Cost Analysis of Programs Based on the Refinement of Cost Relations

Kostenanalyse von Programmen durch die Verfeinerung von Kostenrelationen
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Antonio Flores Montoya M.Sc. aus Madrid, Spanien
Tag der Einreichung: 30.05.2017, Tag der Prüfung: 14.07.2017
Darmstadt 2017— D 17

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Jürgen Giesl

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Software Engineering

Cost Analysis of Programs Based on the Refinement of Cost Relations
Kostenanalyse von Programmen durch die Verfeinerung von Kostenrelationen

Genehmigte Dissertation von Antonio Flores Montoya M.Sc. aus Madrid, Spanien

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Jürgen Giesl

Tag der Einreichung: 30.05.2017
Tag der Prüfung: 14.07.2017

Darmstadt 2017 — D 17

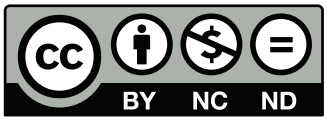
Wissenschaftlicher Werdegang
Doktorand am Fachgebiet Software Engineering der Technischen Universität Darmstadt
von September 2012 bis Juli 2017

Master Informatik Forschung
Complutense Universität Madrid, Spanien
von September 2011 bis Juni 2012

Informatikingenieur
Complutense Universität Madrid, Spanien
von September 2006 bis Juni 2011

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-67465
URL: http://tuprints.ulb.tu-darmstadt.de/6746

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Acknowledgements

First, I am grateful to my PhD advisor Reiner Hähnle for his support and guidance. He gave me the freedom to work on the problems that I found interesting and, at the same time, he always provided me with interesting suggestions on how to apply my research in novel ways. I also have to thank him for showing me the importance of writing scientific documents that are not only correct, but also clear and understandable. If this thesis is comprehensible, it is definitely thanks to him.

I thank Jürgen Giesl for being my second reviewer and for his detailed feedback and Richard Bubel for giving me early feedback on some chapters of this thesis.

I am grateful to all my colleagues from the software engineering department at TU Darmstadt. They created a great atmosphere at work, and made the overall experience much more fun. I am grateful as well to Florian Zuleger and Moritz Sinn for welcoming me in Vienna during my internship. We had many interesting discussions together that shaped the way I approach research problems. I thank Clemens Danninger as well. He found plenty of bugs and suggested several improvements in CoFloCo.

Finally, I thank my parents. I own my mother my fascination for science and I own my father my love for solving problems. I thank my family, my friends, and my girlfriend Sauvanithi for their immense support and encouragement throughout my PhD. Without them I would have never had the strength to finish this thesis.

Abstract

Cost analysis aims at statically inferring the amount of resources, such as time or memory, needed to execute a program. This amount of resources is the cost of the program and it depends on its input parameters. Obtaining a function (in terms of the input parameters) that represents the cost of a program precisely is generally not possible. Thus, cost analyses attempt to infer functions that represent upper or lower bounds of the cost of programs instead.

Many existing cost analyses approach the problem in two stages. First, the target program is transformed into an integer abstract representation where the resource consumption is explicit and second, the abstract representation is analyzed and cost bounds are inferred from it. The advantage of this approach is that the second part is language independent and resource independent. That is, it can be reused across different programming languages and to analyze the program cost with respect to different resources. Cost relations are a possible abstract representation. They are similar to constraint logic programs annotated with costs and they can easily represent both imperative and functional programs.

Existing cost analyses based on cost relations have limited support for programs that have a complex control flow, or present amortized complexity, that is, when the sum of the cost of the parts yields a higher asymptotic cost than the cost of the whole. This thesis identifies these limitations, and presents a new analysis of cost relations that overcomes them.

The analysis can obtain upper and lower bounds of programs expressed as cost relations and it contains three parts:

1. The first part reduces any mutually recursive cost relations to cost relations that only have direct recursion and performs some simplifications.

2. The second part consists of a refinement of cost relations that partitions all possible executions of the program into a finite set of execution patterns named chains. The refinement also infers precise invariants for each of the chains, discards unfeasible execution patterns and proves termination.

3. In the third part of the analysis, cost bounds are inferred compositionally. For that purpose, a novel cost representation, named cost structures, is presented. Cost structures reduce the computation of complex bounds to the inference of simple constraints using linear programming. They can represent polynomial upper and lower bounds of programs with max and min operators.

The analysis is proven sound with respect to a new semantics of cost relations. This semantics distinguishes between terminating and non-terminating executions and models the behavior of non-terminating executions accurately.

In addition, the analysis has been implemented in the tool CoFloCo and it has been extensively evaluated against other state-of-the-art tools and with respect to a variety of benchmarks. These benchmarks include imperative programs, functional programs, and term rewrite systems. CoFloCo performs well in all categories which demonstrates both the power of the analysis and its versatility.

Zusammenfassung

Die Kostenanalyse von Programmen ermöglicht es, den für die Ausführung eines Programmes notwendigen Ressourcenbedarf, wie zum Beispiel Zeit oder Speicher statisch, zu bestimmen. Den Ressourcenbedarf eines Programms bezeichnet man auch als die Kosten des Programms. Die Programmkosten hängen im Allgemeinen von Eingabeparametern (den Eingabedaten) ab. Das Bestimmen einer Funktion, die in Abhängigkeit von den Eingabeparametern die exakten Kosten eines Programms angibt, ist in der Regel nicht möglich. Stattdessen versucht man bei der Kostenanalyse Funktionen zu ermitteln, die obere und untere Schranken für die Kosten eines Programmes darstellen.

Viele der existierenden Ansätze zur Kostenanalyse gehen das Problem in zwei Stufen an. Zuerst wird das zu analysierende Programm in eine abstrakte Integer-Repräsentation überführt, in welcher der Ressourcenbedarf/-verbrauch explizit dargestellt ist. In der zweiten Stufe wird diese abstrakte Repräsentation analysiert und obere bzw. untere Schranken bestimmt. Der Vorteil dieses Ansatzes ist, dass die zweite Stufe unabhängig von der Programmiersprache und der betrachteten Ressource (Zeit, Speicher usw.) ist. Dies ermöglicht den Einsatz der für die zweite Stufe entwickelten Techniken und Werkzeuge zur Analyse von Programmen in unterschiedlichsten Programmiersprachen bzgl. des Verbrauchs unterschiedlich Ressourcen. Kostenrelationen bieten sich als eine Wahl für die abstrakte Repräsentationen der Programme an. Sie ähneln Programmen aus der *Constraint-logischen Programmierung*, die mit Kostenannotationen versehen sind. Kostenrelationen erlauben es ferner, funktionale sowie imperative Programme auf einfache Art zu repräsentieren.

Existierende Kostenanalysen haben eine Reihe von Nachteilen betreffend der Analyse von Programmen mit komplexem Kontrollfluss sowie bei der Bestimmung amortisierter Komplexität zur Beschreibung von Szenarien, bei denen die Gesamtkosten eines Programmes kleiner sind als die Summe der asymptotischen Einzelkosten. In dieser Arbeit werden die Einschränkungen der existierenden Ansätze identifiziert und eine neuartige Analyse entwickelt, die diese überwindet.

Die entwickelte Analyse bestimmt obere und untere Schranken für in Kostenrelationen überführte Programme. Sie besteht aus den folgenden drei Teilen:

1. Im ersten Teil der Analyse werden wechselseitig-rekursive Kostenrelationen in äquivalente Kostenrelationen überführt, die nur noch einfache Rekursionen enthalten. Ausserdem werden weitere Vereinfachungsschritte durchgeführt.

2. Im zweiten Teil werden die Kostenrelationen mit Hinblick auf den Kontrollfluss verfeinert. Die Verfeinerung partitioniert die Kostenrelationen in eine endliche Menge von Ausführungsmustern, sogenannten *Ketten* (*Chains*), die alle möglichen Ausführungen beschreibt. Die Verfeinerung gibt des weiteren präzise Invarianten für die einzelnen Ketten an, eliminiert nicht erreichbare Programmpfade und beweist die Terminierung des Programms.

3. Im dritten Teil der Analyse werden schließlich Schranken für die Kosten auf kompositionelle Art und Weise berechnet. Zu diesem Zweck wird eine neuartige Art der Kostenrepräsentation, genannt *Kostenstrukturen*, entwickelt und vorgestellt. Kostenstrukturen erlauben es, die Berechnung komplexer Schranken auf das Lösen einfacher Constraints mit Hilfe linearer Programmierung zu reduzieren. Kostenstrukturen können polynomielle obere und untere Schranken von Programmen repräsentieren und unterstützen dabei max- und min-Operatoren.

Die entwickelte Analyse wird als korrekt bzgl. einer neuen Semantik von Kostenrelationen bewiesen. Die im Rahmen dieser Arbeit entwickelte neue Semantik erlaubt es, zwischen terminierenden und nichtterminierenden Programmenausführungen zu unterscheiden und modelliert nicht terminierende Programmausführungen akkurat.

Schließlich wurde die neue Kostenanalyse im Werkzeug CoFloCo implementiert und extensiv evaluiert. Die Evaluierung vergleicht CoFloCo mit anderen, dem aktuellen Stand der Forschung entsprechenden Tools anhand einer Vielzahl von Benchmarks. Diese Benchmarks bestehen aus imperativen und funktionalen Programmen sowie aus Termersetzungssystemen. CoFloCo zeigt durchgehend eine sehr gute Leistung in allen Kategorien und demonstriert damit die Leistungsfähigkeit und Vielseitigkeit der entwickelten Kostenanalyse.

Contents

1 Introduction 1
- 1.1 Applications . . . 2
- 1.2 State of the Art . . . 3
  - 1.2.1 Cost Relations . . . 4
  - 1.2.2 Cost Relation Extraction . . . 6
  - 1.2.3 Cost Relation Solving . . . 9
  - 1.2.4 Limitations of Existing Approaches . . . 13
- 1.3 Contributions . . . 16
  - 1.3.1 Overview of the Publications . . . 17
  - 1.3.2 Structure of the Dissertation . . . 18

2 Informal Account 21
- 2.1 Preprocessing . . . 21
- 2.2 Cost Relation Control-flow Refinement . . . 21
  - 2.2.1 Invariants . . . 22
  - 2.2.2 Refinement Propagation . . . 22
- 2.3 Computing Bounds with Cost Structures . . . 23
  - 2.3.1 Cost Structures . . . 23
  - 2.3.2 Bound Computation . . . 24
  - 2.3.3 Loop with Reset . . . 25
  - 2.3.4 Amortized cost example . . . 25

3 Technical Background 27
- 3.1 Basic Definitions . . . 27
- 3.2 Cost Relations . . . 27
- 3.3 Semantics . . . 28
- 3.4 Failed Evaluations vs Runtime Failure . . . 31
- 3.5 Costs . . . 32
- 3.6 Cost Preserving Transformations . . . 33

4 Preprocessing 35
- 4.1 Call-graphs, Strongly Connected Components and Feedback Sets . . . 35
- 4.2 Unfolding Cost Relations . . . 36
- 4.3 Dealing with Non-unitary Feedback Sets . . . 37
- 4.4 Simplifying Transformations . . . 38
- 4.5 Algorithm . . . 39
- 4.6 Proofs . . . 40
  - 4.6.1 Proof of Theorem 4.3 . . . 40
  - 4.6.2 Proof of Theorem 4.7 . . . 45
  - 4.6.3 Proof of Theorem 4.12 . . . 47
  - 4.6.4 Proof of Theorem 4.14 . . . 47
  - 4.6.5 Proof of Theorem 4.16 . . . 48
  - 4.6.6 Proof of Theorem 4.18 . . . 49

5 Refinement **51**
5.1 Partially Refined Cost Equations . . . 51
5.2 Control-flow Refinement of a Cost Relation . . . 52
5.2.1 Chain Evaluations . . . 53
5.2.2 Discarding Divergent Phases . . . 55
5.2.3 Refined Cost Equations . . . 55
5.2.4 Refined Chains . . . 58
5.3 Invariants . . . 62
5.3.1 Chain Summaries . . . 62
5.3.2 Calling Contexts . . . 64
5.3.3 Discarding Unfeasible Chains . . . 65
5.3.4 Strengthening Cost Equations . . . 65
5.4 Refinement Propagation . . . 66
5.5 Extension to Multiple Recursion . . . 68
5.5.1 Partially Refined Cost Equations . . . 68
5.5.2 Control-flow Refinement of a Cost Relation . . . 68
5.5.3 Invariants . . . 73

6 Bound Computation **75**
6.1 Infinite Evaluations of Cumulative *CRS* . . . 77
6.2 Cost Definitions and Cost Composition . . . 78
6.3 Cost Structures . . . 79
6.4 Cost Equations . . . 82
6.5 Phases . . . 84
6.5.1 Transforming Non-final Constraints . . . 85
6.5.2 Transforming Final Constraints . . . 87
6.5.3 Example of Phase Cost Structure Inference . . . 95
6.6 Chains . . . 97
6.7 Chains with Multiple Recursion . . . 99
6.8 Solving Cost Structures . . . 104
6.9 Piece-Wise Symbolic Bounds . . . 105
6.10 Proofs . . . 107
6.10.1 Theorem 6.8: Cost Composition . . . 107
6.10.2 Strategy Proofs Preliminaries . . . 110
6.10.3 Theorem 6.27: Inductive Sum Strategy . . . 110
6.10.4 Theorem 6.31: Inductive Strategy with Resets . . . 112
6.10.5 Theorems 6.33 and 6.34: Max-Min Strategy . . . 114
6.10.6 Theorem 6.37: Triangular Sum Strategy . . . 116
6.10.7 Theorem 6.43: Inductive Sum Strategy for Multiple Chains . . . 117
6.10.8 Theorem 6.44: Inductive Strategy with Resets for Multiple Recursion . . . 120

