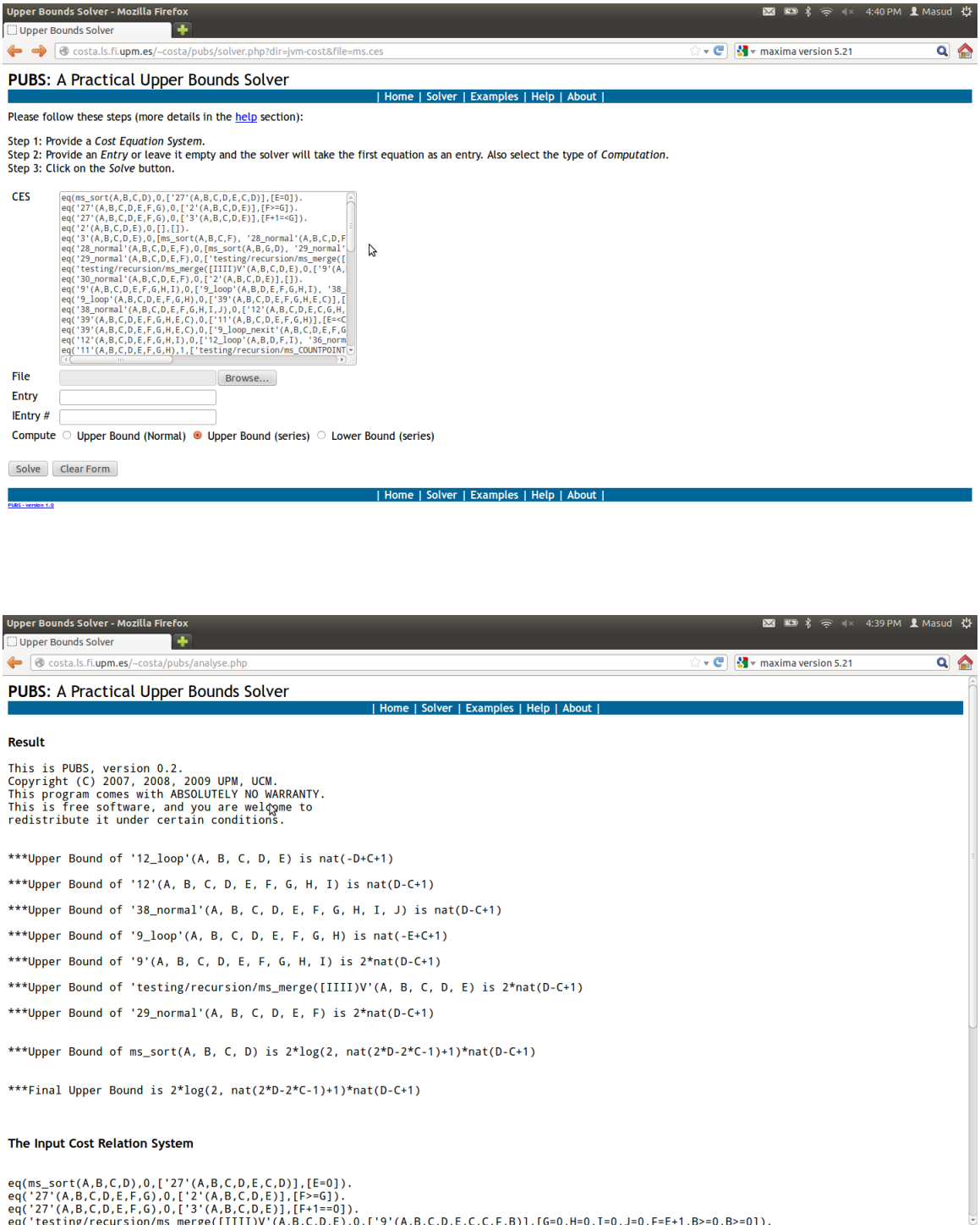
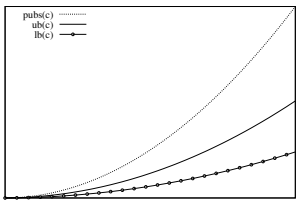
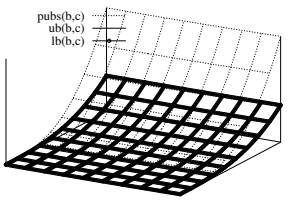


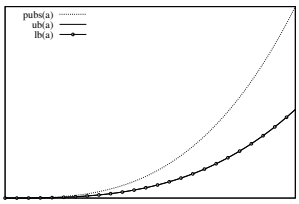
Figure 6.1: Web interface of the cost relations solver.



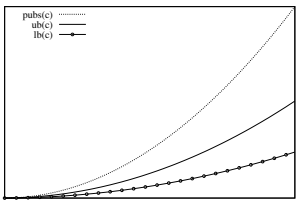
DetEval



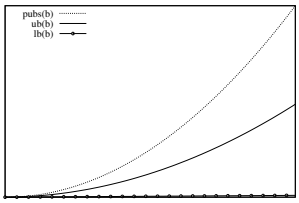
LinEqSolve



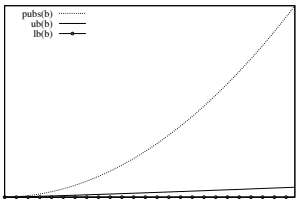
MatrixInv



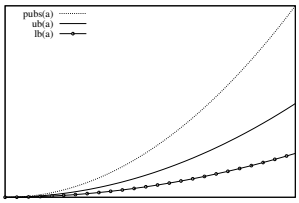
MatrixSort



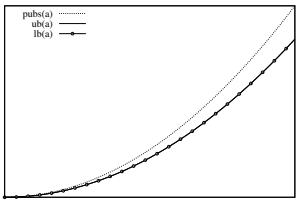
InsertSort



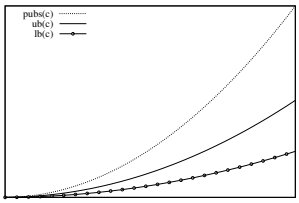
MergeSort



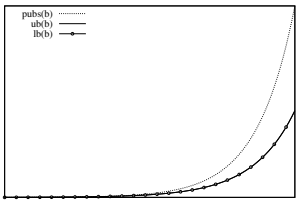
SelectSort



PascalTriangle



BubbleSort



NestedRecIter

Figure 6.2: Graphical comparisions of UBs and LBs.

(or recursive calls) have the same worst-case or best-case costs, since in such cases our method infers the same bounds as [5]. Our benchmarks are written in Java, and we analyze them using COSTA which, for each benchmark, it first generates a corresponding *CRs* and then it solves it into closed-form UBs and LBs using our solver. We use a cost model that counts the "*number of executed (bytecode) instructions*". The (complete) source code and the generated *CRs* for each benchmark are available at `http://costa.ls.fi.upm.es/pubs`. Next, for each benchmark: we present the interesting parts of the source code; discuss its interesting features for cost analysis; show the UB and LB that we infer using our solver and compare our UB to that inferred by [5]. In order to facilitate the comparison of the UBs and LBs of each benchmark, we also provide the graphical representations for each benchmark in Figure 6.2 where *pubs*, *ub*, and *lb* represent respectively the plots for the UBs obtained by [5], our UBs and LBs.

DetEval. This program computes the determinant of a matrix. The interesting part is a method gaussian which converts a given matrix into an upper triangular matrix. The code of this method is depicted in Figure 6.3. The interesting part of this method is the loop that starts at line 19 and ends at line 39. Note that the value of the outer loop counter j affects the number of iterations of the inner loops, and thus the cost contributed by the inner loops is different in each iteration of the outer loop. For this benchmark we obtain the following bounds

(A) $24 \cdot \|a-1\|^3+36 \cdot \|a-1\|^2+18 \cdot \|a\|^2+30 \cdot \|a\| \cdot \|a-1\|+35 \cdot \|a-1\|+72 \cdot \|a\|+54$	1776
(B) $8 \cdot \|a-1\|^3+18 \cdot \|a\|^2+45 \cdot \|a-1\|^2+72 \cdot \|a\|+102 \cdot \|a-1\|+54$	3272
(C) $8 \cdot \|a-1\|^3+16 \cdot \|a\|^2+43 \cdot \|a-1\|^2+55 \cdot \|a\|+96 \cdot \|a-1\|+54$	2568

In the first column: (A) the UB obtained by [5]; (B) the UB obtained by our approach; and (C) our LB. The second column is the runtime in milliseconds. This notation will be used for all benchmarks that follows and thus we will not explain it again later. Looking at the corresponding graph in Figure 6.2, we can see that our UB is more precise than that of [5] and our LB is very tight.

LinEqSolve. This program solves a set of linear equations given as a matrix. The main method solve of this program is depicted in Figure 6.4. Note that it

```
1 public static void gaussian(double a[][],
2   int index[]) {
3
4   int n=index.length;
5   double c[]=new double[n];
6
7   for (int i=0; i<n; ++i) index[i]=i;
8
9   for (int i=0; i<n; ++i) {
10    double c1=0;
11    for (int j=0; j<n; ++j) {
12      double c0=Math.abs(a[i][j]);
13      if (c0 > c1) c1=c0;
14    }
15    c[i]=c1;
16  }
17
18  int k=0;
19  for (int j=0; j<n-1; ++j) {
20    double pi1=0;
21    for (int i=j; i<n; ++i) {
22      double pi0=Math.abs(a[index[i]][j]);
23      pi0 /= c[index[i]];
24      if (pi0 > pi1) {
25        pi1=pi0;
26        k=i;
27      }
28    }
29
30    int itmp=index[j];
31    index[j]=index[k];
32    index[k]=itmp;
33    for (int i=j+1; i<n; ++i) {
34      double pj=a[index[i]][j]/a[index[j]][j];
35      a[index[i]][j]=pj;
36      for (int l=j+1; l<n; ++l)
37        a[index[i]][l] -= pj*a[index[j]][l];
38    }
39  }
40}
```

Figure 6.3: Source code of the `DetEval` program.

calls (at line 6) method gaussian of Figure 6.3. Apart from the call to method gaussian, the nested loops at lines 7-11 and 14-20 are challenging for cost analysis since the number of iterations of the inner loops depends on the counters of the outer loops. For this benchmark we obtain the following bounds

(A) $24 \cdot \|c-1\|^3+18 \cdot \|c\|^2+36 \cdot \|c-1\|^2+30 \cdot \|c-1\| \cdot \|c\|+35 \cdot \|c-1\|+25 \cdot \|c\|+48 \cdot \|b-1\|^2+46 \cdot \|b-1\|+74$	1870
(B) $8 \cdot \|c-1\|^3+18 \cdot \|c\|^2+45 \cdot \|c-1\|^2+25 \cdot \|c\|+102 \cdot \|c-1\|+24 \cdot \|b-1\|^2+92 \cdot \|b-1\|+74$	3480
(C) $8 \cdot \|c-1\|^3+16 \cdot \|c\|^2+43 \cdot \|c-1\|^2+25 \cdot \|c\|+96 \cdot \|c-1\|+24 \cdot \|b-1\|^2+92 \cdot \|b-1\|+74$	2796

```
1 public static double[] solve(double a[][],
2   double b[], int ind[]) {
3
4   int n = b.length;
5   double x[] = new double[n];
6   gaussian(a, index);
7   for(int i=0; i<n−1; ++i) {
8     for(int j =i+1; j<n; ++j) {
9       b[ind[j]] −= a[ind[j]][i]∗b[ind[i]];
10    }
11  }
12  x[n−1] = b[ind[n−1]]/a[ind[n−1]][n−1];
13
14  for (int i=n−2; i>=0; −−i) {
15    x[i] = b[ind[i]];
16    for (int j=i+1; j<n; ++j) {
17      x[i] −= a[ind[i]][j]∗x[j];
18    }
19    x[i] /= a[ind[i]][i];
20  }
21  return x;
22 }
```

Figure 6.4: Source code of the `LinEqSolve` program.

Looking at the corresponding graph in Figure 6.2, we can see that our UB is more precise than that of [5] and our LB is very tight.

```
1  public double[][] invert(double a[][]) {
2
3    int n = a.length;
4    double x[][] = new double[n][n];
5    double b[][] = new double[n][n];
6    int ind[] = new int[n];
7
8    for (int i=0; i<n; ++i) b[i][i] = 1;
9    gaussian(a, ind);
10   for (int i=0; i<n−1; ++i)
11     for (int j=i+1; j<n; ++j)
12       for (int k=0; k<n; ++k)
13         b[ind[j]][k] −=
14           a[ind[j]][i]∗b[ind[i]][k];
15   for (int i=0; i<n; ++i) {
16     x[n−1][i] =
17       b[ind[n−1]][i]/a[ind[n−1]][n−1];
18     for (int j=n−2; j>=0; −−j) {
19       x[j][i] = b[ind[j]][i];
20       for (int k=j+1; k<n; ++k) {
21         x[j][i] −= a[ind[j]][k]∗x[k][i];
22       }
23       x[j][i] /= a[ind[j]][j];
24     }
25   }
26
27   return x;
28 }
```

Figure 6.5: Source code of the `MatrixInverse` program.