7 Evaluation **123**
7.1 Implementation . . . 123
7.2 Experiments on Imperative programs . . . 124
7.2.1 Examples from the Literature . . . 124
7.2.2 Loopus's Real World Experimental Evaluation . . . 126
7.2.3 Loopus's Challenging Loop Patterns Evaluation . . . 128
7.3 Evaluation on Functional Programs . . . 128
7.4 Evaluation on Term-rewrite Systems . . . 130
7.5 Limitations of the Evaluations . . . 131

8 Related Work 133
8.1 Competing Approaches . . . 133
8.1.1 Recurrence relations . . . 133
8.1.2 SPEED . . . 134
8.1.3 Loopus . . . 136
8.1.4 KoAT . . . 136
8.1.5 Rank . . . 137
8.1.6 RAML . . . 138
8.1.7 Lower Bounds . . . 138
8.2 Related Research Areas . . . 139
8.2.1 Term Rewriting . . . 139
8.2.2 COSTA and SACO . . . 139
8.2.3 WCET . . . 140
8.2.4 Cost Bounds Verification . . . 140
8.2.5 Termination and Ranking Functions . . . 141
8.2.6 Verification of Programs with Constrained Horn Clauses . . . 141

9 Conclusion 143

10 Limitations and Future Work 145
10.1 Logarithmic and Exponential Bounds . . . 145
10.2 Non-linear and Non-local Size Change . . . 145
10.3 Cost Structures with Binomial Coefficients . . . 146
10.4 Scalability . . . 147
10.5 Other Challenges: Pointers, Partial Failures and Contracts . . . 148

Bibliography 149

A CoFloCo Implementation Details 159
A.1 CoFloCo Quick Reference . . . 159
A.2 Refinement . . . 159
A.3 Cost Structure Maintenance and Simplification . . . 160

List of Figures

1.1 Program 1 and two different cost models . . . 2
1.2 Cost relations of Program 1 . . . 5
1.3 Program 2: Nested loop . . . 5
1.4 Evaluation tree of Program 2 . . . 6
1.5 Program 3: ABS Example with lists . . . 7
1.6 Rule-based Representation of Program 3 . . . 8
1.7 Abstract compilation of the while loop . . . 8
1.8 Cost relations of Program 3 . . . 9
1.9 Merged evaluation tree of for_2 Program 2 . . . 10
1.10 Program 4: Multiple recursion . . . 11
1.11 Program 5: Example with multiple phases . . . 13
1.12 Program 6: Example with two sequential phases . . . 13
1.13 Program 7: Example with resets . . . 14
1.14 Program 8: Example with multiple bounds . . . 15
1.15 Cost relation based cost analysis: Old versus new . . . 16

2.1 Cost relations of Program 3 with the output variable *so* . . . 21
2.2 Cost Equations, call-graph and chains of Program 6 . . . 22
2.3 Cost structure of phase $(2 \vee 3)^+$ in Program 7 . . . 25
2.4 Cost structures of Program 3 . . . 26

3.1 Evaluation of Program 3 . . . 29
3.2 Program 9: Possibly non-terminating program with non-cumulative cost . . . 30
3.3 Finite and infinite evaluation of Program 9 . . . 30
3.4 Program 10: Different interpretations of program assertions . . . 31

4.1 Cost relations of Program 3 with output variables . . . 36

5.1 Program 11: Example with different kinds of phases . . . 53
5.2 Program 12: Example where strengthening is important . . . 66
5.3 Extension of Program 8 . . . 67
5.4 Cost Relations of Program 4 . . . 69

6.1 Program 13: Running example for bound computation . . . 76
6.2 Evaluation of Program 13 . . . 77
6.3 Some cost structures of Program 13 . . . 80
6.4 Program 14: Example where multiple candidates are important . . . 91
6.5 Refined cost relations of Program 2 and its lower bound . . . 94
6.6 Program 15: Example with complex phase . . . 95
6.7 Program 16: Example with non-linear size relation . . . 98

7.1 Analysis time histogram of real world experimental evaluation . . . 127
7.2 Analysis time histogram of term-rewrite systems evaluation . . . 131

8.1 Program 17: Challenging example for KoAT's bottom-up approach . . . 137

10.1 Refined cost relations of Program 16: Modular representation 145
10.2 Program 18: Example that requires non-linear non-local size relations 146
10.3 Program 19: Motivating example for binomial coefficients 147

List of Tables

6.1 CE Classification conditions for strategies 89
6.2 Cost structure computation of phase $(2 \vee 3 \vee 4 \vee 5)^+$ in Program 15. 96
6.3 Classification conditions for Inductive Sum Strategy in multiple chains 100
6.4 Classification conditions for Inductive Sum Strategy with Resets in multiple chains 101
6.5 Classification conditions for Max-Min strategy in multiple chains 103

7.1 Upper bounds of examples from the literature 125
7.2 Lower bounds of examples from the literature 126
7.3 Replication of real world experimental evaluation of [SZV17] 126
7.4 Replication of challenging loop patterns experimental evaluation of [SZV17] 128
7.5 Evaluation on Functional Programs benchmark 129
7.6 Evaluation on Term-rewrite Systems benchmark 131

1 Introduction

One of the most important properties of computer programs is their time complexity. That is, the amount of time or processing power that they need to be executed. A functionally correct program is of little use if it cannot be executed in a reasonable time. Time complexity is just a specific instance of a more general property that we call the cost of the program. The cost of a program is the amount of resources needed to execute it, where resources can be time, CPU cycles, memory, etc. In general, given two programs that perform a certain task, we prefer the one that has a lower cost, that is, the one that is more efficient.

The cost of programs and algorithms is a well studied problem and it is part of most computer science curricula. However, manual cost analysis is complex, error-prone, and thus not applicable in practice for large scale programs. The focus of this work is to develop techniques for automatic static cost analysis (also known as resource analysis).

In order to formulate the problem in a generic form, the cost of a program is defined with respect to a *cost model*. A cost model determines the resource being measured and the cost of each instruction of the program (or each program location) with respect to the chosen resource. A positive cost represents a resource being consumed and a negative cost a resource being released (or produced). In this manner, the cost of a program can be measured with respect to different resources (e.g. execution steps, memory or bandwidth) by applying different cost models.

Cost models can be more or less realistic depending on our needs. For example, if we are only interested in the asymptotic complexity, it is often enough to count the number of loop iterations (or recursive calls) of a program. In contrast, if we want to obtain precise estimates of the execution times in a given runtime environment, we can apply a more realistic cost model that assigns different costs to different instructions depending on the amount of time (or any other resource) that they require to be executed. Obtaining cost models that can realistically represent the behavior of modern computer architectures and garbage collection is a challenging problem but is not the topic of this work. Here we assume the cost model to be given. In the examples, we use simple cost models to ease the presentation. There has been some recent work in this aspect where precise cost models are learned based on profiling of some sample programs [DH17].

Example 1.1. For instance, consider the C Program 1 in Figure 1.1. This program checks whether an integer value `val` is contained in the array `l` of length `size` (we assume `size` is a positive number). In Figure 1.1, two cost models are considered. Cost model 1 assigns cost 1 to the back-edge of the loop and thus counts the number of iterations of the loop. Cost model 2 assigns different costs to each of the instructions. It assigns cost 1 to assignments, cost 2 to integer comparisons and cost 3 to return statements.

In general the cost of a program (given a cost model) depends on its input parameters. The input parameters can be complex so in order to obtain a cost expression, they are usually abstracted with respect to some notion of size. This, together with the fact that programs can be intrinsically non-deterministic, prevents us from obtaining a precise function that maps the input parameters to the cost of the program. Instead, static cost analysis attempts to obtain functions that represent upper or lower bounds of the cost of the program.

Definition 1.2 (Static Cost Analysis)**.** Given a cost model and a program, a static cost analysis infers upper or lower bounds on the cost of the program in terms of (the size of) its input parameters.

Example 1.3. The cost of Program 1 depends on the specific contents of array `l`, in particular on whether `val` is contained in array `l` and in which position. The worst case (upper bound) occurs when `val` is

Program 1	Line	Cost model 1	Cost model 2
1 `bool search(int *l,int size,int val){`			
2 `int i=0;`	2	$\to 0$	$\to 1$
3 `bool fnd=false;`	3	$\to 0$	$\to 1$
4 `while(i<size && !fnd){`	4	$\to 0$	$\to 2$
5 `if(l[i]==val)`	5	$\to 0$	$\to 2$
6 `fnd=true;`	6	$\to 0$	$\to 1$
7 `++i;`	7	$\to 0$	$\to 1$
8 `}`	8	$\to 1$	$\to 0$
9 `return fnd;`	9	$\to 0$	$\to 3$
10 `}`			

Figure 1.1.: Program 1 and two different cost models

not in the array. In that case, the loop is executed `size` times so the cost upper bound using the cost model 1 is `size`. In the best case (for a non-empty array), `val` is in the first element of the array and the back-edge is taken only once. The lower bound is then 1 for cost model 1.

If cost model 2 is considered, the upper bound occurs when `val` is in the last element of the array (in this case an additional assignment to `fnd` in Line 6 is executed)

$$1+1+2\cdot(\texttt{size}+1)+(2+1)\cdot\texttt{size}+1+3=5\cdot\texttt{size}+8$$

and the lower bound (for a non-empty array) corresponds to the case where the loop body is executed once $1+1+2+2+1+1+2+3=13$. Note that in that case Line 4 is reached twice. If `size` can be 0, the lower bound corresponds to the case where the loop is never entered $1+1+2+3=7$.

Note that this is very different (and complementary) from profiling. With profiling one can obtain actual execution costs for specific inputs whereas static cost analysis provides bounds that are valid for all possible inputs. Analyzing programs statically has additional advantages. Static analysis does not require a complete program, it can also be applied to specific modules or procedures. Therefore, the analysis can be applied in the early stages of software development when there is not a working prototype yet. Besides, a function in terms of the input parameters can provide useful information not only on the cost of the program, but also on which input parameters are more relevant to the cost.

There is a related research area, usually referred to as WCET (Worst Case Execution Time) analysis, that is mainly orthogonal to the work on cost analysis [WEE+08]. WCET analysis attempts to obtain precise time upper bounds of low level programs in a given architecture. A great effort is directed to model aspects of modern architectures such as multilevel cache memories, segmented processors, etc. However, these analyses usually assume that loop bounds are given as an input. The cost analysis developed here can be used to infer such loop bounds.

1.1 Applications

Some applications of static cost analysis are:

Feedback to Developers

During the software development, programmers can apply cost analysis tools to functions or procedures of interest and obtain a measure of their cost behavior. This feedback can help programmers to spot performance bugs in the code and to identify bottlenecks. This allows them to detect problems early in the development. This application is also one of the most realistic and immediate for two reasons: First, this kind of feedback is useful for analyzing small pieces of code such as specific procedures or

modules where the scalability of the analysis is not such a big issue. Second, simple and relatively machine independent cost models can be useful for developers even if they do not model the details of the computer architecture precisely.

Library Documentation

To make good use of a library, a programmer must understand its behavior. This includes not only functional properties but also non-functional properties such as time and memory cost of its different procedures. Static cost analysis can be used to enrich the library documentation with automatically inferred cost bounds so users can take this information into account. In this case, simple cost models can also provide useful results. The scalability of the analysis can become an issue as libraries can be huge. However, this process does not need to be done in real time, it can be performed in the background and parallelized to some extent.

Cloud Architecture Design

Nowadays, programs are often executed in distributed environments, that is, in the cloud. As a result, additional resources become relevant such as the used network bandwidth and the amount of machines or processors involved in the computation. The inclusion of additional resources makes the cost of the programs in terms of different resources even more relevant. It is often the case that different algorithms present various trade-offs with respect to their cost. A distributed algorithm might need less time to execute than its non-distributed counterpart. In turn, the distributed algorithm might need more machines and cause greater stress over the communication network. Cost analysis can provide specific information about these trade-offs early in the development phase.

Certified Software

Finally, static cost analysis can also play an important role in software certification. There are multiple software applications that need to comply with very strict time and memory constraints. A clear example of such applications is the software in embedded systems. This is the field of application of WCET analysis. Cost analysis, as already mentioned, can be used to infer loop bounds that are then provided as an input to WCET analysis tools.

1.2 State of the Art

Automatic static cost analysis of programs is a very active field of research. Since the seminal work of [Weg75], many different approaches and techniques have been published that focus on different programming paradigms. The cost of a program is an undecidable property, so there cannot be a general and precise algorithm that works for every program. The quest has been to expand the class of programs that can be automatically analyzed and to increase the precision of the analysis, while at the same time retaining scalability.

Most early works are based on extracting and solving recurrence relations that represent the cost of the program [Weg75, DLH90, DL93, DLHL94, DLHL97, Ben01, Gro01]. However, both extracting and solving recurrence relations is challenging. Programs, in particular imperative programs, have features that are hard to model directly with recurrence relations. Imperative programs often have *non-monotonic* cost that depends on *multiple variables*. They can be *non-deterministic* and have *complex control flow*. All of that makes them hard to model with recurrence relations.