MatrixInverse. This program computes the inverse of a matrix. The source code is depicted in Figure 6.5. The cost analysis of this program is challenging since it also has nested loops (at lines 10-14 and 15-25) in which the number of iterations of the inner loops depends on the counter of the outer loop. For this benchmark we obtain the following bounds

(A) $24 \cdot \|a{-}1\|^3{+}56 \cdot \|a\| \cdot \|a{-}1\|^2{+}18 \cdot \|a\|^2{+}46 \cdot \|a{-}1\|^2{+}75 \cdot \|a\|{+}68 \cdot \|a\| \cdot \|a{-}1\| + 49 \cdot \|a{-}1\|{+}62$	3617
(B) $8 \cdot \|a{-}1\|^3{+}28 \cdot \|a\| \cdot \|a{-}1\|^2{+}18 \cdot \|a\|^2{+}50 \cdot \|a{-}1\|^2{+}92 \cdot \|a\| \cdot \|a{-}1\|{+}75 \cdot \|a\| + 121 \cdot \|a{-}1\|{+}62$	4620
(C) $8 \cdot \|a{-}1\|^3{+}28 \cdot \|a\| \cdot \|a{-}1\|^2{+}16 \cdot \|a\|^2{+}48 \cdot \|a{-}1\|^2{+}92 \cdot \|a\| \cdot \|a{-}1\|{+}75 \cdot \|a\| + 115 \cdot \|a{-}1\|{+}62$	3792

Looking at the corresponding graph in Figure 6.2, we can see that our UB is more precise than that of [5] and our LB is very tight.

InsertSort. This program implements the insertion sort algorithm as depicted in Figure 6.6. This example is interesting from the point of view of complexity analysis. We get the worst-case cost when the array is sorted in a reversed order and the best-case cost when the array is already sorted. If the array is sorted, the inner while loop will not be executed and hence the best-case cost will be linear in terms of its input arguments. In case of worst-case cost, there is a precision issue as the cost of the inner while loop is different for different values of i. For this benchmark we obtain the following bounds

(A) $19 \cdot \|b{-}1\|^2 + 25 \cdot \|b{-}1\| + 7$	110
(B) $\frac{19}{2} \cdot \|b{-}1\|^2 + \frac{69}{2} \cdot \|b{-}1\| + 7$	170
(C) $18 \cdot \|b{-}1\| + 7$	110

Looking at the corresponding graph in Figure 6.2, we can see the our UB is more precise than that of [5] and we obtain linear LB.

```
1 void insertSort(int[] arr,int i_0, int length) {
2  int i, j, newValue;
3  for (i = i_0; i < length; i++) {
4      newValue = arr[i];
5      j = i;
6      while (j > 0 && arr[j − 1] > newValue) {
7              arr[j] = arr[j − 1];
8              j−−;
9      }
10     arr[j] = newValue;
11   }
12 }
13
14 public void matrixsort(int [][] A, int size) {
15  for(int i=0; i<size; i++)
16     insertsort(A,i,size);
17 }
```

Figure 6.6: The source code of the `InsertSort` and `MatrixSort` programs.

MatrixSort. This program sorts the rows in the upper triangle of a matrix. The source code is depicted in Figure 6.6. Note that method insertSort is called for sorting each row. The important feature of this examples is that the call to insertSort accumulates different worst-case and best-case cost in each iteration of the loop. For this benchmark we obtain the following bounds

(A) $25 \cdot \|b\|^2 \cdot \|b-1\| + 30 \cdot \|b\|^2 + 16 \cdot \|b\| + 6$	130
(B) $\frac{25}{3} \cdot \|b\|^3 + 15 \cdot \|b\|^2 + \frac{68}{3} \cdot \|b\| + 6$	200
(C) $\frac{21}{2} \cdot \|b\|^2 + \frac{53}{2} \cdot \|b\| + 6$	60

Looking at the corresponding graph in Figure 6.2, we can see that our UB is more precise than that of [5] and our LB is very tight. Note that the LB is quadratic while the UB is cubic.

```
1 public void bubbleSort(int arr[], int n) {
2   for(int j=0;j<n;j++) {
3     for (int i = 0; i < n − j; i++) {
4       if (arr[i] > arr[i + 1]) {
5         int tmp = arr[i];
6         arr[i] = arr[i + 1];
7         arr[i + 1] = tmp;
8       }
9     }
10  }
11 }
```

```
1 public static void selectionSort(int[] arr) {
2   for (int i=0; i<arr.length−1; i++) {
3     for (int j=i+1; j<arr.length; j++) {
4       if (arr[i] > arr[j]) {
5         int temp = arr[i];
6         arr[i] = arr[j];
7         arr[j] = temp;
8       }
9     }
10  }
11 }
```

Figure 6.7: The source code of `SelectSort` and `BubbleSort` programs.

SelectSort and BubbleSort. These are classical sorting algorithms implemented in java. The source code is depicted in Figure 6.7. These examples are interesting since the cost contributed by the inner loops depends on the value of the outer loops counters. For these benchmark we obtain the following bounds

SelectSort		**BubbleSort**	
(A) $27 \cdot \|a{-}1\|^2 + 16 \cdot \|a{-}1\| + 9$	140	**(A)** $34 \cdot \|c\|^2 + 12 \cdot \|c\| + 8$	180
(B) $\frac{27}{2} \cdot \|a{-}1\|^2 + \frac{59}{2} \cdot \|a{-}1\| + 9$	224	**(B)** $17 \cdot \|c\|^2 + 29 \cdot \|c\| + 8$	320
(C) $\frac{13}{2} \cdot \|a{-}1\|^2 + \frac{45}{2} \cdot \|a{-}1\| + 9$	184	**(C)** $8 \cdot \|c\|^2 + 20 \cdot \|c\| + 8$	232

Looking at the corresponding graphs in Figure 6.2, we can see that our UBs are more precise than those of [5] and our LBs are very tight.

MergeSort. This program implements the classical merge-sort algorithm. The source code is depicted in Figure 6.8. It is an example of divide-and-conquer algorithms which first divides the input list into two lists, sort each list and finally merges the two sorted list into one list. This example illustrates the reason for which the geometric progression behavior is required. In fact, as we have seen in Section 4.2.2, we are able to infer a very tight bound of such divide-and-conquer algorithms. For this benchmark we obtain the following bounds

```
1 void msort(int[] data) {
2   sort(data,0,data.length);
3 }
4 void sort(int[] data, int fm, int to) {
5   int mid;
6   if (fm < to) {
7     mid = (fm + to) / 2;
8     sort(data, fm, mid);
9     sort(data, mid + 1, to);
10    merge(data, fm, to, mid);
11  }
12 }
```

```
13 void merge(int[] data, int fm, int to, int mid) {
14   int i, From = fm, To = mid + 1;
15   int scratch[] = new int[data.length];
16   for (i = fm; i <= to; i++) {
17     if ((From <= mid) &&
18           ((To > to) || data[From]>data[To]))
19       scratch[i] = data[From++];
20     else scratch[i] = data[To++];
21   }
22   for (i = fm; i <= to; i++)
23     data[i]=scratch[i];
24 }
```

Figure 6.8: The sorce code of the `MergeSort` program.

(A) $37 \cdot \|b+1\| \cdot \|2b-1\| + 53 \cdot \|2b-1\| + 11$	1235
(B) $37 \cdot \|b+1\| \cdot log_2(\|2b-1\|+1) + 53 \cdot \|2b-1\| + 11$	2080
(C) 4	1248

We can clearly see that our UB is more precise than that of [5], however, our LB is not precise in this case because our approach for inferring LB on the number of iterations supports only linear functions (in this case it is logarithmic).

PascalTriangle. This program computes and prints Pascal's Triangle as depicted in Figure 6.9. Note that in the nested loops at lines 8-10 and 11-16, the cost accumulated by the inner loops depends on the value of the outer loops counters. For this benchmark we obtain the following bounds

(A) $30 \cdot \|a\|^2 + 27 \cdot \|a-1\|^2 + 33 \cdot \|a\| + 10 \cdot \|a-1\| + 25$	716
(B) $\frac{41}{2} \cdot \|a\|^2 + 27 \cdot \|a-1\|^2 + 10 \cdot \|a-1\| + \frac{85}{2} \cdot \|a\| + 25$	924
(C) $\frac{41}{2} \cdot \|a\|^2 + 27 \cdot \|a-1\|^2 + 10 \cdot \|a-1\| + \frac{85}{2} \cdot \|a\| + 25$	908

Looking at the corresponding graph in Figure 6.2, we can see the our UB is more precise than that of [5], and our LB is very tight.

```
1 public static void pt(int n) {
2   int trian[][] = new int[n][n];
3   for (int i = 0; i < n; i++)
4     for (int j = 0; j < n; j++)
5       trian[i][j] = 0;
6   for(int i = 0; i < n; i++)
7     trian[i][0] = 1 ;
8   for (int i = 1; i < n; i++)
9     for (int j = 1; j < n; j++)
10      trian[i][j]=trian[i-1][j-1]+trian[i-1][j];
11  for (int i = 0; i < n; i++) {
12    for(int j=0;j<=i;j++) {
13      System.out.print(trian[i][j]+ " ");
14    }
15    System.out.println();
16  }
17}
```

Figure 6.9: Source code of the `PascalTriangle` program.

```
1 void f(int n) {
2   for(int i=0;i<n;i++)
3     System.out.println(n+" "+i);
4   for(int i=n-1;i>=0;i--)
5     f(i);
6 }
```

Figure 6.10: Source code of the `NestedRecIter` program.

NestedRecIter. This example uses a programming pattern, depicted in Figure 6.10, which we have found in the Java libraries. Note that the second loop invokes n recursive calls, and each one with a different value for the argument, and each such call will have a linear cost that is consumed by the first loop. Note

that this programming pattern is similar to the pattern that one would use to write a program that prints all permutations of a given array with n elements. For this benchmark we obtain the following bounds

(A) $10 + 5 \cdot \|a\| + (2^{\|a\|} - 1) \cdot (17 + 5 \cdot \|a - 1\|) + 2 \cdot 2^{\|a\|}$	180
(B) $29 \cdot 2^{\|a\|} - 17$	650
(C) $29 \cdot 2^{\|a\|} - 17$	630

We can clearly see that our UB is asymptotically more precise than that of [5], and our LB is very tight.

Let us summarize the results that we have obtained for the benchmarks above, and see how good they are with respect to [5]. As regards UBs, we improve the precision over [5] in all benchmarks. This improvement, in all benchmarks except MergeSort, is due to nested loops where the inner loops bounds depend on the outer loops counters. In these cases, we accurately bound the cost of each iteration of the inner loops, rather than assuming the worst-case cost. Moreover, our UBs are very close to the real cost (the difference is only in some constants). In Figure 6.2, it can be seen that the precision gain is greater for larger values of the inputs. This is because, for larger inputs, the length of chains of calls is larger. Since, in our approach, at each iteration each $\|.\|$ expression increases towards the maximum (or decreases from the maximum) gradually, the overall gain becomes larger. For MergeSort, we obtain a tight bound in the order of $b * log(b)$. Note that [5] could obtain $b * log(b)$ only for simple cost models that count the visits to a specific program point but not for number of instructions, while ours works with any cost model.

As regards LBs, it can be observed from row C of each benchmark that we have been able to prove the positive $\|.\|$ condition and obtain nontrivial LBs in all cases except in MergeSort. For MergeSort, the LB on loop iterations is logarithmic which cannot be inferred by our linear invariant generation tool and hence we get the trivial bound 4. Note that for InsertSort we infer a linear LB which happens when the array is sorted. Our approach is slightly slower than [5] mainly due to the overhead of connecting COSTA to the external *CAS*.