Some more recent analyses are based on type systems, mainly for functional programs, based on the idea of sized types [Vas08, VH04] or based on the potential method for amortized cost analysis [HH10a, HDW17]. There are also approaches for imperative programs based on abstract interpretation and counter instrumentation [GMC09], control-flow refinement [GJK09], proof rules [GZ10], and the inference of ranking functions [ADFG10, AAGP11, ZGSV11, BEF+16]. Finally, some approaches

combine several methods, for example the work of [SLH14] relies on sized types, abstract interpretation and recurrence relations.

There are many cost analyses [AAGP11, AGM13, ABAG13, ADFG10, ZGSV11, SZV14, SZV17, BEF+16] that follow a general approach divided into two phases:

1. The first phase takes a program and a cost model and generates an integer abstract representation of the program.

2. The second phase analyzes the integer abstract representation and produces upper (or lower) bounds of the program cost.

The main advantage of this approach is that the integer abstract representation is much simpler than the original source code (or binary code) and it is language independent. An algorithm to obtain bounds from the abstract representation can be used to obtain bounds of programs written in different languages and with respect to different cost models. This is exemplified in the evaluation (Chapter 7) in which the research prototype is evaluated against imperative programs, functional programs and even term rewrite systems.

As already mentioned, early works extract and solve recurrence relations. Zuleger et al. [ZGSV11] abstract programs to size-change graphs, Sinn et al. [SZV17] to difference constraint programs, [BEF+16] and [ADFG10] consider integer transition systems (ITS) and the works [AAGP11, AGM13, ABAG13] are based on extracting and solving cost relations. The present work adopts the latter approach.

The following subsections provide an introduction to cost relations and the existing techniques to extract them and solve them. Chapter 8 contains a more detailed account of other approaches to cost analysis and their comparison to this work.

1.2.1 Cost Relations

A *cost relation system* is an abstract representation of the cost of programs used for cost analysis. It is composed of set of cost relations (CR) defined with recursive equations. Each of these equations, denoted cost equations (CE), has the following format:[1]

$$c\colon C(\boldsymbol{x}) = \varphi_0, b_1, \varphi_2, \cdots, b_n, \varphi_n$$

where c is a unique identifier; $C(\boldsymbol{x})$ is the head and $\varphi_0, b_1, \varphi_2, \cdots, b_n, \varphi_n$ is the body. The terms b_i are costs or references to other cost relations and φ_i are (possibly empty) constraint sets. A cost equation states that the cost of CR C with variables $\boldsymbol{x}$ is defined by the costs of $b_1, \cdots, b_n$ under the constraints $\varphi_0, \cdots, \varphi_n$. The costs and the constraints in the CE's body are accumulated sequentially from left to right. The constraint sets of a CE capture its applicability conditions and also express the (possibly non-deterministic) relations among different variables. Cost relation systems can be seen as constraint logic programs with cost annotations. Under this view, a cost relation corresponds to a predicate and a cost equation corresponds to a clause.

Each fragment of code, typically a function or a loop, is translated into a cost relation which is a set of cost equations. Each cost equation defines the cost of a possible behavior of the function, that is, the cost of an execution path of the function.

Example 1.4. Figure 1.2 contains the cost relation system of Program 1 using cost model 2 (defined in Figure 1.1). The cost relation system contains 2 cost relations *search* and *while* and a total of 5 numbered cost equations. In this abstraction booleans (`fnd`) are represented as integers in the usual way and array 1 has been abstracted to its length.

[1] This format is later slightly extended as part of the contributions of the thesis.

Cost relations of Program 1

1: $search(l, size, val) = \{i = 0, fnd = 0\}$, 2, $while(l, size, val, i, fnd)$, 3

2: $while(l, size, val, i, fnd) = \{i \geq size\}$, 2
3: $while(l, size, val, i, fnd) = \{i < size, fnd = 1\}$, 2
4: $while(l, size, val, i, fnd) = \{i < size, fnd = 0, fnd' = 1, i' = i + 1\}$, 6, $while(l, size, val, i', fnd')$
5: $while(l, size, val, i, fnd) = \{i < size, fnd = 0, i' = i + 1\}$, 5, $while(l, size, val, i', fnd)$

Figure 1.2.: Cost relations of Program 1

The loop of Program 1 is defined recursively. Each CE corresponds to a loop path. CEs 2 and 3 represent the two possibilities to exit the loop and CEs 4 and 5 represent the cases where the loop body is executed.

The constraint sets define the applicability conditions of each CE, for example, the constraints $i < size, fnd = 0$ in CE 4 indicate that the loop body can only be executed if the counter i has not reached *size* and the element has not been found yet. Additionally, they define the behavior resulting from executing the path, for example, the constraints $fnd' = 1, i' = i + 1$ in CE 4 encode the assignment to `fnd` and the increment of `i`. Note that as a result of the abstraction of `l`, the condition `l[i]==val` does not generate any constraint. As a result, the conditions of CEs 4 and 5 are not mutually exclusive.

A cost relation system can be evaluated with respect to some input values into evaluation trees where each node of the tree corresponds to the application of a cost equation and its children are the evaluations of the calls (or cost annotations) that appear on the CE body. The input values of the children satisfy the constraints of the cost equation. Each evaluation node has a cost associated and the cost of an evaluation is the sum of the costs of its nodes. If at some point of the evaluation the constraints of a CE are not satisfied, the evaluation fails. As we shall later see (Section 3.3), only evaluations that do not fail are taken into account.

Example 1.5. Program 2 in Figure 1.3 contains a function with two nested loops which are encoded into two cost relations for_2 and for_3. The function `tick` is used to specify the cost model. A call to `tick(n)` consumes `n` resource units.

Figure 1.4 contains an evaluation of Program 2 with input value $n = 3$. Each node is represented with the cost equation number that has been applied, the cost relation symbol to which it belongs and the values of its variables. The nodes that correspond to costs simply have the amount of resources consumed (or released). The cost of the evaluation is 6.

Cost relations have some advantages over other abstract representations. They support recursive programs naturally. In fact, loops are modeled as recursive definitions and that allows us to analyze loops and recursive functions uniformly. In contrast, other abstract program representations such as difference constraint programs [SZV17] and integer rewrite systems (ITS) do not support recursion naturally or need to be extended [BEF+16]. More importantly, cost relations have a modular structure.

Program 2

```
1 void tri(int n){
2 for(x=0;x<n;x++)
3   for(y=x;y<n;y++)
4     tick(1);
5 }
```

Cost relations

1: $tri(n) = \{x = 0\}, for_2(x, n)$
2: $for_2(x, n) = \{y = x, x < n, x' = x + 1\}, for_3(y, n), for_2(x', n)$
3: $for_2(x, n) = \{x \geq n\}$
4: $for_3(y, n) = \{y < n, y' = y + 1\}$, 1, $for_3(y', n)$
5: $for_3(y, n) = \{y \geq n\}$

Figure 1.3.: Program 2: Nested loop

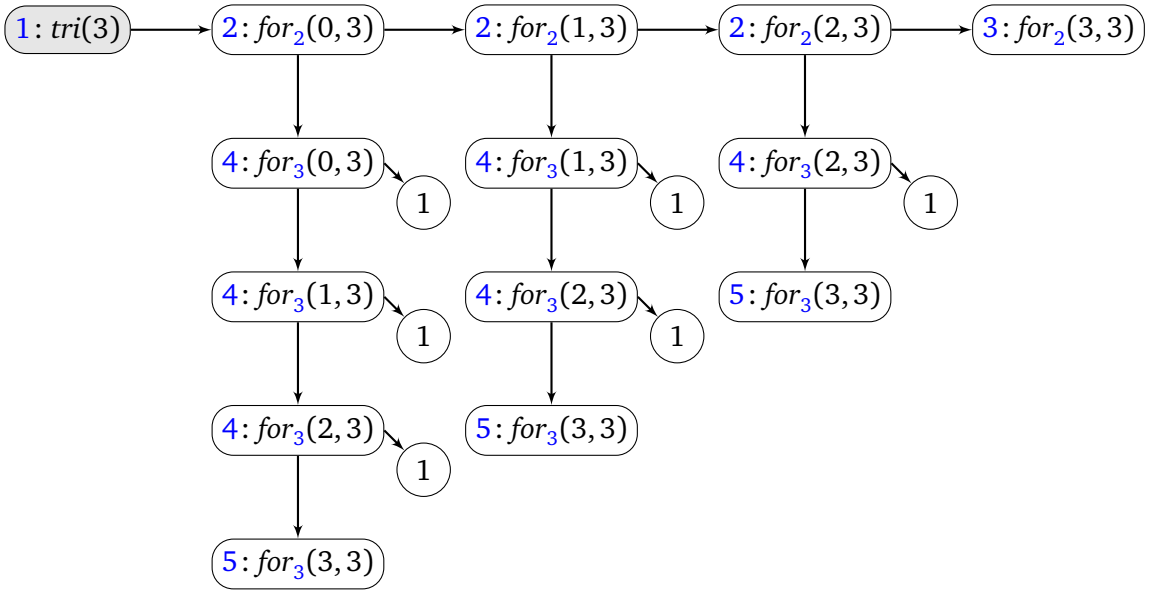


Figure 1.4.: Evaluation tree of Program 2

Each loop or function is abstracted into a separate cost relation. This enables a compositional approach to compute the cost of a program by combining the costs of its parts.

Example 1.6. We can compute the cost of Program 2 incrementally. First, we compute the cost of the inner loop (CR for_3); then we substitute the reference $for_3(y, n)$ in CE 2 by its cost; and finally we compute the cost of the outer loop (CR for_2).

From the two phases of the analysis, cost relation extraction and solving, this work focuses on the second one, that is, the inference of upper and lower bound for programs expressed as cost relations. However, as we shall see later, some small changes are done on the overall approach. The next two subsections contain a description of some of the previously existing methods used for each of the phases.

1.2.2 Cost Relation Extraction

Cost relation extraction has been studied extensively. Cost relations can be generated for programs written in different languages (Java bytecode [AAG+08, AAG+12], ABS [AAG+11, GLL15], LLVM IR [GGP+15]); different programming paradigms such as imperative and functional or mixed (such as ABS [JHS+11]); and to measure different resources such as time, memory [AGGZ13], information transmitted over a network [ACMMRD14], etc.

This section describes in a high-level fashion the main steps in the cost relation extraction method presented in [AAG+11] and implemented in the SACO tool [AAF+14]. This method is very similar to the one presented in [AAG+12] (implemented in the COSTA tool [AAG+08]) but the latter extracts cost relations from Java Bytecode which makes the process less intuitive. In contrast, the work [AAG+11] extracts cost relations from programs written in ABS which is a high level object-oriented programming language designed for modeling distributed applications. The language has an imperative part with Java-like syntax and a first order functional sub-language with algebraic data types. Throughout this thesis, the considered example programs are either C programs or ABS programs if they contain functional elements.

Consider Program 3 in Figure 1.5. It contains two methods `amortized` and `popSome`. Method `amortized` contains a loop that iterates over the list `l` and adds its elements to another list `s` until `l` is empty (`Nil`). Besides, in each iteration it checks if the shared global variable `consume` is set to `true`. In

Program 3

```
Unit amortized(List<A> l,List<A> s){
 while(l!=Nil){
  s=Cons(head(l),s);
  l=tail(l);
  if(consume)
    s=popSome(s);
 }
}
```

```
List<A> popSome(List<A> s){
 if(!consume || s==Nil)
  return s;
 else
  popSome(tail(s));
}
```

Figure 1.5.: Program 3: ABS Example with lists

that case it makes a call to `popSome` and stores the result in `s`. `popSome` is a recursive method that iterates over the elements of `s` as long as the list is not empty or the shared variable `consume` is set to `false`. The constructors for lists are `Cons` and `Nil` and `head` and `tail` are primitive operations that return the head and the tail of a list respectively. [2]

The cost relation extraction can be divided in three steps as follows.

Rule-based representation

First, the source code is parsed and a rule-based representation (RBR) is generated. In this rule based representation each procedure, loop and conditional statement is abstracted into a set of rules that define its behavior. Each rule has the format $m(\boldsymbol{x} : \boldsymbol{y}) \leftarrow g, b_1, \ldots, b_n$. There, $m(\boldsymbol{x} : \boldsymbol{y})$ is the head of a rule with name m, input parameters $\boldsymbol{x}$ and output parameters $\boldsymbol{y}$. The body of the rule is $g, b_1, \ldots, b_n$ where g is a guard that defines the applicability conditions of the rule and $b_1, \ldots, b_n$ is a sequence of statements. The statements can be either variable assignments or calls to other rules. The RBR can be generated syntactically from the abstract syntax tree of the program.

Figure 1.6 contains the RBR of Program 3. Note how each procedure, loop and if statement has been translated into a set of rules. For instance, the rules $wh(l, s : l, s)$ correspond to the `while` loop. The first rule corresponds to the exit condition and the second rule corresponds to the body of the loop. Note that the conditional in function *popSome* is abstracted to three rules due to the short circuit semantics of the operator ||. In addition, the shared variable `consume` has not been included in the list of input and output parameters. Because `consume` is a shared variable, it is assumed that its value can change at any point and it is thus undefined.