Examples	(A) Our UBs (B) UBs of RAML	Time
`appendAll`	**(A)** $\|a\|\cdot\|b\| + 2\cdot\|a\| + 1$	64
	(B) $a\cdot b + 2\cdot a + 1$	46
`remove`	**(A)** $\|b\|\cdot\|c\| + 2\cdot\|b\| + 1$	62
	(B) $b\cdot c + 2\cdot b + 1$	70
`nub`	**(A)** $\frac{1}{2}\cdot\|a\|^2\cdot\|b\| + \frac{1}{2}\cdot\|a\|\cdot\|b\| + \|a\|^2 + 3\cdot\|a\| + 1$	105
	(B) $\frac{1}{2}\cdot a^2\cdot b + a^2 - \frac{1}{2}\cdot a\cdot b + a + 1$	120
`insertsort`	**(A)** $\frac{1}{2}\cdot\|a\|^2 + \frac{5}{2}\cdot\|a\| + 1$	33
	(B) $\frac{1}{2}\cdot a^2 + \frac{3}{2}\cdot a + 1$	74
`lcs`	**(A)** $4\cdot\|a\|\cdot\|b\| + \|b\| + 2\cdot\|a\| + 3$	88
	(B) $4\cdot a\cdot b + b + 2\cdot a + 3$	116
`isortlist`	**(A)** $\frac{1}{2}\cdot\|a\|^2\cdot\|b\| + \|a\|^2 + \frac{3}{2}\cdot\|a\|\cdot\|b\| + 4\cdot\|a\| + 1$	106
	(B) $\frac{1}{2}\cdot a^2\cdot b + a^2 - \frac{1}{2}\cdot a\cdot b + a + 1$	114
`minSort`	**(A)** $\frac{1}{2}\cdot\|a\|^2 + \frac{5}{2} * \|a\| + 2$	27
	(B) $\frac{1}{2}\cdot a^2 + \frac{5}{2}\cdot a + 2$	45
`eratos`	**(A)** $\frac{1}{2}\cdot\|a\|^2 + \frac{5}{2}\cdot\|a\| + 1$	35
	(B) $\frac{1}{2}\cdot a^2 + \frac{3}{2}\cdot a + 1$	44
`dyade`	**(A)** $\|a\|\cdot\|b\| + 2\cdot\|a\| + 1$	40
	(B) $2\cdot a\cdot b + 2\cdot a + 1$	44
`matrixmul`	**(A)** $\|a\|\cdot\|c\|\cdot\|d\| + 2\cdot\|a\|\cdot\|c\| + 2\cdot\|a\| + 1$	147
	(B) $a\cdot c\cdot d + 2\cdot a\cdot b + 2\cdot a + 1$	331
`mult3`	**(A)** $2\cdot\|a\|^2 + 8\cdot\|a\| + 3$	70
	(B) $4\cdot a\cdot b + 6\cdot a + 3$	193
`mergesort`	**(A)** $log_2(\|2\cdot a - 3\|+1)\cdot\|a-\frac{1}{2}\|+log_2(\|2\cdot a-3\|+1)\cdot\|\frac{a}{2}\|$ $+4\cdot\|2\cdot a-3\|+1$	76
	(B) $\frac{7}{2}\cdot a^2 - \frac{5}{2}\cdot a + 1$	73
`quicksort`	**(A)** $8\cdot 2^{\|a\|} - 2\cdot\|a\| - 7$	83
	(B) $a^2 + 3\cdot a + 1$	76
`apendAll3`	**(A)** $3\cdot\|a\|\cdot\|b\|\cdot\|c\|\cdot\|d\|+2\cdot\|a\|\cdot\|b\|\cdot\|c\|+3\cdot\|a\|\cdot\|b\|+3\cdot\|a\|+1$	598
	(B) $3\cdot a\cdot b\cdot c\cdot d + 2\cdot a\cdot b\cdot c + 3\cdot a\cdot b + 3\cdot a + 1$	3113

Table 6.1: Comparing Our UB Results with Hofmann et. al [45]

In addition to the above benchmarks and comparison to [5], we have also compared experimentally our approach to [45], which was developed in parallel to our work. The comparison is made on their examples, which are available at `http://raml.tcs.ifi.lmu.de`. These examples are written in a first-order functional languages (RAML). In order to perform a fair comparison, we have done the following: (i) RAML programs are first translated into an extended form of *CRs* in order handle polynomial input-output relations that are not handled by the basic framework of COSTA, these *CRs* are available at `http://costa.ls.fi.upm.es/pubs`; and (ii) we used a cost model that counts the number of visits to a specific point (entries of functions) as this can be easily defined in RAML.

The comparison is presented in Table 6.1. The first column illustrates the benchmarks, the second column illustrates the computed UBs by our approach (marked with (A)) and the UBs computed by the RAML prototype (marked with (B)) and finally the last column indicates the time measured in milliseconds to perform the experiment. The main conclusions drawn from these comparisons are: (1) in most cases we are as precise as [45] and sometimes the results differ only in the constants; (2) for QuickSort our analysis fails to infer the precise bound, this is because the input list is divided into two lists of different length; and (3) for MergeSort, our analysis is more precise since [45] cannot infer bounds with logarithmic expressions. Note that we measured time to solve UBs from *CRs*. It does not include the time to generate *CRs* from RAML code since it was generated manually. In most cases our experiments take less time than RAML. From this, we conclude that our approach is also experimentally efficient.

Last thing to note is that the approach of [45] is based on using a polynomial template with fixed degree k (which is given by the user). This means that the inferred polynomial bound must be expressible within the degree of the template, otherwise the analysis fails. Moreover, if the degree k is higher than what is required to express an UB for the analyzed program, then the analysis runtime increases significantly. For example, for the `matrixmult` program, it computes UBs in 169 milliseconds when the provided degree is 3, where it takes almost 22 seconds when the provided degree is 6. Our approach does not suffer from this problem and is completely automatic.

6.3 Concluding Remarks

We believe that the experiments presented in the previous section demonstrate that our approach is precise, efficient, and can succeed on example where [5] fails to obtain precise bounds. Unfortunately, there are no other cost analysis tools for imperative languages that are available to perform experimental comparison (e.g., SPEED [42] is not available) on those benchmarks. Our solver performs well also when compared to [45], since it succeeded to obtain similar bounds for all examples except those that require amortized analysis. Moreover, unlike [45], our solver can obtain non-polynomial bounds.

Part II

Deciding Termination of Integer Loops

Chapter 7

Overview of the Problems, Challenges, and Contributions

In this chapter we overview the problem of deciding termination of several variations of integer loops, the challenges one faces when solving this problem, and a brief informal overview of our solutions. Section 7.1 describes the problems and its related challenges, Section 7.2 summarizes the contributions of part II of this thesis, and Section 7.3 briefly overviews the organization of this part.

7.1 The Problems and the Challenges

As we have seen in Part I of this thesis, solving *CRs* into closed-form bounds requires bounding the number of iterations that a given loop can make, a problem that is clearly related to its termination behavior. This means that features like precision, scalability, and applicability of *CRs* solving techniques are directly related to the corresponding features of deciding termination of such loops. One can explore these features for a specific termination algorithm, by studying its complexity, which gives an indication on how the algorithm will perform in practice. An alternative approach is to study the computational complexity of the problem (and not a specific algorithm), which gives an indication on how practical *any* algorithm that solves this problem can be. In this part of the thesis we conduct such study for the problem of deciding termination of (simple) integer

loops, a form of loops that is very common in cost analysis. This study is of interest not only for cost analysis, but rather for termination analysis in general, and thus we study this problem in a more wider context rather than being restricted to those cases that occur in cost analysis.

Much of the recent development in termination analysis has benefited from techniques that deal with one simple loop at a time, where a simple loop is specified by (optionally) some initial conditions, a loop guard, and a "loop body" of a very restricted form. Very often, the state of the program during the loop is represented by a finite set of scalar variables (this simplification may be the result of an abstraction, such as size abstraction of structured data [56, 55, 76, 26]).

Regarding the representation of the loop body, the most natural one is, perhaps, a block of straight-line code, namely a sequence of assignment statements, as in the following example:

$$\mathit{while}\ (X > 0)\ \mathit{do}\ \{\, X := X + Y;\, Y := Y - 1;\, \} \tag{7.1}$$

To define a restricted problem for theoretical study, one just has to state the types of loop conditions and assignments that are admitted.

By symbolically evaluating the sequence of assignments, a straight-line loop body may be put into the simple form of a simultaneous deterministic update, namely loops of the form

$$\mathit{while}\ C\ \mathit{do}\ \langle x_1, \ldots, x_n \rangle := f(\langle x_1, \ldots, x_n \rangle) \tag{7.2}$$

where f is a function of some restricted class. For function classes that are sufficiently simple to analyze, one can hope that termination of such loops would be decidable; in fact, the motivation to this work comes not only from problems that we encountered in cost analysis, but rather from the remarkable results by [78] and [25] on the termination of *linear loops*, a kind of loops where the update function f is linear. The loop conditions in these works are conjunctions of linear inequalities. Specifically, Tiwari proved that the termination problem is decidable for loops of the following form:

$$\mathit{while}\ (B\bar{x} > \bar{b})\ \mathit{do}\ \bar{x} := A\bar{x} + \bar{c} \tag{7.3}$$

where the arithmetic is done over the reals; thus the variable vector $\bar{x}$ has values in $\mathbb{R}^n$, and the constant matrices in the loop are $B \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{n \times n}$, $\bar{b} \in \mathbb{R}^m$ and $\bar{c} \in \mathbb{R}^n$.

Subsequently, Braverman proved decidability of termination of loops of the following form:

$$\textit{while } (B_s\bar{x} > \bar{b}_s) \wedge (B_w\bar{x} \geq \bar{b}_w) \textit{ do } \bar{x} := A\bar{x} + \bar{c} \tag{7.4}$$

where the constant matrices and vectors are *rational*, and the variables are of either real or rational type; moreover, in the homogeneous case ($\bar{b}_s, \bar{b}_w, \bar{c} = 0$) he proved decidability when the variables range over $\mathbb{Z}$. This is a significant and nontrivial addition, since algorithmic methods that work for the reals often fail to extend to the integers (a notorious example is finding the roots of polynomials; while decidable over the reals, over the integers, it is the undecidable *Hilbert* 10^{th} *problem*[1]). Regarding the loop form (7.4), we note that the constant vector $\bar{c}$ may be assumed to be zero with no loss of generality, since variables can be used instead, and constrained by the loop guard to have the desired (constant) values. Over the integers it is also sufficient to have only $\geq$ or only $>$ in the loop guard. However, replacing $>$ by $\geq$ (or vice versa) alters the homogeneous loop to a non-homogeneous one, which is why including both inequality types is important in the context of [25].

Going back to program analysis, we note that it is typical in this field to assume that some degree of approximation is necessary in order to express the effect of the loop body by linear arithmetics alone. Hence, rather than loops with a linear update as above, one defines the representation of a loop body to be a set of *constraints* (again, usually linear). The general form of such a loop is

$$\textit{while } (B\bar{x} \geq \bar{b}) \textit{ do } A \begin{pmatrix} \bar{x} \\ \bar{x'} \end{pmatrix} \leq \bar{c} \tag{7.5}$$

where the loop body is interpreted as expressing a relation between the new values $\bar{x}'$ and the previous values $\bar{x}$. Thus, in general, this representation is a nondeterministic kind of program and may over-approximate the semantics of

[1]Over the rationals, the problem is still open, according to [59].

the source program analyzed. But this is a form which lends itself naturally to analysis methods based on linear programming techniques, and there has been a series of publications on proving termination of such loops [75, 62, 67] — all of which rely on the generation of *linear ranking functions*. For example, the termination analysis tools *Terminator* [30], *COSTA* [6], and *Julia* [76] are based on proving termination of such loops by means of a linear ranking function.

It is known that the linear-ranking approach cannot completely resolve the problem [67, 25], since not every terminating program has such a ranking function — this is the case, for example, for loop (7.1) above. Moreover, the linear-programming based approaches are not sensitive to the assumption that the data are integers. Thus, the problem of decidability of termination for linear-constraint loops (7.5) stays open, in its different variants. We feel that the most intriguing problem is the following:

> Is the termination of a single linear-constraint loop decidable, when the coefficients are rational numbers and the variables range over the integers?

The problem may be considered for a given initial state, for any initial state, or for a (linearly) constrained initial state.

7.2 Informal overview of the Contributions

In this research, we focus on hardness proofs. Our basic tool is a new simulation of counter programs (also known as counter machines) by a simple integer loop. The termination of counter programs is a well-known undecidable problem. While we have not been able to fully answer the major open problem above, this technique led to some interesting results which improve our understanding of the simple-loop termination problem. We next summarize our main results. All concern integer variables.

1. We prove undecidability of termination, either for all inputs or a given input, for simple loops (a variation of loop form (7.4)) which iterate a straight-line sequence of simple assignment instructions. The right-hand sides are

integer linear expressions except for one instruction type, which computes the step function

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

At first sight it may seem like the inclusion of such an instruction is tantamount to including a branch on zero, which would immediately allow for implementing a counter program. This is not the case, because the result of the function is put into a variable which can only be combined with other variables in a very limited way. We complement this result by pointing out other natural instructions that can be used to simulate the step function. This include integer division by a constant (with truncation towards zero) and truncated subtraction.

2. We show that the undecidability result can be achieved even for loops whose body is a deterministic update of the form "if $x > 0$ then (one linear update) else (another linear update)." Thus, the update function consists of two linear pieces. This is a nontrivial refinement of the first result, which limits the number of times the step function is used in the loop body.

3. Building upon the previous result, we prove undecidability of termination, either for all inputs or for a given input, of linear-constraint loops (a variation of loop form (7.5)) where *one irrational number may appear* (more precisely, the coefficients are from $\mathbb{Z} \cup \{r\}$ for an arbitrary irrational number r).

4. Finally, we observe that while linear-constraint loops (7.5) with rational coefficients seem to be insufficient for simulating *all* counter programs, it is possible to simulate a subclass, namely Petri nets, leading to the conclusion that termination for a given input is at least EXPSPACE-hard.

We would like to highlight the relation of our results to a discussion at the end of [25]. Braverman notes that linear-constraint loops are nondeterministic and asks:

How much nondeterminism can be introduced in a linear loop with no initial conditions before termination becomes undecidable?

It is interesting that our reduction to linear-constraint loops, when using the irrational coefficient, produces constraints that are *deterministic*. The role of the constraints is not to create nondeterminism; it is to express complex relationships among variables. We may also point out that some limited forms of linear-constraint loops (that are very nondeterministic since they are weaker constraints) have a *decidable* termination problem (see Section 10.2). Braverman also discusses the difficulty of deciding termination for a given input, a problem that he left open. Our results apply to this variant, providing a partial answer to this open problem.

7.3 Organization

The rest of this part of the thesis is organized as follows: Chapter 8 provides some background material; Chapter 9 studies the termination of straight-line while loops with a "built-in" function that represents the step function as well as integer linear-constraint loops. It also discusses undecidable extensions of integer linear-constraint loops. Finally, it ends with some concluding remarks.

Chapter 8

Background on Integer Loops

In this chapter we define the syntax of integer piecewise linear while loops, integer linear-constraint loops, and counter programs.

8.1 Integer Piecewise Linear Loops

An integer piecewise linear loop (*IPL* loop for short) with integer variables $X_1, \ldots, X_n$ is a while loop of the form

$$\mathit{while}\ b_1 \wedge \cdots \wedge b_m\ \mathit{do}\ \{c_1; \ldots; c_n\}$$

where each condition b_i is a linear inequality $a_0 + a_1 * X_1 + \cdots + a_n * X_n \geq 0$ with $a_i \in \mathbb{Z}$, and each c_i is one of the following instructions

$$X_i := X_j + X_k \mid X_i := a * X_j \mid X_i := a \mid X_i := \mathit{isPositive}(X_j)$$

such that $a \in \mathbb{Z}$ and

$$\mathit{isPositive}(X) = \begin{cases} 0 & X \leq 0 \\ 1 & X > 0 \end{cases}$$

We consider *isPositive* to be a primitive, but we will also consider alternatives. The semantics of an *IPL* loop is the obvious: starting from initial values for the variables $X_1, \ldots, X_n$ (the input), if the condition $b_1 \wedge \cdots \wedge b_n$ (the loop guard)

holds (we say that the loop is enabled), instructions $c_1, \ldots, c_n$ are executed sequentially, and the loop is restarted at the new state. If the loop guard is false, the loop *terminates*. For simplicity, we may use a composite expressions, e.g., $X_1 := 2 * X_2 + 3 * X_3 + 1$, which should be taken to be syntactic sugar for a series of assignments, possibly using temporary variables.

8.2 Integer Linear-Constraint Loops

An integer linear-constraint loop (*ILC* loop for short) over n variables $\bar{x} = \langle X_1, \ldots, X_n \rangle$ has the form

$$\textit{while } (B\bar{x} \geq \bar{b}) \textit{ do } A \begin{pmatrix} \bar{x} \\ \bar{x'} \end{pmatrix} \leq \bar{c}$$

where for some $m, p > 0$, $B \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{p \times 2n}$, $\bar{b} \in \mathbb{R}^m$ and $\bar{c} \in \mathbb{R}^p$. The case we are most interested in is that in which the constant matrices and vectors are composed of rational numbers; this is equivalent to assuming that they are all integer (just multiply by a common denominator).

Semantically, a state of such a loop is an n-tuple $\langle x_1, \ldots, x_n \rangle$ of integers, and a transition to a new state $\bar{x}' = \langle x'_1, \ldots, x'_n \rangle$ is possible if $\bar{x}, \bar{x}'$ satisfy all the constraints in the loop guard and the loop body. We say that the loop terminates for a given initial state if all possible executions from that state are finite, and that it universally terminates if it terminates for every initial state. We say that the loop is *deterministic* if there is at most one successor state to any state. Note that the guard $B\bar{x} \geq \bar{b}$ is actually redundant, since its constraints can be incorporated in those of the loop body. However, we prefer to keep this form for its similarity with other loop forms studied in previous works, as well as ours (see loop forms (7.1)–(7.5) in the introduction).

8.3 Counter Programs

A (deterministic) counter program P_C with n counters $X_1, \ldots, X_n$ is a list of labeled instructions $1{:}I_1, \ldots, m{:}I_m, m{+}1{:}\textit{stop}$ where each instruction I_k is one of

the following:

$$incr(X_j) \mid decr(X_j) \mid \textit{if } X_j > 0 \textit{ then } k_1 \textit{ else } k_2$$

with $1 \leq k_1, k_2 \leq m{+}1$ and $1 \leq j \leq n$. A state is of the form $(k, \langle a_1, \ldots, a_n \rangle)$ which indicates that Instruction I_k is to be executed next, and the current values of the counters are $X_1 = a_1, \ldots, X_n = a_n$. In a valid state, $1 \leq k \leq m+1$ and all $a_i \in \mathbb{N}$ (it will sometimes be useful to also consider invalid states, and assume that they cause a halt). Any state in which $k = m+1$ is a halting state. For any other valid state $(k, \langle a_1, \ldots, a_n \rangle)$, the successor state is defined as follows.

- If I_k is $decr(X_j)$ (resp. $incr(X_j)$), then X_j is decreased (resp. increased) by 1 and the execution moves to label $k+1$.
- If I_k is "*if* $X_j > 0$ *then* k_1 *else* k_2" then the execution moves to label k_1 if X_j is positive, and to k_2 if it is 0. The values of the counters do not change.

The following are known facts about the halting problem for counter programs.

THEOREM 8.3.1 ([64])**.** *The halting problem for counter programs with $n \geq 2$ counters and the initial state $(1, \langle 0, \ldots, 0 \rangle)$ is undecidable.*

The *termination problem* is the problem of deciding whether a given program halts for every input[1]. The *Mortality* problem asks whether the program halts when started at any state (even a state that cannot be reached in a valid computation).

THEOREM 8.3.2 ([21])**.** *The mortality problem for counter programs with $n \geq 2$ counters is undecidable.*

As mentioned in the introduction, the *termination problem* usually addressed in the context of program analysis is close (or even identical) to the mortality problem, since one takes a program loop (possibly without any context) and asks whether it can be shown to halt on *every* initial state. Hence, the last theorem is useful for proving undecidability of such termination problems.

[1]We also use this term when considering a given input and the termination of all paths of a non-deterministic program.

Chapter 9

Complexity of Deciding Termination of Integer Loops

In this chapter we discuss decidability and complexity issues of *IPL* and *ILC* loops. It is organized as follows

1. Section 9.1 proves that termination of *IPL* loops is undecidable. The undecidability is proved in section 9.1.1 by a reduction of halting and mortality problem for counter machine. Section 9.1.2 shows some examples of piecewise linear functions the presence of which in *IPL* loops make it undecidable.

2. Section 9.2 proves that *IPL* loops with two linear pieces are enough to achieve undecidability of termination.

3. Section 9.3 explains an unsuccessful attempt of proving the undecidability of termination of *ILC* loops and section 9.3.3 shows some extensions of *ILC* loops whose termination is undecidable.

4. Section 9.4 proves that deciding termination of an *ILC* loop with a given input has EXPSPACE lower bound, and Section 9.5 shows that this lower bound is still valid even if the updates are deterministic.

9.1 Termination of *IPL* loops

In this section, we investigate the decidability of the following problems: given an *IPL* loop P,

1. Does P terminate for a given input?
2. Does P terminate for all inputs?

We show that both problems are undecidable by reduction from the halting and mortality problems, respectively, for counter programs. To see where the challenge in this reduction lies, note that the loops under consideration iterate a fixed block of straight-line code, while a counter program has a program counter that determines the next instruction to execute. While one can easily keep the value of the PC in a variable, it is not obvious how to make the computation depend on this variable, and how to simulate branching.

9.1.1 A Reduction from Counter Programs

Given a counter program $P_C \equiv 1{:}I_1, \ldots, m{:}I_m, m{+}1{:}\mathit{stop}$ with counters $X_1, \ldots, X_n$, we generate a corresponding *IPL* loop $\mathcal{T}(P_C)$ as follows:

```
1 while ( A_1 ≥ 0 ∧ ··· ∧ A_m ≥ 0 ∧ A_1 + ··· + A_m = 1 ∧ X_1 ≥ 0 ∧ ··· ∧ X_n ≥ 0 ) do {
2   N_0 := 0; N_1 := A_1; ... N_m := A_m;
3   F_1 := isPositive(X_1); ... F_n := isPositive(X_n);
4   T(1:I_1)
5       ⋮
6   T(m:I_m)
7   A_1 := N_0; ... A_m := N_{m-1}
8 }
```

where $\mathcal{T}(k{:}I_k)$ is defined as follows

- If $I_k \equiv \mathit{incr}(X_j)$, then $\mathcal{T}(k{:}I_k)$ is $X_j := X_j + A_k$;
- If $I_k \equiv \mathit{decr}(X_j)$, then $\mathcal{T}(k{:}I_k)$ is $X_j := X_j - A_k$;

- If $I_k \equiv \mathit{if}\ X_j > 0\ \mathit{then}\ k_1\ \mathit{else}\ k_2$, then $\mathcal{T}(k{:}I_k)$ is

$$
\begin{aligned}
&T_k := \mathit{isPositive}(A_k + F_j - 1);\\
&R_k := \mathit{isPositive}(A_k - F_j);\\
&N_k := N_k - A_k;\\
&N_{k_1-1} := N_{k_1-1} + T_k;\\
&N_{k_2-1} := N_{k_2-1} + R_k;
\end{aligned}
$$

EXAMPLE 9.1.1. *Consider the following 2-counter program P_C, which decrements x and y until one of them reaches 0*

```
1  1: x=x−1
2  2: if x>0 then 3 else 5
3  3: y=y−1
4  4: if y>0 then 1 else 5
5  5: stop
```

Applying $\mathcal{T}(P_C)$ results in the following IPL loop

```
1  while(A1 ≥ 0 ∧ A2 ≥ 0 ∧ A3 ≥ 0 ∧ A4 ≥ 0 ∧ A1 + ··· + A4 = 1 ∧ x ≥ 0 ∧ y ≥ 0) do {
2    N0 := 0; N1 := A1; N2 := A2; N3 := A3; N4 := A4;
3    Fx := isPositive(x); Fy := isPositive(y);
4
5    x := x − A1;
6
7    T2 := isPositive(A2 + Fx − 1);
8    R2 := isPositive(A2 − Fx);
9    N2 := N2 − A2;
10   N2 := N2 + T2;
11   N4 := N4 + R2;
12
13   y := y − A3;
14
15   T4 := isPositive(A4 + Fy − 1);
16   R4 := isPositive(A4 − Fy);
17   N4 := N4 − A4;
```

18 $N_0 := N_0 + T_4$;
19 $N_4 := N_4 + R_4$;
20
21 $A_1 := N_0$; $A_2 := N_1$; $A_3 := N_2$; $A_4 := N_3$;
22 }

Line 5 *corresponds to instruction* I_1*, lines* $7 - 11$ *to instruction* I_2*, Line* 13 *to instruction* I_3*, and lines* $15 - 19$ *to instruction* I_4*.*

Let us first state, informally, the main ideas behind the reduction, and then formally prove Lemma 9.1.3 which in turn implies Theorem 9.1.4.

1. Variables $A_1, \ldots, A_m$ are flags that indicate the instruction to be executed next. They take values from $0, 1$, and only one of them can be 1 as stated by the loop guard. Note that an operation $X_j := X_j + A_k$ (resp. $X_j := X_j - A_k$) will have effect only when $A_k = 1$, and otherwise is a no-op. This is a way of simulating only one instruction in every iteration.

2. The values of A_i are modified in a way that simulates the control of the counter machine. Namely, if $A_k = 1$, and the instruction I_k is $incr(X_j)$ or $decr(X_j)$, then the last line in the loop body sets A_{k+1} to 1 and the rest to 0. If I_k is a condition, it will set A_{k_1} or A_{k_2}, depending on the tested variable, to 1, and the rest to 0. The variables F_k, N_k, R_k, and T_k are auxiliary variables for implementing this.

LEMMA 9.1.2. *Let* P_C *be a counter program,* $\mathcal{T}(P_C)$ *its corresponding IPL loop,* $S \equiv (k, \langle a_1, \ldots, a_n \rangle)$ *a valid state for* P_C*, and* $S_{\mathcal{T}}$ *a state of* $\mathcal{T}(P_C)$ *where* $A_1 = 0, \ldots, A_k = 1, \ldots, A_m = 0$*,* $X_1 = a_1, \ldots, X_n = a_n$*.*

If S *has a successor state* $(k', \langle a'_1, \ldots, a'_n \rangle)$ *in* P_C*, then the loop of* $\mathcal{T}(P_C)$ *is enabled at* $S_{\mathcal{T}}$ *and its execution leads to a state in which* $A_1 = 0, \ldots, A_{k'} = 1, \ldots, A_m = 0$*,* $X_1 = a'_1, \ldots, X_n = a'_n$*. If* S *is a halting configuration of* P_C*, the loop of* $\mathcal{T}(P_C)$ *is disabled at* $S_{\mathcal{T}}$*.*

Proof. It is clear that if an execution step is possible in P_C then $0 \leq k \leq m$ and all X_j are nonnegative, and thus the condition of the loop $\mathcal{T}(P_C)$ is true. Now

note that when $A_k = 0$ the encoding of I_k does not change the value of any N_i or X_j, and consider the following two cases: (1) If I_k is $incr(X_j)$ (resp. $decr(X_j)$), then P_C increments (resp. decrements) X_j and moves to label $k' = k+1$. Clearly the encoding of I_k increments (resp. decrements) X_j and all N_i are not modified. Since $N_k = A_k = 1$, the last line of the loop sets A_{k+1} to 1 (unless $k+1 = m+1$) and all other A_i to 0. (2) if I_k is *if* $X_j > 0$ *then* k_1 *else* k_2, then the counter machine moves to k_1 (resp. k_2) if $X_j > 0$ (resp. $X_j = 0$). Suppose $X_j > 0$, then $T_k = 1$ and $R_k = 0$, $N_{k_1-1} = 1$ and $N_{k_2-1} = 0$. Thus, when reaching the last line the instruction $A_{k_1} := N_{k_1-1}$ sets A_{k_1} (unless $k_1 = m+1$). The case where $X_j = 0$ is similar.