Size abstraction and abstract compilation

In the next step, data structures are mapped to integer values according to some selected size abstraction. Size abstractions typically abstract data structures to the number of constructors in the data structure or the maximum depth of the data structure (path-depth abstraction [AAG+12]). More sophisticated size measures can be implemented using sized-types [AGG13]. In this example, lists l and s are mapped onto their length. Then, each rule in the rule-based representation is transformed into a cost equation. First, each rule is transformed into a single static assignment (SSA) representation. Then, a set of linear constraints is generated for each guard and each statement in the rule. Finally, the cost of the rule is defined by applying a given cost model to each of its statements.

Consider the rules of the while loop (Figure 1.6). Figure 1.7 contains the SSA versions of the rules (on the left) and the mapping from each guard/statement to a linear constraint set and to a cost. For instance, the statement $s' = \text{Cons}(tmp, s)$ induces the constraint $s' = s + 1$ that indicates that the length of s is incremented in one unit. The selected cost model counts the number of iterations and recursive

[2] The syntax of Program 3 has been slightly simplified. The ABS language distinguishes between synchronous and asynchronous calls so the calls to `popSome` should actually be "`await this!popSome(s)`" and "`await this!popSome(tail(s))`", respectively.

$$
\begin{array}{lcl}
amortized(l,s:) & \leftarrow & True, \\
 & & wh(l,s:l,s) \\
wh(l,s:l,s) & \leftarrow & l == \text{Nil} \\
wh(l,s:l,s) & \leftarrow & l! = \text{Nil}, \\
 & & tmp = head(l), \\
 & & s = \text{Cons}(tmp,s), \\
 & & l = tail(l), \\
 & & if_1(l,s:l,s), \\
 & & wh(l,s:l,s) \\
if_1(l,s:l,s) & \leftarrow & consume, \\
 & & popSome(s:s), \\
if_1(l,s:l,s) & \leftarrow & !consume
\end{array}
\qquad
\begin{array}{lcl}
popSome(s:r) & \leftarrow & True, \\
 & & if_2(s:r) \\
if_2(s:r) & \leftarrow & !consume, \\
 & & r = s \\
if_2(s:r) & \leftarrow & consume \;\&\&\; s == \text{Nil}, \\
 & & r = s \\
if_2(s:r) & \leftarrow & consume \;\&\&\; s! = \text{Nil}, \\
 & & tmp = tail(s), \\
 & & popSome(tmp:r)
\end{array}
$$

Figure 1.6.: Rule-based Representation of Program 3

SSA Rule			Constraints	Cost
$wh(l,s:l,s)$	$\leftarrow$	$l == \text{Nil}$	$\{l=0\}$	0
$wh(l,s:l'^3,s'^3)$	$\leftarrow$	$l! = \text{Nil}$,	$\{l>0\}$	0
		$tmp = head(l)$,		0
		$s' = \text{Cons}(tmp,s)$,	$\{s'=s+1\}$	0
		$l' = tail(l)$,	$\{l'=l-1\}$	0
		$if_1(l',s':l'^2,s'^2)$,		$if_1(l',s':l'^2,s'^2)$
		$wh(l'^2,s'^2:l'^3,s'^3)$		$1, wh(l'^2,s'^2:l'^3,s'^3)$

Figure 1.7.: Abstract compilation of the while loop

calls of the program so it assigns one cost unit to each call to *wh* and 0 to any other statement. The resulting cost relation for the while loop is:

$$
\begin{array}{lcl}
wh(l,s:l,s) & = & \{l=0\} \\
wh(l,s:l'^3,s'^3) & = & \{l>0, s'=s+1, l'=l-1\},\ 1,\ if_1(l',s':l'^2,s'^2),\ wh(l'^2,s'^2:l'^3,s'^3)
\end{array}
$$

Input-output size analysis

In the usual definition of cost relations, these are only defined in terms of the input variables so the output variables have to be removed. However, there can be calls to rules that depend on the output of other rules. For instance, in the cost relations of the while loop, the recursive call to *wh* is defined in terms of l'^2 and s'^2 which are the output variables of the call $if_1(l',s':l'^2,s'^2)$. In order to deal with this issue, a global size analysis is performed that infers input-output size relations [BK96] for each cost relation. Then, the cost equations are enriched with the input-output size relationships of the called cost relations. The input-output size relation of the call $if_1(l',s':l'^2,s'^2)$ is the constraint set $\{l'^2=l', s'^2 \leq s'\}$ so that constraint set is added to cost equation after the call to if_1. The final cost equations of the while loop are:

$$
\begin{array}{lcl}
wh(l,s) & = & \{l=0\} \\
wh(l,s) & = & \{l>0, s'=s+1, l'=l-1\},\ 1,\ if_1(l',s'),\ \{l'^2=l', s'^2 \leq s'\},\ wh(l'^2,s'^2)
\end{array}
$$

Figure 1.8 contains the complete cost relations of Program 3. Because the variable *consume* is undefined, the conditions *consume* and !*consume* are abstracted to an empty constraint set and the cost

Cost relations of Program 3

$amortized(l,s) = wh(l,s)$

$wh(l,s) = \{l = 0\}$
$wh(l,s) = \{l > 0, s' = s+1, l' = l-1\},\ 1,\ if_1(l',s'),\ \{l'^2 = l', s'^2 \leq s'\},\ wh(l'^2, s'^2)$

$if_1(l,s) = popSome(s)$
$if_1(l,s) = 0$

$popSome(s) = if_2(s)$

$if_2(s) = 0$
$if_2(s) = \{s > 0, tmp = s-1\},\ 1,\ popSome(tmp)$

Figure 1.8.: Cost relations of Program 3

relations if_1 and if_2 become non-deterministic. Each rule in the RBR representation gives rise to one cost equation with one exception. The first and second rules of if_2 give rise to the cost equations $if_2(s) = \{\},\ 0$ and $if_2(s) = \{s > 0\},\ 0$ respectively. However, the second equation is simply a special case of the first one (it is subsumed by the first one) and can be discarded.

As we shall see later, the last point of the cost relation extraction, that is, the elimination of output variables and the inference of input-output relations, presents important limitations. The analysis presented here skips this step and analyzes cost relations that include the output variables instead.

1.2.3 Cost Relation Solving

There are several approaches to solve cost relations [AAGP11, AGM13, ABAG13] implemented in the tool PUBS [3]. All of these approaches compute bounds of cost relations incrementally considering one cost relation at a time. Once the bound of a cost relation C has been computed, the bound is used to substitute any calls to C that appear in other cost relations.[4] The approaches differ on how they compute bounds of each cost relation. Below, there is an informal description of different methods to compute bounds of cost relations.

Node-Count method

The first method, presented in [AAGP11], over-approximates the cost of an arbitrary evaluation tree by considering the number of nodes in the tree and the maximum cost per evaluation node. This method considers evaluation trees where all the nodes that do not belong to the cost relation being analyzed have been merged together and their cost has been assigned to the first ancestor node that belongs to the cost relation.

Example 1.7. Figure 1.9 contains the evaluation of for_2 from Figure 1.4 where all the nodes that do not belong to for_2 have been merged to the corresponding node of for_2. The nodes of for_2 are now annotated with costs.

Given a cost relation C. Let $\#i$ and $\#l$ be upper bounds on the number of internal and leaf nodes of an evaluation tree of C. Let $\widehat{ei}$ and $\widehat{el}$ be upper bounds on the cost of any internal and leaf node in an evaluation tree of C. The following expression is an upper bound of C:

$$\#i \cdot \widehat{ei} + \#l \cdot \widehat{el}$$

[3] http://costa.ls.fi.upm.es/pubs/solver.php

[4] If there are mutually recursive cost relations, they are merged into one using unfolding (details in Chapter. 4).

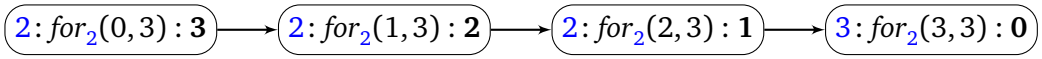


Figure 1.9.: Merged evaluation tree of for_2 Program 2

Example 1.8. In the evaluation of Figure 1.9, there are three internal nodes and one leaf node. The costs of the internal nodes are 3, 2 and 1 and the cost of the leaf node is 0. The expression $3 \cdot 3 + 1 \cdot 0 = 9$ is a valid upper bound of this particular evaluation.

The number of internal nodes $\#i$ and leaf nodes $\#l$ of a tree can be over-approximated using its depth dp and its maximum branching factor br:

$$\#i \leq \begin{cases} dp & \text{if } br = 1 \\ (br^{dp} - 1)/(br - 1) & \text{if } br \geq 2 \end{cases} \qquad \#l \leq br^{dp}$$

These formulae come from assuming that the evaluation are complete trees. The maximum branching factor br corresponds to the maximum number of recursive calls in a CE of C. Knowing that, it is enough to infer dp, $\widehat{ei}$ and $\widehat{el}$.

1. The approach computes an upper bound on the depth of the evaluation tree dp by inferring a linear ranking function over the recursive CEs [PR04].

2. Conversely, $\widehat{ei}$ and $\widehat{el}$ are approximated using a transitive linear invariant that relates the values of the variables of any evaluation node to the initial values of the variables.

Example 1.9. Let us compute the upper bound of Program 2 using the Node-Count method. First, we compute the cost of the inner loop for_3. The CEs in for_3 have at most one recursive call, thus the branching factor is $br = 1$. The expression $n - y$ is a valid linear ranking function of CE 4 (it is positive and decreases by 1 in each recursive call). Therefore, the maximum depth of an evaluation tree is $dp = \|n - y\|$ where $\|n - y\|$ stands for $\max(n - y, 0)$. The costs of applying CEs 4 and 5 are constant so we have $\widehat{ei} = 1$ and $\widehat{el} = 0$. In conclusion, the cost of for_3 is

$$\#i \cdot \widehat{ei} + \#l \cdot \widehat{el} = \|n - y\| \cdot 1 + 1^{\|n-y\|} \cdot 0 = \|n - y\|$$

We substitute the call to for_3 by its upper bound in the CE 2 of for_2 thus getting:

$$2 : for_2(x, n) = \{y = x, x < n, x' = x + 1\}, \|n - y\|, for_2(x', n)$$
$$3 : for_2(x, n) = \{x \geq n\}$$

Here again we have to compute the maximal cost that the expression $\|n - y\|$ can take in a CE application with respect to the initial variables. This is achieved with relational linear invariants and we conclude that $\widehat{ei} = \|n\|$. The cost of the leaf nodes is $\widehat{el} = 0$. In this case, the expression $n - x$ is a valid ranking function so $dp = \|n - x\|$. Thus, the upper bound of for_2 is

$$\#i \cdot \widehat{ei} + \#l \cdot \widehat{el} = \|n - x\| \cdot \|n\| + 1^{\|n-x\|} \cdot 0 = \|n - x\| \cdot \|n\|$$

The cost relation *tri* simply calls for_2 with $x = 0$ so its upper bound is $\|n\|^2$.

Series method

The paper [AGM13] presents an improvement over the described method. One of the sources of imprecision of the Node-Count method is that it approximates the cost of all evaluation nodes to the

Program 4

```
data Tree= Leaf | Node(Int,Tree,Tree);
def List<Tree> subtrees(Tree t) =
 case t{
  Leaf => Nil;
  Node(x, t1, t2) =>
   Cons(Node(x, t1, t2),append(subtrees(t1),subtrees(t2)));
};
```

Cost relations

1: $subtr(t) = \{t = 0\}$

2: $subtr(t) = \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0\}$, 1, $subtr(t1)$, $\{l1 = t1\}$, $subtr(t2)$, $\{l2 = t2\}$, $append(l1, l2)$, $\{l1 + l2 = l3, l = l3 + 1\}$

Figure 1.10.: Program 4: Multiple recursion

worst case. This is the case of the previous example where the cost per node $\|n-y\|$ is over-approximated to $\|n\|$.

The Series method tries to compute the sum of the cost of the nodes precisely by generating a recurrence relation that models how the cost per node varies during the evaluation. In order to be able to generate a recurrence relation, the cost per node has to change linearly or geometrically. Besides, a bound for the depth of the evaluation tree dp is also needed. The implementation of the approach uses Maxima [5], a computer algebra system, to solve the generated recurrences.

Example 1.10. In case of CR for_2 in Program 2, the cost of the initial node is $c_0 = n - y = n - x$, the cost of consecutive nodes decreases by $\Delta c = 1$ in each iteration and $dp = n - x$ is a valid ranking function. Therefore, the recurrence that over-approximates the cost of for_2 generated by the Series method

$$P(N) = \|c_0 - \Delta c \cdot dp\| + \Delta c \cdot N + P(N-1)$$

corresponds to $P(N) = 1 \cdot N + P(N-1)$. By computing the closed-form of P and substituting N by $\|n-x\|$, we obtain an upper bound $\|n-x\|^2/2 + \|n-x\|/2$ for CR for_2 which is precise.

In cases where the cost per node is a complex expression, the paper presents some techniques to split it and compute the sums of sub-expressions independently. The work [AGM13] also presents some heuristics to approximate the cost per node in the case where a cost relation has several recursive CEs.