In a halting state, $k = m+1$ which means that $A_1, \ldots, A_m = 0$. Hence, the loop is disabled. □

LEMMA 9.1.3. *A counter program P_C with $n \geq 2$ counters terminates for the initial state $(k, \langle a_1, \ldots, a_n \rangle)$ if and only if $\mathcal{T}(P_C)$ terminates for input $A_1 = 0, \ldots, A_k = 1, \ldots, A_m = 0$, $X_1 = a_1, \cdots, X_n = a_n$.*

Proof. An immediate consequence of Lemma 9.1.2. □

Note that when values of the variables in $\mathcal{T}(P_C)$ do not correspond to a valid state for P_C, then the guard of $\mathcal{T}(P_C)$ is disabled and thus $\mathcal{T}(P_C)$ terminates for such input. This, together with Lemma 9.1.3, and theorems 8.3.1 and 8.3.2, imply

THEOREM 9.1.4. *The halting problem and the termination problem for IPL loops are undecidable.*

9.1.2 Examples of Piecewise Linear Operations

The *isPositive* operation can easily be simulated by other natural instructions, yielding different instruction sets that suffice for undecidability.

EXAMPLE 9.1.5 (Integer division)**.** *Consider an instruction that divides an integer by an integer constant and truncates the result towards zero (also if it is negative). Using this kind of division, we have*

$$isPositive(X) = X - \frac{2 * X - 1}{2}$$

and thus, termination is undecidable for loops with linear assignments and integer division of this kind.

EXAMPLE 9.1.6 (Truncated subtraction)**.** *Another common piecewise-linear function is* truncated subtraction*, such that $x \dot{-} y$ is the same as $x - y$ if it is positive, and otherwise 0. This operation allows for implementing isPositive thus:*

$$isPositive(X) = 1 \dot{-} (1 \dot{-} X)$$

9.2 Loops with Two Linear Pieces

The reduction in Section 9.1.1 presented the loop body as a sequence of instructions that compute either linear or piecewise-linear operations. This means that the loop body, considered as a function from the entry state to the exit state, is piecewise-linear. In order to get closer to the simplest form where decidability is open, namely a body which is an affine-linear deterministic update, in this section we reduce the number of nonlinearities in that reduction. More precisely, we consider the update to be a function, the union of several linear pieces, and ask how many such pieces make the termination problem undecidable. Next, we improve the proof from Section 9.1.1 in this respect, reducing the usage of the *step function*. This will imply the following theorem.

THEOREM 9.2.1. *The halting problem and the termination problem are undecidable for loops of the following form*

$$\textit{while } (B\bar{x} \geq \bar{b}) \textit{ do } \bar{x} := \begin{cases} A_0\bar{x} & X_i \leq 0 \\ A_1\bar{x} & X_i > 0 \end{cases}$$

where the state vector $\bar{x} = \langle X_1, \ldots, X_n \rangle$ ranges over $\mathbb{Z}^n$, $A_0, A_1 \in \mathbb{Z}^{n \times n}$, $\bar{b} \in \mathbb{Z}^p$ for some $p > 0$, $B \in \mathbb{Z}^{p \times n}$, and $X_i \in \bar{x}$.

The proof is a reduction from the corresponding problems for *two-counter* machines. Recall that [64] proved that halting for a given input is undecidable with two counters, and [21] proved it for mortality. The reduction shown in Section 9.1, instantiated for the case of two counters, almost establishes the result.

Observe that if the values of F_1 and F_2 are known, then the flags T_k and R_k can be set to a linear function of A_k, e.g., $T_k := \mathit{isPositive}(A_k + F_1 - 1)$ can be rewritten to $T_k := A_k$ when $F_1 = 1$, and to $T_k := 0$ when $F_1 = 0$.

Thus, the body of the loop can be expressed by a linear function in each of the four regions determined by the signs of X_1 and X_2 (which define the values of F_1 and F_2). In what follows we modify the construction to reduce the four regions to only two regions.

The basic idea is to replace the two instructions $F_1 := \mathit{isPositive}(X_1)$ and $F_2 := \mathit{isPositive}(X_2)$ by the single instruction $F := \mathit{isPositive}(X_1)$, which will compute the signs of both X_1 and X_2. This is done by introducing an auxiliary iteration such that in one iteration F is set according to the sign of X_2, and in the next iteration it is set according to the sign of X_1 (by swapping the values of X_1 and X_2).

We now assume given a counter program $P_C \equiv 1{:}I_1, \ldots, m{:}I_m, m{+}1{:}\mathit{stop}$ with two counters X_1 and X_2. We first extend the set of flags A_k to range from A_1 to A_{2m}, and N_k to range from N_0 to N_{2m}. We also let $k_1, \ldots, k_i$ be indices of all instructions that perform a zero-test. Then, P_C is translated to an *IPL* loop $\mathcal{T}'(P_C)$ as follows

1 **while** $(A_1 \geq 0 \wedge \cdots \wedge A_{2m} \geq 0 \wedge A_1 + \cdots + A_{2m} = 1 \wedge X_1 \geq 0 \wedge X_2 \geq 0 \wedge$
2 $\quad 0 \leq T_{k_1} + R_{k_1} \leq A_{2k_1} \wedge \cdots \wedge 0 \leq T_{k_i} + R_{k_i} \leq A_{2k_i})$
3 $N_0 := 0$; $N_1 := A_1$; $\ldots$ $N_{2m} := A_{2m}$;
4 $(X_2,X_1) := (X_1,X_2)$; // swap X_1, X_2
5 $F := \mathit{isPositive}(X_1)$;
6 $\mathcal{T}'(1{:}I_1)$;
7 $\quad\vdots$
8 $\mathcal{T}'(m{:}I_m)$
9 $A_1 := N_0$; $A_2 := N_1$; $\ldots$ $A_{2m} := N_{2m-1}$
10 }

The translation $\mathcal{T}'$ of counter-program instructions follows. For increment and decrement, it is similar to what we have presented in Section 9.1, we only modify the indexing of the A_k variables.

- If $I_k \equiv \mathit{incr}(X_j)$, then $\mathcal{T}'(k{:}I_k)$ is $X_j := X_j + A_{2k}$

- If $I_k \equiv decr(X_j)$, then $\mathcal{T}'(k{:}I_k)$ is $X_j := X_j - A_{2k}$

For the conditional instruction, there are different translations for a test on X_1 and for a test on X_2:

- If $I_k \equiv \textit{if } X_1 > 0 \textit{ then } k_1 \textit{ else } k_2$, then $\mathcal{T}'(k{:}I_k)$ is

$$\begin{aligned} T_k &:= isPositive(A_{2k} + F - 1);\\ R_k &:= isPositive(A_{2k} - F);\\ N_{2k} &:= N_{2k} - A_{2k};\\ N_{2k_1-2} &:= N_{2k_1-2} + T_k;\\ N_{2k_2-2} &:= N_{2k_2-2} + R_k; \end{aligned}$$

- If $I_k \equiv \textit{if } X_2 > 0 \textit{ then } k_1 \textit{ else } k_2$, then $\mathcal{T}'(k{:}I_k)$ is

$$\begin{aligned} N_{2k} &:= N_{2k} - A_{2k};\\ N_{2k_1-2} &:= N_{2k_1-2} + T_k;\\ N_{2k_2-2} &:= N_{2k_2-2} + R_k;\\ T_k &:= isPositive(A_{2k-1} + F - 1);\\ R_k &:= isPositive(A_{2k-1} - F); \end{aligned}$$

Note that the above *IPL* loop can be represented in the form described in Theorem 9.2.1. This is because when the value of F is known, each of T_k and R_k can be set to a linear function of the corresponding A_k.

EXAMPLE 9.2.2. *Consider again the counter program of Example 9.1.1, and note that, in $\mathcal{T}(P_C)$, the loop body can be expressed as a four-piece linear function depending on the signs of x and y. This is because, as we have mentioned before, once the flags F_x and F_y are known, then the flags T_2, R_2, T_3 and R_3 can be defined by means of linear expressions. Applying the new transformation $\mathcal{T}'$ results in the following IPL loop:*

```
1 while(A_1 ≥ 0 ∧ ... ∧ A_8 ≥ 0 ∧ A_1 + ··· + A_8 = 1 ∧
2                 x ≥ 0 ∧ y ≥ 0 ∧ 0 ≤ T_2 + R_2 ≤ A_4 ∧ 0 ≤ T_4 + R_4 ≤ A_8) do {
3   N_0 := 0; N_1 := A_1; ...; N_8 := A_8;
4   (y, x) := (x, y); // swap x and y
5   F := isPositive(x);
6
7   x := x − A_2;
8
9   T_2 := isPositive(A_4 + F − 1);
10  R_2 := isPositive(A_4 − F);
11  N_4 := N_4 − A_4;
12  N_4 := N_4 + T_2;
13  N_8 := N_8 + R_2;
14
15  y := y − A_6;
16
17  N_8 := N_8 − A_8;
18  N_0 := N_0 + T_4;
19  N_8 := N_8 + R_4;
20  T_4 := isPositive(A_7 + F − 1);
21  R_4 := isPositive(A_7 − F);
22
23  A_1 := N_0; A_2 := N_1; ... A_8 := N_7;
24 }
```

Line 7 *corresponds to instruction* I_1*, lines* $9-13$ *to instruction* I_2*, line* 15 *to instruction* I_3*, and lines* $17-21$ *to instruction* I_4*. Note that the body of this loop can be expressed as a two-piece linear function depending on the sign of* x*, since once the value of* F *is known, the values of* T_2, R_2, T_3 *and* R_3 *can be defined by linear expressions.*

Let us explain the intuition behind the above reduction. First note that even indices for A_k represent labels in the counter program, while odd indices are used to introduce the extra iteration that computes the sign of X_2. Suppose the counter program is in a state $(k, \langle a_1, a_2 \rangle)$. To simulate one execution step of the counter program, we start the *IPL* loop from a state in which $A_{2k-1} = 1$ (all other

A_i are 0), $X_1 = a_1$, $X_2 = a_2$, and all T_i and R_i are set to 0. Starting from this state, in the first iteration the counter variables are swapped, F is set according to the sign of X_2, and executing the encodings of all instructions is equivalent to no-op, except when I_k is a test on X_2 in which case the corresponding R_k and T_k record the result of the test. At the end of this iteration the last line of the loop body sets A_{2k} to 1. In the next iteration, the counter variables are swapped again, and F is set to the sign of X_1. Then

- if $I_k \equiv incr(X_j)$ or $I_k \equiv decr(X_j)$, then $\mathcal{T}'(I_k)$ simulates the corresponding counter-program instruction (since in such encoding we use the flag A_{2k}), and $A_{2(k+1)-1}$ is set to 1.

- if $I_k \equiv$ *if* $X_1 > 0$ *then* k_1 *else* k_2, then $\mathcal{T}'(I_k)$, as in Section 9.1, sets either A_{2k_1-1} or A_{2k_2-1} to 1, i.e., it simulates a jump to k_1 or k_2.

- if $I_k \equiv$ *if* $X_2 > 0$ *then* k_1 *else* k_2, then the first 3 lines of $\mathcal{T}'(I_k)$, together with the last line of the loop body, set either A_{2k_1-1} or A_{2k_2-1} to 1, i.e., it simulates a jump to k_1 or k_2. Note that it uses the values of T_k and R_k computed in the previous iteration. In addition, T_k and R_k are set to 0.

This basically implies that if one execution step of the counter program leads to a configuration $(k', \langle a'_1, a'_2 \rangle)$, then two iterations of the *IPL* loop lead to a state in which $A_{2k'-1} = 1$ (and all other A_i are 0), $X_1 = a'_1$, $X_2 = a'_2$, and all R_i and T_i are 0. Thus, with a proper initial state, we obtain a step-by-step simulation of the counter program, proving that the halting problem has been reduced correctly.

Recall that we prove undecidability of the termination problem for our loops by reducing from the mortality problem for counter programs, in which any initial configuration of the counter program is admissible. We have seen that every initial state in which only one A_{2k-1} is set to 1, for any k, and all T_k and R_k (when I_k is a test on X_2) are 0, simulates a possible state of the counter program. To establish correctness of the reduction, we should extend the argument to cover the cases that the program is started with A_{2k} set to 1, or some T_k and R_k are not 0. We refer to such states as *improper* since they do not arise in a proper simulation of the counter program.

- When A_{2k-1} is set to 1, and some T_k and R_k are not 0, the condition $0 \leq T_k + R_k \leq A_{2k}$ is false, and thus the loop is not enabled.

- When A_{2k} is set to 1, it is easy to verify that after one iteration: if I_k is increment (or decrement), then A_{2k+1} is set to 1 (unless $k = m$). If I_k is a test, then either A_{2k_1-1} or A_{2k_2-1}, or none of the A_i, is set to 1, depending on the values of T_k and R_k (at most one of them can be 1). In all cases, all T_k and R_k are set to the intended values.

We conclude that starting at an improper state either leads to immediate termination, or into a proper state. Thus, termination of the loop for all initial states reflects correctly the mortality of the counter program.

9.3 Reduction to *ILC* Loops

In this section we turn to integer linear-constraint loops. We first attempt to modify the reduction described in Section 9.1.1 to produce constraint loops in which all coefficients are rational, and explain where and why it fails. So we do not obtain undecidability for *ILC* loops with rational coefficients, but we show that if there is one irrational number that we are allowed to use in the constraints (any irrational will do) the reduction can be completed and undecidability of termination proved. Undecidability can also be achieved by allowing a special constant ∞. In Section 9.4 we describe another way of handling the failure of the reduction with rational coefficients only: reducing from a weaker model, and thereby proving a lower bound which is weaker than undecidability (but still non-trivial).

Observe that the loop constructed in Section 9.1.1 uses non-linear expressions only for setting the flags T_k,R_k and F_j, the rest is clearly linear. Assuming that we can encode these flags with integer linear constraints, adapting the rest of the reduction to *ILC* loops is straightforward: it can be done by rewriting $\mathcal{T}(P_C)$ to avoid multiple updates of a variable (that is, to *static single assignment* form) and then representing each assignment as an equation instead. Thus, in what follows we concentrate on how to represent those flags using integer linear constraints.

9.3.1 Encoding the Control Flow

In Section 9.1.1, we defined T_k as $isPositive(A_k+F_j-1)$ and R_k as $isPositive(A_k-F_j)$. Since $0 \leq A_k \leq 1$ and $0 \leq F_j \leq 1$, it is easy to verify that this is equivalent to respectively imposing the constraint $A_k + F_j - 1 \leq 2 \cdot T_k \leq A_k + F_j$ and $A_k - F_j \leq 2 \cdot R_k \leq A_k - F_j + 1$.

9.3.2 Encoding the Step Function

Now we discuss the difficulty of encoding the flag F_j using integer linear constraints with rational coefficients only. The following lemma states that such encoding is not possible.

LEMMA 9.3.1. *Given non-negative integer variables X and F, it is impossible to define a system of integer linear constraints Ψ (with rational coefficients) over X, F, and possibly other integer variables, such that $\Psi \wedge (X = 0) \rightarrow (F = 0)$ and $\Psi \wedge (X > 0) \rightarrow (F = 1)$.*

Proof. The proof relies on a theorem in [63] which states that the following piecewise linear function

$$f(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0, \end{cases}$$

where x is a non-negative *real* variable, cannot be defined as a minimization mixed integer programming (*MIP* for short) problem with rational coefficients only. More precisely, it is not possible to define $f(x)$ as

$$f(x) = \text{minimize } g \text{ w.r.t. } \Psi$$

where Ψ is a system of linear constraints with rational coefficients over x and other integer and real variables, and g is a linear function over $vars(\Psi)$. Now suppose that Lemma 9.3.1 is false, i.e., there exists Ψ such that $\Psi \wedge (X = 0) \rightarrow (F = 0)$ and $\Psi \wedge (X > 0) \rightarrow (F = 1)$, then the following *MIP* problem

$$f(x) = \text{minimize } F \text{ w.r.t. } \Psi \wedge (x \leq X)$$

defines the function $f(x)$, which contradicts the results in [63]. □

9.3.3 Undecidable Extensions

There are certain extensions of the *ILC* (with rational coefficients) model that allow our reduction to be carried out. Basically, the extensions should allow for encoding the flag F_j.

Using an Arbitrary Irrational Constant

The extension which we describe in this section allows the use of a single, arbitrary irrational number r (we do not require the specific value of r to represent any particular information). Thus, the coefficients are now over $\mathbb{Z} \cup \{r\}$. The variables still hold integers.

LEMMA 9.3.2. *Let r be an arbitrary positive irrational number, and let*

$$\begin{aligned} \Psi_1 \equiv &\quad (0 \leq F_j \leq 1) \wedge (F_j \leq X) \\ \Psi_2 \equiv &\quad (rX \leq B) \wedge (rY \leq A) \wedge (-Y \leq X) \wedge (A + B \leq F_j)\,. \end{aligned}$$

Then $(\Psi_1 \wedge \Psi_2 \wedge X = 0) \rightarrow F_j = 0$ and $(\Psi_1 \wedge \Psi_2 \wedge X > 0) \rightarrow F_j = 1$.

Proof. The constraint Ψ_1 force F_j to be 0 when X is 0, and when X is positive F_j can be either 0 or 1. The role of Ψ_2 is to eliminate the non-determinism for the case $X > 0$, namely, for $X > 0$ it forces F_j to be 1. The property that makes Ψ_2 work is that for a given *non-integer* number d, and two integers A and B, the condition $-A \leq d \leq B$ implies $A + B \geq 1$ for $d \neq 0$, whereas for an integer d the sum may be zero.

To prove the desired result, we first show that if $X = 0$, $F_j = 0$ is a solution. In fact, one can choose $B = A = Y = 0$ and all conditions are then fulfilled. Secondly, we consider $X > 0$. Note that rX is then a non-integer number, so necessarily $B > rX$. Similarly, $A > rY$, or equivalently $-A < r(-Y) \leq rX$. Thus, $-A < B$, and $A + B \leq F_j$ implies $0 < F_j$. Choosing $B = \lceil rX \rceil$, $Y = (-X)$ and $A = \lceil rY \rceil$ yields $A + B = 1$, so $F_j = 1$ is a solution. □

Remark: the variable Y was introduced in order to avoid using another irrational coefficient $(-r)$.

EXAMPLE 9.3.3. *Let us consider* $r = \sqrt{2}$ *in lemma 9.3.2. When* $X = 0$, Ψ_1 *forces* F_k *to be* 0*, and it is easy to verify that* Ψ_2 *is satisfiable for* $X = Y = A = B = F_k = 0$*. Now, for the positive case, let for example* $X = 5$*, then* Ψ_1 *limits* F_k *to the values* 0 *or* 1*, and* Ψ_2 *implies* $(\sqrt{2} * 5 \leq B) \wedge (-\sqrt{2} * 5 \leq A)$ *since* $Y \geq -5$*. The minimum values that* A *and* B *can take are respectively* -7 *and* 8*, thus it is not possible to choose* A *and* B *such that* $A + B \leq 0$*. This eliminates* $F_k = 0$ *as a solution. However, for these minimum values we have* $A + B = 1$ *and thus* $A + B \leq F_k$ *is satisfiable for* $F_k = 1$*.*

THEOREM 9.3.4. *The termination of ILC loops where the coefficients are from* $\mathbb{Z} \cup \{r\}$*, for a single arbitrary irrational constant* r*, is undecidable.*

We have mentioned, above, Meyer's result that *MIP* problems with rational coefficients cannot represent the step function over reals. Interestingly, he also shows that it *is* possible using an irrational constant, in a manner similar to our Lemma 9.3.2. Our technique differs in that we do not make use of minimization or maximization, but only of constraint satisfaction, to define the function.

Using Sufficiently Large Constant

Our second extension is the use of sufficiently large constants ∞ with the properties $0 * \infty = 0$ and $n * \infty = \infty$ for $X > 0$. Using this constant the flag F_k can be defined as stated by the following lemma.

LEMMA 9.3.5. *Let* $\Psi = 0 \leq F_k \leq 1 \wedge F_k \leq X \wedge 0 \leq X \leq F_k * \infty$*, then* $\Psi \wedge X = 0 \rightarrow F_k = 0$ *and* $\Psi \wedge X > 0 \rightarrow F_k = 1$

The proof of the above lemma is straightforward.

THEOREM 9.3.6. *The termination of ILC loops, over* $\mathbb{Z}_+$*, with a sufficiently large constant* ∞ *is undecidable.*

The use of such constants is common in (mixed) integer programming, for the very particular purpose of modeling piecewise linear functions.

9.4 A Lower Bound for Integer Linear-Constraints Loops

Let us consider a counter machine as defined in Section 8.3, but with a *weak* conditional statement "*if* $X_j > 0$ *then* k_1 *else* k_2" which is interpreted as: if X_j is positive then the execution may continue to *either* label k_1 or label k_2, otherwise, if it is zero, the execution *must* continue at label k_2. This computational model is equivalent to a Petri net. From considerations as those presented in Section 9.3, we arrived at the conclusion that the weak conditional, and therefore Petri nets, *can* be simulated by an *ILC* loop with rational coefficients. In this section, we describe this simulation and its implications.

A (place/transition) Petri net [69] is composed of a set of counters $X_1, \ldots, X_n$ (known as *places*) and a set of transitions $t_1, \ldots, t_m$. A transition is essentially a command to increment or decrement some places. This may be represented formally by associating with transition t its set of decremented places ${}^\bullet t$ and its set of incremented places $t^\bullet$. A transition is said to be enabled if all its decremented places are non-zero, and it can then be *fired*, causing the decrements and increments associated with it to take place. Starting from an initial marking (values for the places), the state of the net evolves by repeatedly firing one of the enabled transitions.

LEMMA 9.4.1. *Given a Petri net P with initial marking M, a simulating ILC loop (with rational coefficients) with an initial condition Ψ_M can be constructed in polynomial time, such that the termination of the loop from an initial state in Ψ_M is equivalent to the termination of P starting from M.*

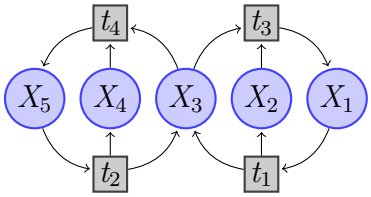
Proof. The *ILC* loop will have variables $X_1, \ldots, X_n$ that represent the counters in a straight-forward way, and flags $A_1, \ldots, A_m$ that represent the choice of the next transition much as we did for counter programs. The body of the loop is

$\Delta \wedge \Psi \wedge \Phi$ where

$$\begin{aligned}
\Delta &= \bigwedge_{k=1}^{m} (A'_k \geq 0) \wedge (A'_1 + ... + A'_m = 1) \\
\Psi &= \bigwedge_{i=1}^{n} (X_i \geq \sum_{k:i \in {}^\bullet t_k} A'_k) \\
\Phi &= \bigwedge_{i=1}^{n} (X'_i = X_i - \sum_{k:i \in {}^\bullet t_k} A'_k + \sum_{k:i \in t_k{}^\bullet} A'_k)
\end{aligned}$$

The loop guard is $X_1 \geq 0 \wedge \cdots \wedge X_n \geq 0$. The initial state Ψ_M simply forces each X_i to have the value as stated by the initial marking M. Note that the initial values of A_i are not important since they are not used (we only use A'_k). As before, the constraint Δ ensures that one and only one of the A'_k will equal 1 at every iteration. The constraint Ψ ensure that A'_k may receive the value 1 only if transition k is enabled in the state. The constraint Φ (the update) simulates the chosen transition. □

EXAMPLE 9.4.2. *Consider the following Petri net*



which has 5 *places* $X_1, \ldots, X_5$ *and* 4 *transitions* $t_1, \ldots, t_4$. *The translation, as described above, of this net to an ILC loop results in*

```
1 while( X_1 ≥ 0 ∧ X_2 ≥ 0 ∧ X_3 ≥ 0 ∧ X_4 ≥ 0 ∧ X_5 ≥ 0 ) do {
2   A'_1 ≥ 0 ∧ A'_2 ≥ 0 ∧ A'_3 ≥ 0 ∧ A'_4 ≥ 0 ∧ A'_1 + A'_2 + A'_3 + A'_4 = 1 ∧
3
4   X_1 ≥ A'_1 ∧
5   X_2 ≥ A'_3 ∧
6   X_3 ≥ A'_3 + A'_4 ∧
7   X_4 ≥ A'_4 ∧
8   X_5 ≥ A'_2 ∧
9
```

$$X_1' = X_1 + A_3' - A_1' \wedge$$
$$X_2' = X_2 + A_1' - A_3' \wedge$$
$$X_3' = X_3 + A_1' + A_2' - A_3' - A_4' \wedge$$
$$X_4' = X_4 + A_2' - A_4' \wedge$$
$$X_5' = X_5 + A_4' - A_2'$$
}

Line 2 *corresponds to* Δ*, lines 4–8 to* Ψ*, and lines 10–14 to* Φ*.*

The importance of this result lies in the fact that complexity results for Petri net are now lower bounds on the complexity of the corresponding problems for *ILC* loops, and in particular, from a known result about the termination problem [37, 57], we obtain the following.

THEOREM 9.4.3. *The termination problem for ILC loops (with rational coefficients), for a given input, is at least EXPSPACE-hard.*

Note that the reduction does not provide useful information on universal termination of *ILC* loops with rational coefficients, since universal termination of Petri nets (also known as *structural boundedness* is PTIME-decidable [61, 38].

9.5 A Lower Bound for Deterministic Updates

The *ILC* loop we constructed to prove Theorem 9.4.3 *was* non-deterministic, but we will now show that the result also holds for loops which are deterministic (though defined by constraints). The result will require, however, that the loop precondition be non-deterministic, that is, we ask about termination for a set of states, not for a single state (and not for all possible states, either).

To explain the idea, we look at the Petri nets constructed in Lipton's hardness proof. This proof is a reduction from the halting problem for counter programs with a certain space bound (note that the halting problem for a space-bounded model is the canonical complete problem for a space complexity class). Given a counter program P, the reduction constructs a Petri net N_P that has the following behavior when started at an appropriate initial state. N_P has two kinds of computations, *successful* and *failing*. Failing computations are caused by taking

non-deterministic branches which are not the correct choice for simulating P. Failing computations always halt. The (single) successful computation simulates P faithfully. If (and only if) P halts, the successful computation reaches a state in which a particular flag, say *HALT*, is raised (that is, incremented from 0 to 1). This flag is never raised in failing computations.