This method can also compute lower bounds of cost relations following an approach that is almost symmetric. Instead of using ranking functions to over-approximate the depth of the evaluation tree, it uses counter instrumentation and invariants to under-approximate it and the generated recurrence relations are also under-approximating.

Tree-Sum and Visit-Bound method

Finally, Alonso et.al. [ABAG13] presents an alternative approach, orthogonal to the one of [AGM13], that tackles a class of programs where the previous methods are imprecise. Let us illustrate this kind of programs with an example:

Example 1.11. Consider the ABS Program 4 in Figure 1.10. This program computes a list with all subtrees of a given tree. Without counting the cost of *append*, the cost of Program 4 is linear with respect to the size of the initial t (each node of the tree is visited once). If the cost of *append* is linear on $l1$, the cost of *subtr* is quadratic. However, the methods presented until this point yield an exponential cost. Take the Node-Count method, the branching factor is $br = 2$, the depth of the tree is at most $dp = \|t\|$.

[5] http://maxima.sourceforge.net/

Even if the cost of *append* is not considered ($\widehat{ei} = 1$ and $\widehat{el} = 0$), the resulting upper bound is $2^{\|t\|}$. This is because this method ignores the complementary relation of the two recursive calls in CE 2 $t = 1+t1+t2$ which is essential to guarantee that the cost of *subtr* is not exponential.

The method presented in [ABAG13] uses linear programming and Farkas' lemma to obtain precise sums of the cost of all evaluation nodes in any CR evaluation tree.

Example 1.12. Let us consider the case of Program 4 where the cost of *append* is ignored. The (simplified) recursive cost equation is:

$$2\colon subtr(t) = \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0\},\ 1,\ subtr(t1),\ subtr(t2)$$

The cost of one evaluation node is 1. The Tree-Sum method generates a linear template $q * t + q_0$ and finds an instantiation that satisfies

$$\{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0\} \Rightarrow (q * t + q_0) \geq 1 + (q * t1 + q_0) + (q * t2 + q_0)$$

Such an instantiation, $[q := 1, q_0 := 0]$ yields a linear expression t that represents an upper bound on the sum of the cost of all the evaluation nodes for *subtr*. Intuitively, this method can be seen as an application of the potential method for cost analysis [Tar85]. The expression $(q * t + q_0)$ represents the potential associated to the input variables and the condition requires that such a potential can pay for the cost of the node 1 plus the potential of the recursive calls $(q * t1 + q_0) + (q * t2 + q_0)$.

The Visit-Bound method is used when the Tree-Sum method fails. In many cases, there is not a linear expression that represents the sum of the cost of all evaluation nodes, but it is possible to find an expression that bounds the number of evaluation internal nodes $\#i$ (the number of visits). Therefore, the formula proposed in the Node-Count method can be applied with the improved estimate of $\#i$.[6]

Example 1.13. If we consider Program 4 where the cost of *append* is $\|l1\|$, we can apply the Visit-Bound method. The cost per node is bounded by $\widehat{ei} = 1 + \|t\|$ (this is still obtained with linear invariants), the cost of the leaf nodes is 0, and the expression t bounds the number of internal evaluation nodes $\#i$ (Each visit to CE 2 counts 1 so the expression obtained in the previous example bounds the number of visits). Therefore, the cost upper bound of *subtr* is $\|t\| + \|t\|^2$.

Alonso et.al. [ABAG13] also propose how to split complex cost expressions into simpler ones so they can be solved independently with the methods mentioned above and how to deal with cost relations with more than one recursive cost equation.

This approach is very powerful and serves as a basis for the bound inference technique developed in this dissertation. It always obtains results that are asymptotically better or equal than the approach in [AAGP11] but it is orthogonal to the approach of [AGM13]. For instance, it obtains the same bound as the Node-Count method for Program 2, which is less precise than the bound obtained with the Series method.

Finally, it is worth mentioning the work of Alonso et al.[ABG12]. It presents a method to obtain bounds of *abstract cost rules*. Abstract cost rules are equivalent to the cost relation systems presented here but without removing the output variables in the cost relation extraction process. This work contains two key insights:

1. It contains the realization that including the output variables in the abstract representation can be important to obtain precise amortized bounds.

[6] If there are several recursive CEs the number of visits to a CE with a certain cost does not exactly coincide with the number of internal evaluation nodes $\#i$ but the reasoning is similar.

Program 5	Cost relations
`while(0<i&&i<n){`	
`  if(fwd)`	$1: wh(i,n,fwd) = \{i \leq 0\}$
`   i++;`	$2: wh(i,n,fwd) = \{i \geq n\}$
`  else`	$3: wh(i,n,fwd) = \{0 < i < n, fwd \geq 1, i' = i+1\},\ 1,\ wh(i',n,fwd)$
`   i--;`	$4: wh(i,n,fwd) = \{0 < i < n, fwd = 0, i' = i-1\},\ 1,\ wh(i',n,fwd)$
`}`	

Figure 1.11.: Program 5: Example with multiple phases

Program 6	Cost Relations
`while(i<n){`	
`  if(r>0){`	
`    i=0;`	$1: wh(i,n,r) = \{i \geq n\}$
`    r--;`	$2: wh(i,n,r) = \{i < n,\ r > 0,\ i' = 0,\ r' = r-1\},\ 1,\ wh(i',n,r')$
`  }else{`	$3: wh(i,n,r) = \{i < n,\ r \leq 0,\ i' = i+1\},\ 1,\ wh(i',n,r)$
`    i++;`	
`  }`	
`}`	

Figure 1.12.: Program 6: Example with two sequential phases

2. It also generalizes the notions of cost for non-cumulative resources, that is, resources that can be allocated and deallocated such as memory. For this kind of resources, one can consider net-cost bounds and peak-cost bounds. The net-cost upper bound is an upper bound on the overall balance of allocation and deallocation for a complete execution and the peak-cost upper bound is the maximum amount of resources that are allocated at any point during a (possibly non-terminating) execution.

These concepts are adopted and used in the present work. However, the actual method to obtain bounds presented in [ABG12] relies on quantifier elimination for non-linear formulae and thus it does not scale in practice.

1.2.4 Limitations of Existing Approaches

All mentioned approaches [AAGP11, AGM13, ABAG13] suffer from important limitations that make the analysis fail or obtain bounds that are too imprecise for a wide range of programs. Some of the most important limitations are the following:

Multi-phase loops

Existing methods do not take into account the control-flow of the cost relations encoded in their constraints. This makes these analyses fail or yield imprecise bounds for programs where such control flow is not trivial.

An example of this is Program 5. This program increases `i` until it reaches `n` if `fwd` is true, or it decreases `i` until it reaches 0, otherwise. Regardless of whether `fwd` is true or not, all the iterations of the loop execute the same path (`fwd` does not change). This knowledge is essential to obtain a bound of the loop. If it is assumed that the two paths (CEs 3 and 4) can interleave, it is not possible to find a bound and indeed none of the cost relation based approaches finds a bound for Program 5.

Program 6 is another example of a loop with two phases. CE 2 (corresponding to the "then" path) is always executed (if at all) before CE 3 (corresponding to the "else" path). As a result, even though i can be reset r times (in CE 2), the number of iterations of Program 6 is at most $\|n\| + \|r\|$.

Program 7	Cost Relations

`while(i<n){`
`  if(r>0 && *){`
`  i=0;`
`  r--;`
`  }else{`
`  i++;`
`  }`
`}`
	$1 : wh(i,n,r) = \{i \geq n\}$
	$2 : wh(i,n,r) = \{i < n,\ r > 0,\ i' = 0,\ r' = r-1\},\ 1,\ wh(i',n,r')$
	$3 : wh(i,n,r) = \{i < n,\ i' = i+1\},\ 1,\ wh(i',n,r)$

Figure 1.13.: Program 7: Example with resets

Loops with resets

Program 7 is a variant of Program 6 where Line 2 has been replaced by `if(r>0 && *)` (where the symbol $*$ represents a side-effect free undefined condition). In this case CE 2 and CE 3 can interleave (CE 3 will not contain the constraint $r \leq 0$ any longer). Before i can reach n it can be reset to a value between 0 and n. This reset can happen at most r times.

Both approaches [AAGP11] and [AGM13] rely on obtaining a single linear ranking function that is valid for all the recursive CEs of a CR. This is not possible in loops that contain resets. E.g. there is no linear ranking function for CEs 2 and 3. The work [ABAG13] does not rely on linear ranking functions but fails to obtain a bound for loops with resets for similar reasons. In the example, it tries to find a linear expression that represents the sum of the evaluations of CE 3 and such an expression does not exist. The sum of all the evaluations of CE 3 can be as big as $\|n-i\|+\|n\|\cdot\|r\|$. A valid cost upper bound of Program 7 is $\|n-i\|+\|r\|+\|n\|\cdot\|r\|$.

Amortized cost

Program 3 (Figure 1.5) exemplifies amortized cost. Function `amortized` iterates over the list `l` and adds its elements to another list `s` until `l` is empty (`Nil`). Besides, in each iteration it checks if the shared global variable `consume` is set to `true`. In that case it makes a call to `popSome` and stores the result in `s`. Function `popSome` iterates over the elements of `s` as long as the list is not empty or the shared variable `consume` is set to `false`. It is easy to see that the cost of Program 3 is linear. The method *popSome* visits the elements of `s` and `l` at most once. However, the existing analyses based on cost relations infer a quadratic bound $\|l\|\cdot\|l+s\|$ at best. They infer that *popSome* can have at most s recursive calls, s can be at most $l+s$ (in terms of the initial values of the variables) and *popSome* can be called at most $\|l\|$ times in Program 3.

In fact, it is not possible to infer a precise cost for this example with the cost relations extracted in Figure 1.8. This is because these CRs only consider the input values and approximate the output values with linear input-output relations (See the last point of cost relation extraction Section 1.2.2). As noted by Alonso et al. [ABG12] this is a source of imprecision that prevents us from obtaining amortized costs. Therefore, the approach presented in this thesis considers cost relations that also include the output variables and infer bounds that depend on these variables. For example, the considered CEs for *popSome* and if_2 in Program 3 are:

$$popSome(s : so) = if_2(s : so)$$
$$if_2(s : so) = \{s = so\}$$
$$if_2(s : so) = \{s > 0, tmp = s-1\},\ 1,\ popSome(tmp : so)$$

where *so* represents the return value of the function *popSome*.

Program 8

```
def List<A> take(List<A> l,Int n)=
 if (n<=0) Nil else
 case l {
   | Nil => Nil;
   | Cons(h,t) => Cons(h,take(t,n-1));
 };
```

Cost relations

$1 : take(l, n : ret) = \{n \leq 0, ret = 0\}$
$2 : take(l, n : ret) = \{n \geq 1, l = 0, ret = 0\}$
$3 : take(l, n : ret) = \{n > 0, l > 0, t = l - 1, n' = n - 1, ret = ret' + 1\}, 1, take(t, n' : ret')$

Figure 1.14.: Program 8: Example with multiple bounds

Multiple bounds

Programs often have several (possibly incomparable) upper bounds. For example, Program 8 in Figure 1.14 has bounds $\|n\|$ and $\|l\|$ (altough these bounds can be combined into a single bound $min(\|n\|, \|l\|)$). Most existing cost analyses will obtain only one of the bounds. Unfortunately, this can be problematic in an incremental approach in which the result of the bound computation is used in all locations where *take* is called. For instance, consider the expression `take(l,2)` where `l` has 1000 elements. If the cost analysis obtains the upper bound $\|l\|$ for *take*, it assigns the cost 1000 to the expression `take(l,2)` instead of 2. It could get worse, if the length of `l` is unbounded or unknown, the analysis fails to provide a bound because it chose the wrong bound when solving *take*. To prevent that without breaking the modularity of the approach, it is necessary to be able to infer and represent multiple bound candidates for each CR compactly.

Sequential Evaluation in Non-terminating Programs

Depending on the selected cost model, it is possible to have a non-terminating program with finite cost. For example, a server that runs forever with a limited amount of memory. Consider the cost equation $g(x) = f(x), 10$. If the call to f diverges, the cost 10 is never consumed. This is not taken into account in any of the existing bound inference procedures.

Furthermore, assume that the definition of f is as follows:

$$f(x : y) = \{x = y = 0\}$$
$$f(x : y) = \{x < 0, x' = x - 1\}, f(x' : y)$$
$$f(x : y) = \{x > 0, x' = x - 1\}, f(x' : y)$$

The CR f diverges if x is negative. The input-output relation computed in the cost relation extraction procedure (Section 1.2.2) is $x \geq y = 0$. This input-output relation is only valid for the cases where f finishes. The resulting CE of g is 1: $g(x) = f(x), \{x \geq y = 0\}, 10$. Such a CE is not equivalent to 2: $g(x) = \{x \geq y = 0\}, f(x), 10$ which is guaranteed to terminate. Unfortunately, the descriptions and semantics presented in [AAGP11, AGM13, ABAG13] assume cost equations with a single constraint set that is applied at the beginning. Thus, they cannot distinguish between CEs 1 and 2 which can lead to unsoundness.

The approach presented in this thesis does not have this problem because it considers cost relations with multiple constraint sets and considers non-termination explicitly. Moreover, the new approach skips the input-output size analysis of the cost relation extraction so, in practice, it also receives cost equations with only one constraint set at the beginning.