This network N_P can be translated into an *ILC* loop L_P as previously described. We eliminate the non-determinism from L_P by using an unconstrained input variable O as an oracle, to guide the non-deterministic choices. In addition, we reverse the program's behaviour: our loop will terminate (on all states of interest) if and only if P does *not* terminate (note that P is presumably input-free and deterministic).

THEOREM 9.5.1. *The termination problem for ILC loops (with rational coefficients), for a partially-specified input, is at least EXPSPACE-hard, even if the update is deterministic.*

We describe the changes to the previous reduction. We use assignment commands for convenience. We will later show that they can all be translated into linear constraints. We assume that N_P has m transitions and n places. The construction of L_P is obtained as in the previous reduction with the following changes: (1) we introduce a new variable O, and include $O > 0$ in the loop guard; and (2) Δ is replaced by

$$\begin{aligned} PC &:= O \bmod (m+2) \\ O &:= O \operatorname{div} (m+2) \\ A_k &:= [PC = k] \qquad (\text{for all } 1 \le k \le m+1) \\ O &:= O + (m+1) \cdot HALT \end{aligned}$$

The notation $[PC = k]$ means 1 if the $PC = k$ and 0 otherwise. Also, A_{m+1} is a new flag which is not associated with any transition of N_P; it represents a do-nothing transition (the iteration does, however, decrease O).

Let Ψ_M be $HALT = 0 \wedge X_1 = a_1 \wedge \cdots \wedge X_n = a_n \wedge O > 0$ where a_i is the initial value of place X_i in M. We claim that L_P terminates for all input in Ψ_M if and only if N_P does not terminate for M (or equivalently, P does not halt).

Clearly, O guides the choice of transitions. It makes our loop deterministic, but any sequence of net transitions can be simulated: Suppose this sequence is $k_1, k_2, \ldots, k_n$. An initial value for O of $k_1 + (k_2 + (k_3 + \cdots) \cdot (m+2)) \cdot (m+2)$ will cause exactly these transitions to be taken. As long as $HALT$ is not set, O also keeps descending. Since the loop condition includes $O > 0$, a non-halting simulation will become a terminating loop. A halting simulation will reach the point where $HALT = 1$, provided the initial value of O indicated the correct execution trace. Note that O reaches the value 0 exactly when $HALT$ is set. In this iteration, only A_{m+1} is set (so counters will not be modified), while O is restored to $m+1$. In the next iteration, O remains $m+1$, A_{m+1} is set, and $HALT$ is set. Thus, the loop will not terminate.

Finally, the above assignments can be translated to integer linear constraints as follows:

$$(O = (m+2) \cdot O'' + PC') \wedge (1 \leq PC' \leq m+1) \wedge$$
$$(\textstyle\bigwedge_{i=1}^{m+1} A'_i \geq 0) \wedge (1 = A'_1 + \cdots + A'_{m+1}) \wedge (PC' = 1 \cdot A'_1 + \cdots + (m+1) \cdot A'_{m+1}) \wedge$$
$$O' = O'' + (m+1) \cdot HALT'$$

9.6 Concluding remarks

In this chapter we have studied the complexity of deciding termination of some form of simple integer loops. For some we have proved undecidability and for some others we provided an EXPSPACE-hardness lower bound. The most remarkable results that we have achieved are: (i) a single conditional statement in the body of a while loop is enough to make the problem undecidable; (2) a single irrational constraint (or sufficiently large constraint) make the problem undecidable for integer linear-constraint loops.

Due to the strong relation between cost and termination analysis, the results obtained in this part of the thesis have the following consequences on decidability and complexity of cost analysis:

1. The undecidability of termination for *IPL* loops implies that there are certain classes of programs for which inference of cost bounds is not decidable.

Namely, programs that include simple loops (without branching in the loop body) and allowing statements that are sufficient for simulating the step function, e.g., integer division by constant. Note that when all updates inside such loops are linear, and homogeneous, then, termination is decidable [25]. However, this does not imply decidability of the cost bound problem since, in addition to proving termination, we need infer an upper bound on the number of iterations.

2. The undecidability of simple loops with two linear pieces implies that the cost problem for *CRs* with multiple equations, such as those of Figure 4.3, is undecidable already for the case of two recursive equations.

3. The EXPSPACE-hardness lower bound for *ILC* loops with a given or partially specified input implies that the cost bound problem for *CRs* of the form specified in Figure 4.1, when the input is (partiality) specified, is also at least EXPSPACE-hard.

Part III

Conclusions, Related and Future work

Chapter 10

Related Work

In this chapter we overview works related on cost and termination analysis, and discuss their relations to the results obtained in this thesis. In Section 10.1 we overview related work on cost analysis for several programming paradigms, and in Section 10.2 we discuss related work on termination analysis.

10.1 Related Work on Cost Analysis

Since the seminal work of Wegbreit [80] on mechanical cost analysis, there have been an increasing interest in static cost analysis for different programming paradigms. Different works addressed different aspects of cost analysis, such as the kind of resources (e.g., memory, executed instructions), the type of bounds (e.g., best, worst or average case), precision, and efficiency. As we have mentioned in Chapter 1, the classical approach to cost analysis consists of two phases. In the first phase, an *abstract version of the program* is generated, which includes only information relevant to capturing its cost; in the second phase this abstract program is analysed in order to compute closed-form bounds in terms of (an abstract version of) the input parameters. In this thesis we concentrated on the second phase.

In our work, the abstract programs generated in the first phase are called *cost relations* (*CRs*), a terminology that we borrowed from [5]. However, there is no unified terminology or syntax for these abstract programs, and different works call

them with different names, e.g., *worst-case complexity functions* [2], *time-bound programs* [70] and *recursive time-complexity functions* [54]. Moreover, in some cost analysis frameworks the border between the two phases is not clearly drawn, and thus it is difficult to characterize the corresponding abstract programs. The expressiveness of these abstract programs, as well as the kind of analysis applied on them to generate closed-form bounds, directly affect the applicability and precision of the corresponding approach. In what follows, we discuss the most related works of (the second phase of) cost analysis from different perspectives: (1) the kind of abstract programs generated in the first phase; (2) the kind of analyses performed on such abstract programs in order to compute closed-form bounds; (3) the type of bounds (UBs or LBs) generated; and (4) the precision of the computed bounds.

The most related approach to our work is [5], where *CRs* were actually introduced. As we have seen in the first part of this thesis we rely on some of their underlying techniques such as inference of ranking functions and maximization of cost expressions. Although experimentally our approach is more precise (as we have seen in Chapter 6), we cannot prove theoretically that it is always more precise. However, for the case of *CRs* with a single recursive equation as described in sections 4.1 and 4.2, if we use the same ranking functions and maximization procedures as [5], then it is guaranteed that our approach is more precise. For the case of *CRs* with multiple recursive equations, it is not possible to formally compare them. Indeed, one could handcraft examples for which [5] infers more precise UBs. This is because for solving such cases: (1) our first alternative, which generalizes cost expressions, is based on heuristics and thus might be imprecise in some cases; and (2) our second alternative, which analyzes each recursive equation separately, requires inferring the number of visits to a single equation which can be less precise than inferring ranking functions. As regards applicability, when it is not possible to infer the progression parameters (in definitions 4.2.3 and 4.2.10), we use the approach of [5], i.e., replacing the corresponding $\|l\|$ by $\|\hat{l}\|$, thus, assuming that *CAS* is able to handle the corresponding *RRs*, we achieve a similar applicability.

The work of [42], in the context of the SPEED project, computes *worst-case* symbolic bounds for C++ code containing loops and recursions. The loops in the

input code are instrumented with counters, and the intermediate representation is based on what they call counter-optimal proof structure. This structure consists of a set of counters and linear invariants generated for these counters. The invariants are used to bound the counters, and then these bounds are composed into the final desired bound. While the proof structure is fundamentally different from *CRs*, we observe the following: (1) The number of counters in a proof structure is equal to the number of different *CR* in the corresponding *CRs*; and (2) each counter in the proof structure represents the upper bound cost of each *CR* in the corresponding *CRs*. Using the above observations we can conclude that it might infer bounds that are less precise than ours since, as explained in [42] for instance, the worst-case time usage $\sum_{i=1}^{n} i$ is over-approximated by n^2 in this approach, while our approach is able to obtain the precise solution $\frac{n^2}{2} - \frac{n}{2}$. Unfortunately we cannot experimentally compare to this approach since the code is not available for use.

King et al. [51] addressed the problem of inferring conditions (on the input) of a given logic program. It is guaranteed that the number of resolution steps will exceed a predefined amount if the input conditions are satisfied while executing the program. The abstract compilation used in this work generates $CLP(R)$ programs which are similar to *CRs* where (1) each abstract predicate includes a counter that accumulates the corresponding cost, and (2) some constraints are added to state that the counter (i.e., the resources) *cannot exceed* the predefined amount. The conditions are inferred by first over-approximating the input-output semantics of the $CLP(R)$ program, and then examine it to see which (abstract) inputs lead to failure. These cases either fail because of a failure in the original programs, or because the amount of resources exceed the predefined amount. Importantly, this method does not address the problem of inferring symbolic LBs.

There is a series of works [33, 34, 35, 66] on resource analysis of logic programs which are based on generating and solving recurrence equations. Debray et al. [33] developed a semi-automatic worst-case cost analysis for logic programs which generates both linear and nonlinear recurrence equations. However, it is not guaranteed that it will always be able to compute closed-form bounds for the nonlinear case. In a successive work, Debray et al. [35] developed a semi-

automatic method for deriving nontrivial lower bounds on the computation cost of logic programs. The work of Navas et al. [66] infers resource bounds (both upper and lower) for logic programs by generating two sets of difference equations. Navas et al. [65] developed a resource usage analysis for Java bytecode in a similar fashion. Since all these methods are based on solving recurrence equations, the inferred bounds are as precise as ours and the approach is not limited to specific complexity classes. However, since these approaches are based on using CAS, they suffer from the shortcomings described in Section 3.2.

In the functional programming setting, the most related works are [47, 46, 45], which are centered on the static inference of UBs on the resource usage of first-order functional programs. Automatic amortized resource usage of a first-order functional language was introduced by Hofmann and Jost in [47]. It is based on a type system in which the types are the potential functions used in amortized analysis [77]. The type inference is done using linear programming techniques. It is important to note that this technique is limited to the inference of linear UBs. This work has been extended in [46] for univariate polynomial UBs. However, such polynomial cannot express bounds of the form $m*n$, and thus they are over-approximated by $n^2 + m^2$. Recently, in [45], techniques for handling multivariate polynomial UBs, such as $m * n$, have been proposed.

Hofmann and Rodriguez [48] developed a type system for object-oriented Java like languages and was extended in [49] to include to support amortized complexity analysis much like [47, 46, 45]. Note that all these approaches cannot handle programs whose resource usage depend on integer variables. While these techniques can be adapted to handle *CRs* with simple integer linear constraints, it is not clear how it can be extended to handle *CRs* with unrestricted form of integer linear constraints. It is also important to note that currently these techniques cannot compute *logarithmic* or *exponential* UBs. For example, [45] computes $O(n^2)$ as an UB for the `mergesort` program whereas we compute $O(n * log(n))$. On the other hand, these techniques are superior for examples that exhibit amortized cost behavior, but such examples are out of the scope of this thesis since they cannot be modeled precisely with *CRs* [10]. Overall, we believe that our approach is more generic (at least for imperative languages), in the sense that it handles *CRs* with arbitrary integer linear constraints, which might be the output

of cost analysis of any programming language, and, in addition, it is not restricted to any complexity class.

There are some works related on solving recurrence relations [28, 14, 41, 39]. Cohen and Katcoff [28] developed interactive techniques based on guessing solutions and generating functions that solve linear recurrence relations. However, their method does not always admit closed-form expressions. Decision procedures have been developed by Gosper [41] and Everest et al. [39] which admits closed-form expression for a subset of linear recurrence relations. Bagnara et al. [14] extended the previous techniques to recurrence relations with multiple arguments and some nonlinear recurrences of finite order. However, in spite of this extension, still the form of the recurrence relations that can be solved are very limited, when compared to our *CRs*.

10.2 Related Work on Termination

Termination of integer loops has received considerable attention recently, both from theoretical (e.g., decidability, complexity), and practical (e.g., developing tools) perspectives. Research has addressed straight-line while loops as well as loops in a constraint setting, possibly with multiple paths.

For straight-line while loops, the most remarkable results are those of [78] and [25]. Tiwari proved that the problem is decidable for linear deterministic updates when the domain of the variables is $\mathbb{R}$. Braverman proved that this holds also for $\mathbb{Q}$, and for the homogeneous case it holds for $\mathbb{Z}$ (see Section 7.1). Both considered universal termination, the termination for a given input left open.

Decidability and complexity of termination of single and multiple-path integer linear-constraint loops has been intensively studied for different classes of constraints. [55] proved that termination of a multiple-path *ILC* loop, when the constraints are restricted to size-change constraints (i.e., constraints of the form $X_i > X'_j$ or $X_i \geq X'_j$ over $\mathbb{N}$), is PSPACE-complete. [19, 18] identified sub-classes of such loops for which the termination can be decided in, respectively, PTIME and NPTIME. [16] extended the types of constraints allowed to *monotonicity constraints* of the form $X_i > Y$, $X_i \geq Y$, where Y can be a primed or unprimed

variable. Termination for such loops is, again, PSPACE-complete. All the above results involving size-change or monotonicity constraints apply to an arbitrary well-founded domain, although the hardness results only assume $\mathbb{N}$. Monotonicity constraints over $\mathbb{Z}$ were considered in [27, 17], concluding that this termination problem too is PSPACE-complete. Recently, [23] proved that it is still PSPACE-complete for gap-constraints, which are constraints of the form $X - Y \geq c$ where $c \in \mathbb{N}$. In a similar vein, [15] proved that for general difference constraints over the integers, i.e., constraints of the form $X_i - X'_j \geq c$ where $c \in \mathbb{Z}$, the termination problem becomes undecidable. However for a subclass in which each target (primed) variable might be constrained only once (in each path of a multiple-path loop) the problem is PSPACE-complete.