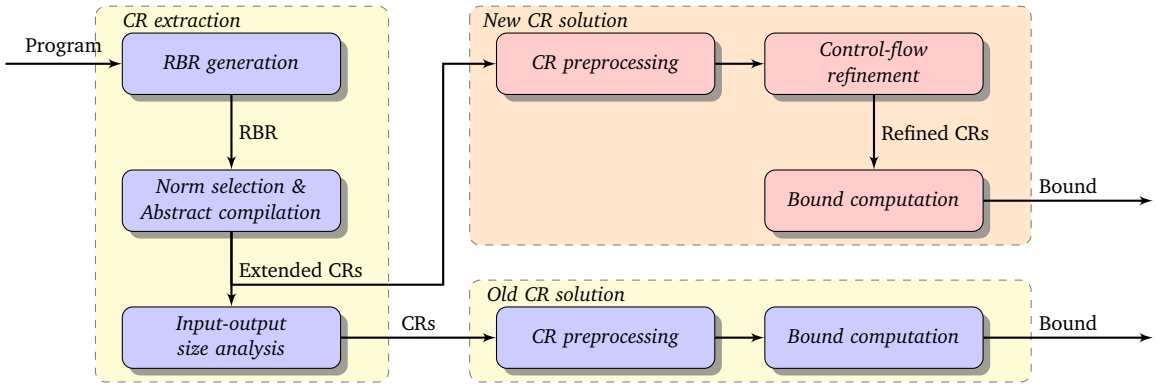


Figure 1.15.: Cost relation based cost analysis: Old versus new

1.3 Contributions

This dissertation extends the notion of cost relations and provides new methods to solve them that overcome the previously mentioned limitations. Cost relations are extended with the inclusion of output variables, support for multiple constraint sets, and support for both positive and negative costs (although the bound computation is limited for CRS with negative costs).

Cost relations are also given a new denotational semantics that considers non-terminating executions explicitly. This semantics facilitates reasoning about cost compositionally and it constitutes a solid foundation for future cost analyses based on cost relations.

The new method for solving cost relations is called CoFloCo (**Co**ntrol **Flo**w refinement of **Co**st Relations) and is divided into three phases:

1. Preprocessing: This phase reduces any mutually recursive cost relations to cost relations that only have direct recursion using unfolding. This phase existed in previous approaches but it has been reformulated for the extended cost relations.

2. Control-flow refinement: This phase refines the cost relations incrementally. For each cost relation, it partitions all possible evaluations into a set of execution patterns, called *chains*. The execution patterns are simpler than the complete CR and more precise invariants can be inferred for each of them. The results of the refinement can be propagated to other cost relations. The control flow refinement distinguishes explicitly between terminating and non-terminating execution patterns.

3. Bound computation: This phase infers upper and lower bounds for each of the cost relations, also incrementally. The main element of the bound inference is a novel cost representation called *cost structure* that can represent multiple complex upper and lower bounds. With cost structures, the bound inference and composition can be reduced to the solution of (relatively) small linear programming problems that can be performed efficiently.

All steps in the analysis have been proven sound with respect to the cost relation semantics.

Although this method focuses on the second phase of the cost relation approach, i.e. the cost relation solving, small modifications have been done on the overall approach to obtain cost relations. Figure 1.15 contains a diagram with the complete approach divided in two phases *CR extraction* and *CR solution* and the modified approach with the *New CR solution* method. As mentioned in the previous section, in order to obtain amortized costs it is necessary to take the output variables of the cost relations into account. Therefore, the new approach skips the *input-output size analysis* of the cost relation extraction procedure (see Section 1.2.2).

This step had two purposes: To remove the output variables from the cost relations and to enrich the cost relations with input-output size relations. In the new approach, input-output size relations are inferred during the control-flow refinement phase. This has additional advantages:

- The size relations inferred with the input-output size analysis are often too imprecise. This is because the size analysis ignores the internal control-flow of the cost relations and obtains a single input-output size relation (a linear constraint set) for each cost relation. These size relations inferred during the control-flow refinement phase can be much more precise because they are specialized for each possible execution pattern and they are generated taking the control flow of the cost relations into account.

- In the new approach, the cost relation extraction becomes simpler because part of the reasoning is moved to the cost relation solution. This is positive because the first phase of the analysis, i.e. the cost relation extraction, is language dependent whereas the second phase can be re-used across different languages. Therefore, making the first phase simpler facilitates the creation of new frontends and the application of this approach to other languages.

The techniques described in this dissertation have been implemented in an open source tool called CoFloCo[7] and an extensive experimental evaluation has been conducted. CoFloCo has been used to analyze imperative programs written in C, functional programs written in ABS and cost relations generated from term rewrite systems. CoFloCo has been compared to the following state-of-the-art tools: Loopus [SZV17], KoAT [BEF+16], C4B [CHS15], PUBS [AGM13, ABAG13], Rank [ADFG10] and RAML [HDW17].

1.3.1 Overview of the Publications

The papers published during my PhD are included below in two categories: papers whose results are included in this thesis and others. Within each category, the papers are presented in chronological order. Each item contains a short description of the work, its relation to this thesis, and whether I was the main author of the paper.

Publications Included in this Thesis

- *Resource analysis of complex programs with cost equations* (APLAS 2014) [FH14] [Main author]: This paper presents a control-flow refinement of cost relations and a bound computation method for cost relations with input and output variables. Chapter 5 contains an updated and extended version of the control-flow refinement. The bound computation method has been superseded by an improved method from a later publication [Flo16].

- *Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations* (FM 2016) [Flo16] [Single author]: This paper presents a method to solve cost relations that are the result of the control-flow refinement presented in [FH14]. This method can infer upper and lower bounds and presents increased precision for programs that present amortized cost. Chapter 6 contains an updated and extended version of this algorithm.

Other Publications

- *May-happen-in-parallel based deadlock analysis for concurrent objects* (FMOODS/FORTE 2013) [FAG13] [Main author]: This paper presents a deadlock analysis for programs written in a language for concurrent objects (ABS [JHS+11]) based on the results of a points-to analysis and a may-happen-in-parallel analysis [AFG12].

[7] https://github.com/aeflores/CoFloCo/

- *Termination and Cost Analysis of Loops with Concurrent Interleavings* (ATVA 2013) [AFGM13]: This paper presents a termination and cost analysis for programs written in ABS. This analysis uses the cost relation approach but focuses on the first part of the analysis. In particular, it focuses on dealing with the concurrent interleaving among different parts of a distributed system and it uses cost relation solvers (like the one presented in this thesis) as a black box.

- *SACO: Static Analyzer for Concurrent Objects* (TACAS 2014) [AAF+14]: This is a tool paper that presents SACO a static analysis tool for ABS programs that integrates several analyses such as termination, cost, deadlock and may-happen-in-parallel. Although the description of this tool is not included in the dissertation, the prototype implementation of the analysis presented in this dissertation (CoFloCo) has been integrated in the tool SACO and SACO has been used in the experimental evaluation.

- *May-Happen-in-Parallel Analysis with Condition Synchronization* (FOPARA 2015) [AFG15]: This publication presents an extension of the may-happen-in-parallel analysis of [AFG12] to treat additional synchronization mechanisms.

- *May-Happen-in-Parallel Analysis for Actor-Based Concurrency* (TOCL 2016) [AFGM16]: This paper is the journal version of [AFG12]. It presents an extended may-happen-in-parallel for ABS programs with increased precision. It includes an improved formalization of the analysis, soundness proofs and a more detailed discussion about the complexity of the analysis.

- *Rely-Guarantee Termination and Cost Analyses of Loops with Concurrent Interleavings* (JAR 2017) [AFGM17]: This paper is the journal version of [AFGM13]. It extends the analyses of [AFGM13] with several improvements and includes detailed soundness proofs.

1.3.2 Structure of the Dissertation

The rest of the dissertation is organized as follows:

Chapter 2: Informal Account This chapter provides an informal description of the analysis based on the examples discussed so far.

Chapter 3: Technical Background This chapter establishes the notation, defines cost relations formally, their semantics, their cost and the definitions of upper and lower bounds.

Chapter 5: Preprocessing This chapter details several cost preserving transformations that are used to simplify cost relations systems and reduce indirect recursion to direct recursion.

Chapter 5: Refinement This chapter presents the control-flow refinement of cost relations together with its soundness proofs. The chapter includes control-flow refinement presented in [FH14] adapted to the new semantics. The chapter also includes:

- An extension of this work to support cost relations with multiple (non-linear) recursion
- A more detailed discussion on the inference of two types of invariants: *chain summaries* and *calling contexts*
- Soundness proofs

Chapter 6: Bound computation This chapter presents the bound computation algorithm. This algorithm infers cost structures for each of the execution patterns detected in the control-flow refinement phase. The algorithm is an extension of the one presented in [Flo16]. In addition to the original algorithm, the chapter includes:

- An extension of the algorithm to support cost relations with multiple (non-linear) recursion

- A description of how to infer piece-wise defined bounds from the results of the bound analysis
- Soundness proofs

Chapter 7: Evaluation This chapter includes an extensive experimental evaluation of the tool CoFloCo divided in several parts:

Imperative programs First, CoFloCo is evaluated against a benchmark of (small) challenging programs written in C and taken from the literature. The tool is used to obtain upper and lower bounds and it is compared to other state-of-the-art tools: Loopus [SZV17], KoAT [BEF+16], C4B [CHS15], PUBS [AGM13, ABAG13] and Rank [ADFG10].

Second, the evaluations from the work [SZV17] are replicated with the latest version of CoFloCo. These consist on the analysis of a large benchmark of C programs (1650 functions) and the analysis of a reduced set of challenging loop iteration patterns. In these evaluations, only upper bounds are computed.

Functional Programs CoFloCo is compared to RAML [HDW17] on a small benchmark of examples taken from the evaluation of RAML. The examples are translated by hand from Ocaml to ABS and SACO [AAF+14] is used to generate cost relations from the ABS programs.

Term Rewrite Systems Recently, a translation from term rewrite systems to cost relations has been implemented [NFB+17] and an evaluation has been performed on the examples from the category "Runtime Complexity - Innermost Rewriting" of the Termination Competition 2016[8]. Here, the cost relations resulting from this translation have been analyzed with the latest version of CoFloCo and with PUBS [ABAG13].

Chapter 8: Related Work This chapter discusses related work.

Chapter 9: Conclusion This chapter concludes this thesis by summarizing its contributions.

Chapter 10: Limitations and Future Work This chapter discusses some limitations of the present work and possible directions for future work.

Appendix A: Implementation Details This appendix describes some details regarding the implementation of CoFloCo.

[8] http://termination-portal.org/wiki/Termination_Competition/

2 Informal Account

This chapter contains an informal description of the different phases of the analysis and an illustration of how it obtains bounds of the challenging examples presented in the introduction (Section 1.2.4).

2.1 Preprocessing

The starting point of the analysis is a cost relation system that describes the cost of a program. All approaches for solving cost relations share a common preprocessing step. This step detects the strongly connected components in the cost relations' call graph and transforms the mutually recursive definitions into direct recursion using unfolding. Once this step is completed, the cost relations can be sorted in a sequence $\langle C_1, C_2, \cdots, C_n \rangle$ such that each C_i can only contain recursive calls to C_i and non-recursive calls to C_j with $j > i$.

Consider Program 3 (in Page 7) whose cost relations without output variables are in Figure 1.8 (in Page 9). The cost relations of *popSome* and if_2 can be reduced to direct recursion by unfolding the call to if_2 in CR *popSome*. In addition, the cost relations can be further simplified by unfolding the call to if_1 in CR *wh*. The result of the unfolding is in Figure 2.1. Note that the cost relation *popSome* contains the return variable of the function *so*. The output variables of *wh* have not been included to keep the presentation simple (they are not necessary to obtain a precise cost). The ordered sequence of cost relations is $\langle amortized, wh, popSome \rangle$. The rest of the analysis is performed incrementally and bottom-up, that is, starting from CR *popSome* and finishing with CR *amortized*.

2.2 Cost Relation Control-flow Refinement

The next step of the analysis is the control-flow refinement of the cost relations. For each cost relation the refinement procedure generates a call-graph where the nodes of the call-graph are the CEs. Let c and c' be CEs, there is an edge $c \to c'$ if and only if c can call c'. The algorithm uses the CEs' constraint sets to compute the edges. Figure 2.2 illustrates the call-graph of Program 6 (in Page 13).