All the above work concerns multiple-path loops. Recently, [22] showed that (universal) termination of a single *ILC* loop with octagonal relations is decidable. Petri nets and various extensions, such as *reset and transfer nets*, can also be seen as multiple-path constraint loops. The termination (for a given input) of place/transition Petri nets and certain extensions is known to be decidable [68, 36].

A related topic that received much attention is the synthesis of ranking functions for such loops, as a means of proving termination. [75] proposed a method for the synthesis of linear ranking functions for (single path) *ILC* loops over $\mathbb{N}$. Later, their method was extended by [62] to $\mathbb{Q}$ and to multiple-path loops. Both rely on the duality theorem of linear programming. [67] also proposed a method for synthesizing linear ranking function for *ILC* loops. Their method is based on Farkas' lemma, which has been used also in [29] for synthesizing linear ranking functions. It is important to note that these methods are complete with respect to synthesizing linear ranking functions when the variables range over $\mathbb{R}$ or $\mathbb{Q}$, but not $\mathbb{Z}$. Recently, [12] proved that [62, 67] are actually equivalent, in the sense that they compute the same set of ranking functions, and that the method of Podelski and Rybalchenko can, potentially, be more efficient since it requires solving rational constraints systems with fewer variables and constraints. [24] presented an algorithm for computing linear ranking functions for straight-line integer while loops with integer division.

Piecewise affine functions have been long used to describe the step of a discrete

time dynamical system. [21] considered systems of the form $x(t+1) = f(x(t))$ where f is a piecewise affine function over $\mathbb{R}^n$ (defined by rational coefficients). They show that some problems are undecidable for $n \geq 2$, in particular, whether all trajectories go through 0 (the mortality problem). This can be seen as termination of the loop `while x` $\neq$ `0 do x := f(x)`.

Chapter 11

Conclusions and Future work

In this thesis we have considered precision, scalability and applicability issues in cost and termination analysis, both from practical and theoretical perspectives. Our main interest was in developing cost analysis techniques that (i) overcome the limitations of existing approaches; and (ii) have a good performance/precision tradeoff. From the practical point of view, we have developed such techniques for solving cost relations, which is the phase of cost analysis where most of the precision, scalability and applicability problems can be found in existing tools. From the theoretical side, since our techniques heavily rely on deciding termination of loops, we have studied the computational complexity of deciding termination for some form of simple loops that arise in the context of cost analysis. This theoretical study gives an insight on the difficulty of the problems under consideration, and thus on the practicality on any algorithm the aims at solving them.

As for the practical side of this thesis, we have proposed a novel approach to infer precise UBs and LBs of *CRs* which, as our experiments show, achieves a very good balance between the accuracy of our analysis and its applicability. The main idea is to automatically transform *CRs* into a simple form of worst-case (resp. best-case) *RRs* that *CAS* can accurately solve to obtain UBs (resp. LBs) on the resource consumption. The required transformation is far from trivial since it requires transforming multiple recursive nondeterministic equations involving multiple increasing and decreasing arguments into a single deterministic equation with a single decreasing argument. It is important to note that it is the first time

that the problem of inferring LBs is addressed for such wide setting.

Importantly, since *CRs* are a universal output of cost analysis for any programming language, our approach to infer closed-form UBs and LBs is completely independent of the programming language from which *CRs* are obtained. Currently, we have applied it to *CRs* obtained from Java bytecode programs, from X10 programs [9] and from actor-based programs [3]. In the latter two cases, the languages have concurrency primitives to spawn asynchronous tasks and to wait for termination of tasks. In spite of being concurrent languages, the first phase of cost analysis handles the concurrency primitives and the generated *CRs* can be solved directly using our approach.

As for the theoretical side, which deals with deciding termination of simple loops, for straight-line while loops, we have proved that if the underlying instruction set allows the implementation of a simple piecewise linear function, namely the step function, the termination problem is undecidable. For integer linear-constraint loops, we have shown that allowing the constraints to include a single arbitrary irrational number makes the termination problem undecidable. For the case of integer constraints loops with rational coefficients only, we could simulate a Petri net. This result provides interesting lower bounds on the complexity of the termination, and other related problems, of *ILC* loops. For example, since marking equivalence (equality of the sets of reachable states) is undecidable for Petri nets [43, 38, 50], it follows that equivalence (in terms of the reachable states) of two *ILC* loops with given initial states is also undecidable, which in turn implies that the reachable states of an *ILC* loop are not expressible in a logic where equivalence is decidable. We think that our results shed some light on the termination problem of simple integer loops and perhaps will inspire further progress on the open problems.

As future work, we plan to assess the scalability of our cost analysis approach by analyzing larger programs, up to now the main concern has been the accuracy of the results obtained. Also, we plan to study new techniques to infer more precise lower/upper bounds on the number of iterations that loops perform. As this is an independent component, our approach will directly be benefited from any improvement in this regard. In addition, so far we have used linear invariants for inferring linear ranking functions, minimum number of iterations of a loop

and maximization or minimization of cost expressions. Another extension of our work would be inferring nonlinear loop invariants using symbolic summation and algebraic techniques. Another possible direction is inferring nonlinear input-output (size) relations for methods by viewing the output as the cost that is consumed by the corresponding method. This way, we can view the problem of inferring such input-output relations as solving corresponding *CRs*, for which we already know how to infer nonlinear bounds. Note that these input-output relations are fundamental in the first phase of cost analysis in order to generate *CRs* that precisely capture the program's cost.

Bibliography

[1] A. Adachi, T. Kasai, and E. Moriya. A theoretical study of the time analysis of programs. In J. Becvár, editor, *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 201–207. Springer, 1979.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[4] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

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

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2007.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In *9th International School on Foundations of Security Analysis and Design (FOSAD'09)*, number 5705 in Lecture Notes in Computer Science, pages 258–288. Springer, 2009.

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

[9] E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-level analysis for a language with async/finish parallelism. In J. Vitek and B. D. Sutter, editors, *LCTES*, pages 21–30. ACM, 2011.

[10] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In A. Miné and D. Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012.

[11] R. Bagnara, P. M. Hill, and E. 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.

[12] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.

[13] R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report, University of Parma, 2005. `arXiv:cs/0512056` available from `http://arxiv.org/`.

[14] R. Bagnara, A. Zaccagnini, and T. Zolo. The automatic solution of recurrence relations. I. Linear recurrences of finite order with constant coef-

ficients. Quaderno 334, Dipartimento di Matematica, Università di Parma, Italy, 2003. Available at `http://www.cs.unipr.it/Publications/`.

[15] A. M. Ben-Amram. Size-Change Termination with Difference Constraints. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.

[16] A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. *Logical Methods in Computer Science*, 6(3), 2010.

[17] A. M. Ben-Amram. Monotonicity constraints for termination in the integer domain. *Logical Methods in Computer Science*, 7(3), 2011.

[18] A. M. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In C. Ramakrishnan and J. Rehof, editors, *14th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5028 of *LNCS*, pages 46–55. Springer, 2008.

[19] A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.

[20] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.

[21] V. D. Blondel, O. Bournez, P. Koiran, C. H. Papadimitriou, and J. N. Tsitsiklis. Deciding stability and mortality of piecewise affine dynamical systems. *Theor. Comput. Sci.*, 255(1-2):687–696, 2001.

[22] M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. In *TACAS'12*, 2012. To appear.

[23] L. Bozzelli and S. Pinchinat. Verification of gap-order constraint abstractions of counter systems. In V. Kuncak and A. Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2012.

[24] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer, 2005.

[25] M. Braverman. Termination of integer linear programs. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.

[26] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[27] M. Codish, V. Lagoon, and P. J. Stuckey. Testing for termination with monotonicity constraints. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2005.

[28] J. Cohen and J. Katcoff. Symbolic solution of finite-difference equations. *ACM Trans. Math. Softw.*, 3(3):261–271, Sept. 1977.

[29] M. Colón and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001.

[30] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI), Ottawa, Canada*, pages 415–426. ACM, 2006.

[31] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *The 5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[32] K. Crary and S. Weirich. Resource Bound Certification. In *Proc. of POPL'00*, pages 184–198. ACM, 2000.

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

[34] S. K. Debray, P. López-García, M. V. Hermenegildo, and N.-W. Lin. Estimating the computational cost of logic programs. In *SAS*, pages 255–265, 1994.

[35] S. K. Debray, P. López-García, M. V. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic programs. In *ILPS*, pages 291–305, 1997.

[36] C. Dufourd, P. Jancar, and P. Schnoebelen. Boundedness of reset p/t nets. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *ICALP*, volume 1644 of *Lecture Notes in Computer Science*, pages 301–310. Springer, 1999.

[37] J. Esparza. Decidability and complexity of Petri net problems—an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets, Vol. I: Basic Models*, volume 1491 (Volume I) of *Lecture Notes in Computer Science (LNCS)*, pages 374–428. Springer-Verlag (New York), Dagstuhl, Germany, Sept. 1996, revised paper 1998.

[38] J. Esparza and M. Nielsen. Decidability issues for petri nets. Technical Report RS-94-8, BRICS, Department of Computer Science, University of Aarhus, 1994.

[39] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence sequences*. American Mathematical Society, Providence, R.I. :, 2003.

[40] P. Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[41] R. W. Gosper, Jr. Decision procedure for indefinite hypergeometric summation. *Proceedings of the National Academy of Sciences of the United States of America*, 75(1):40–42, 1978.

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

[43] M. Hack. Decidability questions for Petri nets. Technical Report MIT/LCS/TR-161, Massachusetts Institute of Technology, June 1976.

[44] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[45] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *The 38th Symposium on Principles of Programming Languages (POPL'11)*, pages 357–370. ACM, 2011.

[46] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *The 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[47] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, 2003.

[48] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *In ESOP 2006, LNCS 3924*, pages 22–37. Springer, 2006.

[49] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[50] Jançar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995. Selected Papers of the Eleventh Symposium on Theoretical Aspects of Computer Science.

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

[52] L. Kovacs. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema*. PhD thesis, RISC, Johannes Kepler University Linz, Austria, October 2007. RISC Technical Report No. 07-16.

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

[54] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[55] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Hankin and D. Schmidt, editors, *Symposium on Principles of Programming Languages, POPL'01*, pages 81–92. ACM, 2001.

[56] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of Prolog programs. In L. Naish, editor, *International Conference on Logic Programming, ICLP'97*, pages 64–77. MIT Press, 1997.

[57] R. J. Lipton. The reachability problem requires exponential space. Technical Report 63, Yale University, 1976. Available at `http://www.cs.yale.edu/publications/techreports/tr63.pdf`.

[58] B. Luca, S. Andrei, H. Anderson, and S.-C. Khoo. Program transformation by solving recurrences. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*, pages 121–129. ACM, 2006.

[59] Matiyasevich. Hilbert's tenth problem: What was done and what is to be done. In Denef, Lipshitz, Pheidas, and V. Geel, editors, *Hilbert's Tenth Problem: Relations with Arithmetic and Algebraic Geometry, AMS, 2000.* 2000.

[60] Maxima, a Computer Algebra System., 2009. `http://maxima.sourceforge.net`.

[61] G. Memmi and G. Roucairol. Linear algebra in net theory. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 213–223. Springer Berlin / Heidelberg, 1980.

[62] F. Mesnard and A. Serebrenik. Recurrence with affine level mappings is p-time decidable for clp(r). *TPLP*, 8(1):111–119, 2008.

[63] R. R. Meyer. Integer and mixed-integer programming models: General properties. *Journal of Optimization Theory and Applications*, 16:191–206, 1975. 10.1007/BF01262932.

[64] M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[65] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[66] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[67] A. Podelski and A. Rybalchenko. A complete Method for the Synthesis of Linear Ranking Functions. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*, pages 465–486. Springer, 2004.

[68] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, Apr. 1978.

[69] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[70] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 144–156, New York, NY, USA, 1989. ACM.

[71] A. Rybalchenko. Constraint solving for program verification: Theory and practice by example. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 57–71. Springer, 2010.

[72] D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[73] C. Schneider. Finding telescopers with minimal depth for indefinite nested sum and product expressions. In M. Kauers, editor, *ISSAC*, pages 285–292. ACM, 2005.

[74] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[75] K. Sohn and A. V. Gelder. Termination detection in logic programs using argument sizes. In *PODS*, pages 216–226. ACM Press, 1991.

[76] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[77] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[78] A. Tiwari. Termination of linear programs. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–390. Springer Berlin / Heidelberg, 2004.

[79] P. Wadler. Strictness Analysis Aids Time Analysis. In *ACM Symposium on Principles of Programming Languages (POPL'88)*. ACM Press, 1988.

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

[81] J. Wielemaker. SWI-Prolog Home Page. http://www.swi-prolog.org/.