Once the call-graph has been created, the refinement procedure enumerates the possible patterns of evaluation (denoted *chains*) within the graph. Consider CEs $c_1, \cdots, c_n$ that form a strongly connected component S in the call-graph. The evaluation of S gives rise to *phases* represented with the notation $(c_1 \vee \cdots \vee c_n)^+$ to denote one or more evaluations of CEs in S; $(c_1 \vee \cdots \vee c_n)$ to denote a single evaluation or $(c_1 \vee \cdots \vee c_n)^\omega$ to denote an infinite sequence of evaluations of CEs in S. The first phase we call an *iterative* phase, for example $(2)^+$; the second one a *non-iterative* phase, for example (1); and the third

Cost relations of Program 3

1: $amortized(l,s) = wh(l,s)$

2: $wh(l,s) = \{l = 0\}$
3: $wh(l,s) = \{l > 0, s \geq 0, s' = s+1, l' = l-1\}, 1, popSome_1(s' : s'^2), wh(l', s'^2)$
4: $wh(l,s) = \{l > 0, s \geq 0, s' = s+1, l' = l-1\}, 1, wh(l', s')$

5: $popSome(s : so) = \{s = so\}$
6: $popSome(s : so) = \{s > 0, tmp = s-1\}, 1, popSome(tmp : so)$

Figure 2.1.: Cost relations of Program 3 with the output variable *so*

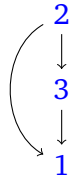
Cost equations of Program 6		Call-graph	Chains	
1 : $wh(i,n,r) =$	$\{i \geq n\}$		Terminating	Non-terminating
2 : $wh(i,n,r) =$	$\{i < n,\ r > 0,\ i' = 0,\ r' = r-1\},\ 1,$ $wh(i',n,r')$		$(2)^+(3)^+(1)$	$(2)^+(3)^\omega$
3 : $wh(i,n,r) =$	$\{i < n,\ r \leq 0,\ i' = i+1\},\ 1,$ $wh(i',n,r)$		$(3)^+(1)$	$(3)^\omega$
			$(2)^+(1)$	$(2)^\omega$
			(1)	

Figure 2.2.: Cost Equations, call-graph and chains of Program 6

one a *divergent* phase, for example, $(2)^\omega$. A *chain* is a sequence of phases. The chains of Program 6 are displayed on the right part of Figure 2.2 divided between terminating and non-terminating.

Next, the algorithm attempts to prove termination of each divergent phase by obtaining a lexicographical ranking function [BAG14]. If it succeeds, it discards all non-terminating chains ending in that phase. In Program 6, the expression r is a ranking function of $(2)^\omega$ and $n-i$ is a ranking function of $(3)^\omega$. Therefore, the algorithm discards all non-terminating chains of CR *wh*.

2.2.1 Invariants

A pivotal aspect of the approach is to propagate information forward and backward along each chain by polyhedral invariant computation. For instance, the algorithm computes a summary of a chain by propagating information backward along the chains. Conversely, calling contexts are computed by propagating information forward. In the case of chain $(2)^+(3)^+(1)$ of Program 6, it starts with the constraint set of CE 1 and applies the constraint set of CE 3 repeatedly until reaching a fixpoint. Then, it applies the constraint set of CE 2 to the result, again repeatedly, until reaching a fixpoint. The resulting summary for $(2)^+(3)^+(1)$ is simply $i < n \wedge r > 0$ which provides a necessary precondition for the whole chain. In the case where a cost relation contains output values, the chain summary relates the input and output variables.

Consider the cost relations of Program 3 in Figure 2.1. The resulting chains of CR *popSome* are (5) and $(6)^+(5)$ (the non-terminating chain $(6)^\omega$ is discarded because it can be shown to be terminating). The chain summaries of (5) and $(6)^+(5)$ are $s = so$ and $s \geq 1 \wedge s > so$, respectively. These summaries relate the input and output values of *popSome*.

2.2.2 Refinement Propagation

The result of the refinement of a CR C is a set of its feasible chains with their propagated summaries. This refinement can then be further propagated to other CRs that call C. Recall that this process starts with the "innermost" cost relation of a program that does not make any call except to itself (for example, *popSome* in Program 3) and proceeds backwards to the "outermost" cost relation.

Consider the situation in Program 3. The refinement procedure simply substitutes the calls to *popSome* in CR *wh* by calls to the chains of *popSome* (that is (5) and $(6)^+(5)$). Additionally, it enriches the constraint set of each refined CE with the summary of the called chain. Hence, CE 3 is specialized into the following two CEs (the underlined constraints are the added summaries resulting from the called chain):

3.1 : $wh(l,s) = \{l > 0, s \geq 0, s' = s+1, l' = l-1, \underline{s' = s'^2}\},\ 1,\ popSome[(5)](s' : s'^2), wh(l', s'^2)$

3.2 : $wh(l,s) = \{l > 0, s \geq 0, s' = s+1, l' = l-1, \underline{s' \geq 1, s' > s'^2}\},\ 1,\ popSome[(6)^+(5)](s' : s'^2),$ $wh(l', s'^2)$

If the chain $[(6)^\omega]$ was feasible, we would also generate the cost equation:

$$3.3 : wh(l,s) = \{l > 0, s \geq 0, \leq s' = s+1, l' = l-1\},\ 1,\ popSome[(6)^\omega](s' : s'^2)$$

in which the cost of $wh(l', s'^2)$ has been removed because the call to $popSome[(6)^\omega](s' : s'^2)$ does not terminate.

To proceed, the refinement computes the call-graph, feasible chains, and summaries of the resulting CR *wh*. Its chains are $(3.1 \vee 3.2 \vee 4)^+(2)$ and (2).

2.3 Computing Bounds with Cost Structures

Once the refinement of all CRs is completed, we have to compute bounds. Similarly to the refinement, the bound computation procedure works incrementally, following a bottom-up approach, from the innermost to the outermost CR. Inside a single CR, it follows an incremental approach as well:

1. Compute the bound for each cost equation without considering recursive calls, i.e. the cost of a cost equation evaluation
2. Compute the cost of each phase by composing the cost of their CEs
3. Compose the cost of the phases to obtain the cost of the chains

Therefore, the key aspect of the analysis is to represent cost bounds in such a way that they can be inferred and composed efficiently and precisely at each level (CE, phase and chain). This is done thanks to a novel data structure called *cost structure*.

2.3.1 Cost Structures

A *cost structure* represents a set of costs with a triple $\langle E, IC, FC(\boldsymbol{x})\rangle$. In a cost structure, E is a linear expression over *intermediate variables* (*iv*) that represents the cost. These intermediate variables are related to the variables of the CRs through two sets of constraints: *Final FC* and *non-final IC* constraints. Both constraint sets admit only constraints of a restricted form:

- *Non-final* constraints *IC* are expressions $\sum_{k=1}^{m} iv_k \leq SE$ [1] where *SE* is of the form

$$SE \quad := \quad l(\boldsymbol{iv}) \mid iv_1 \cdot iv_2 \mid \max(\boldsymbol{iv}) \mid \min(\boldsymbol{iv})$$

  Here $\boldsymbol{iv}$ is a sequence of intermediate variables $iv_1, iv_2, \cdots, iv_n$ and $l(\boldsymbol{iv})$ is a linear expression over the intermediate variables in $\boldsymbol{iv}$.

- The *final* constraints *FC* are expressions of the form $\sum_{k=1}^{m} iv_k \leq \|l(\boldsymbol{x})\|$, where $l(\boldsymbol{x})$ is a linear expression over the CR variables $\boldsymbol{x}$ and $\|l(\boldsymbol{x})\| = \max(l(\boldsymbol{x}), 0)$.

This data structure is able to represent complex polynomial bounds with maximum and minimum operators. At the same time, it makes it possible to define the inference and composition of cost structures in terms of simple rules and heuristics for each kind of constraint.

Note that this representation achieves separation of concerns. Instead of having a monolithic cost expression, there are: (1) A set of basic expressions $\|l(\boldsymbol{x})\|$ over the CR variables in the final constraints; (2) Non-linear combinations of these basic expressions using the non-final constraints and (3) a simple expression E that represent the cost. *Intermediate variables* are simply names that connect the three components of the cost structure.

[1] Cost structures can also be equipped with $\geq$ constraints instead of $\leq$ to infer lower bounds (see Chapter 6).

Example 2.1. Function *take* in Program 8 (Figure 1.14 in Page 15) is an example of a function with multiple upper bounds. The cost structure representation of the cost of this program is:

$$\langle iv, \emptyset, \{iv \leq \|n\|, iv \leq \|l\|\}\rangle$$

This cost structure represents the cost $min(\|n\|, \|l\|)$. Program 8 has two candidate upper bounds: $\|n\|$ and $\|l\|$. Each of these candidates is represented with a constraint over iv and is inferred independently.

2.3.2 Bound Computation

Recall the three steps of computing bounds mentioned in the introduction to this section. Steps **1** and **3** involve a finite composition of cost structures. There is a specific number of cost structures and we have to compute their sum and express it in terms of different variables. For instance, to obtain the cost of $(2)^+(3)^+(1)$ of Program 6, we have to compose three cost structures (those of $(2)^+$, $(3)^+$ and (1)) and express the result in terms of the initial variables (i, n, r) of the chain. This involves the following steps: to sum up the main cost expressions of each cost structure, to merge their non-final and final constraint sets, and to transform the final constraints so they are expressed in terms of the initial variables.

The transformation is based on the constraint sets of the CEs and the inferred summaries from the refinement. Final constraints are almost linear so the transformation can be implemented using Fourier-Motzkin quantifier elimination.

Example 2.2. Let $\langle iv_3, \emptyset, \{iv_3 \leq \|r\|\}\rangle$, $\langle iv_4, \emptyset, \{iv_4 \leq \|n - i\|\}\rangle$, and $\langle 0, \emptyset, \emptyset\rangle$ be the cost structures of $(2)^+$, $(3)^+$, and (1), respectively. Then, the composed cost structure of $(2)^+(3)^+(1)$ is $\langle iv_3 + iv_4, \emptyset, \{iv_3 \leq \|r\|, iv_4 \leq \|n\|\}\rangle$ which represents the bound $\|r\| + \|n\|$. Observe that during the phase $(2)^+$ variable i is set to 0. Therefore, the expression $n - i$ of phase $(3)^+$ is n in the chain $(2)^+(3)^+(1)$.

Step **2**, computing the cost of phases, involves the composition of an unknown number of cost structures. To realize this, the procedure generates fresh intermediate variables that represent the sums of all the instances of the previous intermediate variables. Then, it applies different strategies to generate constraints over these new intermediate variables from the constraints of the original variables.

Example 2.3. Let us compute the cost of $(2)^+$ in Program 6. According to the definition of CE 2, its cost (ignoring the recursive call) is 1. This can be expressed as $\langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$. Assume the jth evaluation of CE 2 has cost $\langle iv_{1_j}, \emptyset, \{iv_{1_j} \leq 1\}\rangle$. Now assume that CE 2 is evaluated $\#c_2$ times in $(2)^+$. Based on that, we create a new intermediate variable $iv_3 := \sum_{j=1}^{\#c_2} iv_{1_j}$ that represents the sum of all instances of iv_1. Now the bound of $(2)^+$ can be expressed as iv_3 and we have to generate constraints that bind iv_3 using $iv_1 \leq 1$ and the constraint set of CE 2.

One of the strategies (called *Inductive Sum*) consists of applying Farkas' Lemma with a linear template $L(i, n, r)$ and the constraint set $\varphi_2 = \{i < n, r > 0, i' = 0, r' = r - 1\}$ of CE 2 (Figure 1.12) to obtain a symbolic expression that satisfies:

$$\varphi_2 \Rightarrow \big(L(i, n, r) \geq 1 \quad \wedge \quad L(i, n, r) \geq 1 + L(i', n', r')\big)$$

The expression r is a valid instantiation of $L(i, n, r)$ and it is a valid upper bound of iv_3. In this case, r is essentially a linear ranking function of $(2)^+$. The resulting cost structure of $(2)^+$ is $\langle iv_3, \emptyset, \{iv_3 \leq \|r\|\}\rangle$ which is the one that was used in Example 2.2. This strategy is based on the same idea as the Tree-Sum method from [ABAG13], but as we will see in the next example, it can deal with cases where the Tree-Sum method fails.

The same process can be applied to obtain a bound for the phase $(3)^+$. The cost of applying CE 3 is 1 which can be represented with the cost structure $\langle iv_2, \emptyset, \{iv_2 \leq 1\}\rangle$. As before, we define a new intermediate variable $iv_4 := \sum_{j=1}^{\#c_3} iv_{2_j}$, and generate constraints over this new variable using the

Chain/Phase/CE: Cost Structure

$(2 \vee 3)^+ : \langle iv_3 + iv_4, \{iv_4 \leq iv_5 + iv_6, iv_6 \leq iv_3 \cdot iv_7\}, \{iv_5 \leq \|n-i\|, iv_3 \leq \|r\|, iv_7 \leq \|n\|\}\rangle$

$2 : \langle iv_1, \emptyset, iv_1 \leq 1\rangle$

$3 : \langle iv_2, \emptyset, iv_2 \leq 1\rangle$

New iv definitions: $iv_3 := \sum_{j=1}^{\#c_2} iv_{1j}$ $iv_4 := \sum_{j=1}^{\#c_3} iv_{2j}$ $iv_6 := \sum_{j=1}^{\#c_3} n_j$ $iv_7 := \max_{j=1}^{\#c_3}(n_j)$

Figure 2.3.: Cost structure of phase $(2 \vee 3)^+$ in Program 7 and the fresh intermediate variables defined in the process.

constraint set of CE 3. Let $\varphi_3 = \{i < n, r \leq 0, i' = i + 1\}$ be the constraint set of CE 3, we instantiate a linear template $L(i,n,r)$ such that $\varphi_3 \Rightarrow (L(i,n,r) \geq 1 \wedge L(i,n,r) \geq 1 + L(i',n',r'))$. As a result, we obtain the expression $n-i$ and the resulting cost structure for the phase $(3)^+$ is $\langle iv_4, \emptyset, \{iv_4 \leq \|n-i\|\}\rangle$.

2.3.3 Loop with Reset

Let us consider Program 7 (Figure 1.13 in Page 14). In this program CE 2 and CE 3 can interleave (CE 3 no longer has the condition $r \leq 0$) which affects their cost. This example is interesting because it makes use of non-final constraints to represent a non-linear bound. The main chain of the example is $(2 \vee 3)^+(1)$. The cost of CE 1 is 0 so let us focus on the phase $(2 \vee 3)^+$.

Figure 2.3 displays the cost structure of phase $(2 \vee 3)^+$ and the fresh intermediate variables defined in the process. In these definitions $\#c_N$ represents the number of times CE N is applied. The computation proceeds incrementally. It starts with the cost structures $\langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$ and $\langle iv_2, \emptyset, \{iv_2 \leq 1\}\rangle$ for CE 2 and 3, respectively. The variables iv_3 and iv_4 are defined in Figure 2.3 and the main cost expression is $iv_3 + iv_4$.

Using the Inductive Sum strategy (cf. Example 2.3), we infer the bounds r and $n-i$ for iv_3 and iv_4, respectively. However, in contrast to the previous examples, these bounds can be influenced by the interleavings of other CEs in the same phase.

Expression r is unmodified in CE 3, so we can generate the constraint $iv_3 \leq \|r\|$. However, expression $n-i$ is reset in CE 2 to n (if i is set to 0 and n is not changed, $n-i$ is set to n). Hence, we add the sum of all these resets to n to obtain a bound of iv_4. We generate the constraints $iv_4 \leq iv_5 + iv_6$ and $iv_5 \leq \|n-i\|$ where iv_6 represents the sum of all the resets to n in CE 2.

Finally, there is no linear expression that can bind iv_6 (the sum of all n in CE 2). Therefore, we apply another strategy (*Basic Product*) that binds iv_6 to the product of the number of iterations of CE 2 (iv_3) and the maximum value of n along the execution (iv_7). The generated constraints are $iv_6 \leq iv_3 \cdot iv_7$ and $iv_7 \leq \|n\|$ (n does not change along the execution) and the cost structure is now complete.

2.3.4 Amortized cost example

Figure. 2.4 contains the cost structures needed for computation of the cost of chain $(3.1 \vee 3.2 \vee 4)^+(2)$ of Program 3 (obtained at the end of Section 2.2.2). Additionally, it contains the intermediate variable definitions used for the computation of the cost of phases. As before, $\#c_N$ represents the number of times CE N is applied. The final cost structure of $(3.1 \vee 3.2 \vee 4)^+(2)$ represents the bound $\|l\| + \|l+s\|$ which is precise.

A key aspect in obtaining amortized cost is to consider the final values of variables. In the computation of CE 6's cost structure, the input variable of the recursive call s' is taken into account. In the computation of phase $(6)^+$'s cost structure, the input variable of the last recursive call of the phase (s_f)

Chain/Phase/CE(Variables): Cost Structure

$[(3.1 \vee 3.2 \vee 4)^+(2)](l,s) : \langle iv_6 + iv_7 + iv_8 + iv_9, \emptyset, \{iv_7 + iv_8 + iv_9 \leq \|l\|, iv_6 \leq \|l+s\|\}\rangle$

- $(3.1 \vee 3.2 \vee 4)^+(l,s:l_f,s_f) : \langle iv_6 + iv_7 + iv_8 + iv_9, \emptyset, \{iv_7 + iv_8 + iv_9 \leq \|l\|, iv_6 \leq \|l+s\|\}\rangle$
  - $3.1(l,s:l',s') : \langle iv_3, \emptyset, \{iv_3 \leq 1\}\rangle$
    - $5(s:so) : \langle 0, \emptyset, \emptyset\rangle$
  - $3.2(l,s:l',s') : \langle iv_2 + iv_4, \emptyset, \{iv_4 \leq 1, iv_2 \leq \|s+1-s'\|\}\rangle$
    - $[(6)^+(5)](s:so) : \langle iv_2, \emptyset, \{iv_2 \leq \|s\|, iv_2 \leq \|s-so\|\}\rangle$
      - $(6)^+(s:s_f) : \langle iv_2, \emptyset, \{iv_2 \leq \|s\|, iv_2 \leq \|s-s_f\|\}\rangle$
        - $6(s:s') : \langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$
      - $(5)(s:so) : \langle 0, \emptyset, \emptyset\rangle$
  - $4(l,s:l',s') : \langle iv_5, \emptyset, \{iv_5 \leq 1\}\rangle$
- $(2)(l,s) : \langle 0, \emptyset, \emptyset\rangle$

New iv definitions:

$$iv_2 := \sum_{j=1}^{\#c_6} iv_{1j} \qquad iv_6 := \sum_{j=1}^{\#c_{3.2}} iv_{2j} \qquad iv_7 := \sum_{j=1}^{\#c_{3.1}} iv_{3j} \qquad iv_8 := \sum_{j=1}^{\#c_{3.2}} iv_{4j} \qquad iv_9 := \sum_{j=1}^{\#c_4} iv_{5j}$$

Figure 2.4.: Cost structures of Program 3 and intermediate variables defined in the process.

is also considered. In fact iv_2 is bound by $\|s-s_f\|$. Intuitively, the number of recursive calls is bound by the initial value of s minus its final value s_f. In chain $(6)^+(5)$ the final value of s (s_f) corresponds to the return value so_f (consider $s = so$ in CE 5) and variable so is unchanged throughout phase $(6)^+$ ($so_f = so$). Therefore, we have $s_f = so_f = so$ and we obtain the constraint $iv_2 \leq \|s-so\|$ for the chain $(6)^+(5)$.

Similarly, the cost structure of CE 3.2 depends on the value of the variables in the recursive call ($iv_2 \leq \|s+1-s'\|$). Applying the Inductive Sum strategy we can infer that $\|l+s\|$ is an upper bound of the sum of all the instances of $\|s+1-s'\|$ (and also of all iv_2). Let $\varphi_{3.2} = \{l > 0, s \geq 0, s' = s+1, l' = l-1, s'^2 < s'\}$ be the constraint set of CE 3.2, we have that

$$\varphi_{3.2} \Rightarrow \big((l+s) \geq (s+1-s') \quad \wedge \quad (l+s) \geq (s+1-s') + (l'+s')\big)$$

The inferred constraint is $iv_6 \leq \|l+s\|$. We could also infer $iv_6 \leq \|(l+s)-(l_f+s_f)\|$ but it is not needed here.

In the cost computation of phase $(3.1 \vee 3.2 \vee 4)^+$ we have to ensure that the sums we infer are not reset or incremented in interleaving CEs. The expression $l+s$ stays invariant in CE 3.1 and 4 (l is decremented, s is incremented by 1). The expression l which bounds iv_7 is not incremented or reset in CE 3.2 or 4, but expression l also bounds iv_8 and iv_9 so the more precise constraint $iv_7 + iv_8 + iv_9 \leq \|l\|$ can be generated. The capability of bounding several intermediate variables with a single linear expression is essential to extend the approach to lower bounds.

3 Technical Background

This chapter establishes the notation used in the rest of the dissertation, the syntax and the semantics of cost relations. Then, based on the semantics, it formally defines the notions of upper and lower bounds. Finally, the chapter contains a definition of what constitutes a sound and precise cost relation transformation.

3.1 Basic Definitions

The symbol $\boldsymbol{x}$ represents a finite sequence of variables $x_1, x_2, \cdots, x_n$ of any length. The expression $\boldsymbol{xy}$ represents the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$. Similarly, $\boldsymbol{a}$ represents a sequence of constants $a_1, a_2, \cdots, a_n$ with $a_i \in \mathbb{Q}$. The symbol ω is defined such that $\omega > a$ holds for any $a \in \mathbb{Q}$.

Let $\boldsymbol{x} = x_1, x_2, \cdots, x_n$, the expression $\sum \boldsymbol{x}$ represents the sum of its elements $\sum \boldsymbol{x} := \sum_{i=1}^{n} x_i$. A *linear expression* is a term of the form $l := a_0 + a_1 x_1 + \cdots + a_n x_n$ where $a_i \in \mathbb{Q}$ and $x_1, x_2, \cdots, x_n$ are variables. A *linear constraint* is a predicate $lc := l \geq 0$ where l is a linear expression. For readability, linear constraints are often expressed as $l_1 \leq l_2$, $l_1 = l_2$ or $l_1 \geq l_2$. These can be easily transformed to the form above e.g. $l_1 = l_2$ is equivalent to the conjunction $l_1 - l_2 \geq 0 \wedge l_2 - l_1 \geq 0$. A *constraint set* φ is a set of linear constraints $\{lc_1, lc_2, \cdots, lc_n\}$ and represents its conjunction $lc_1 \wedge lc_2 \wedge \cdots \wedge lc_n$.

A variable assignment $\sigma : V \to D$ maps variables from the set of variables V to elements of a set D. The function $\mathrm{Dm}(\sigma) := V$ returns the domain of the variable assignment. The variable assignment $\sigma|_{V'}$ is the restriction of σ to the domain V'. The notation $\sigma := [x_1/t_1, \cdots, x_n/t_n]$ is used to denote a variable assignment that maps each x_i to t_i. This notation is also used for general substitutions whose domain in not necessarily a variable set. We lift variable assignments to arbitrary terms and formulae (e.g. linear expressions or constraint sets) as usual. Let t be a term or a formula $t\sigma$ denotes that the variable assignment σ is applied to t.

Let t be a term, $vars(t)$ is the set of variables in t. We often express the variables of t explicitly with the notation $t(\boldsymbol{x})$. Moreover, let $t(\boldsymbol{x})$ be a term or a formula over the variables $\boldsymbol{x} = x_1, x_2, \cdots, x_n$, then $t(\boldsymbol{y})$ represents an instantiation of t over the variables $\boldsymbol{y} = y_1, y_2, \cdots, y_n$ and it is equivalent to $t(\boldsymbol{y}) = t(\boldsymbol{x})[x_1/y_1, x_2/y_2, \cdots, x_n/y_n]$. We also use set notation (e.g. $\in$, $\subseteq$) directly on sequences of variables. For example, let S be a set of variables, $S \subseteq \boldsymbol{x}$ represents $S \subseteq vars(\boldsymbol{x})$.

A constraint set φ is *satisfiable* if there exists an assignment $\sigma : V \to \mathbb{Q}$ such that $\varphi\sigma$ is valid (expressed as $\models \varphi\sigma$). In such a case σ satisfies φ. We say that $\varphi \Rightarrow \varphi'$ if every assignment that satisfies φ satisfies φ' as well. The symbols $\top$ and $\bot$ represent valid and unsatisfiable constraint sets respectively.

Finally, note that a constraint set φ also represent a polyhedron. We use the notation $\varphi \downarrow \boldsymbol{x}$ to denote the *projection* of the polyhedron φ onto the variables in $\boldsymbol{x}$. This is equivalent to perform quantifier elimination over $\exists \boldsymbol{y}(\varphi)$ where $\boldsymbol{y} = vars(\varphi) \setminus vars(\boldsymbol{x})$.

3.2 Cost Relations

Definition 3.1 (Cost relation)**.** A *cost relation* (CR) C is a set of *cost equations* of the form:

$$c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, b_2, \cdots, \varphi_{n-1}, b_n, \varphi_n$$

where c is a unique identifier; C is a cost relation symbol; $\boldsymbol{x}$ and $\boldsymbol{y}$ are the input and output variables of C; φ_i are constraint sets; and b_i are either linear expressions $l_i(\boldsymbol{x_i})$ that represent costs or references to other cost relations $C_i(\boldsymbol{x_i} : \boldsymbol{y_i})$ where $\boldsymbol{x_i}$ and $\boldsymbol{y_i}$ are the input and output variables of C_i.

A cost equation (CE) $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ states that the cost of $C(\boldsymbol{x} : \boldsymbol{y})$ is the sum of the costs of each b_i. We refer to $head(c) := C(\boldsymbol{x} : \boldsymbol{y})$ as the head of c and $body(c) := \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ as the body of c. The constraint sets $\varphi_0, \cdots, \varphi_n$ serve two purposes: they restrict the applicability of the equation with respect to the variables in the head of the equation $\boldsymbol{xy}$ and they relate the variables of the different b_i with $\boldsymbol{xy}$ and among each other. Note that the linear expressions $l_i(\boldsymbol{x_i})$ can be negative to represent the deallocation or generation of resources.

One can view a CR C as a non-deterministic procedure that executes a cost equation $c \in C$. The execution of a CR C with respect to some parameters $\boldsymbol{a}$ proceeds as follows. First, a cost equation $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$ is selected; Then, a variable assignment σ is selected such that $\boldsymbol{a} = \boldsymbol{x}\sigma$, σ covers the input variables in b_1 and satisfies the constraint set φ_0. Next b_1 is recursively executed. Once the execution of b_1 is completed, the variable assignment σ is extended further to match the output values of b_1, to satisfy φ_1, and to cover the input variables of b_2. Then, b_2 is executed. This process continues until all b_i have been executed and the extended variable assignment σ satisfies all the constraint sets φ_i. Finally, σ is extended to cover the output variables $\boldsymbol{y}$. The result of the evaluation is $\boldsymbol{b} := \boldsymbol{y}\sigma$. If at any point the evaluation a call b_i diverges, the remaining calls are not evaluated. If at any point a constraint set φ_i cannot be satisfied, the evaluation fails.

Definition 3.2 (Cost Relation System)**.** A *cost relation system CRS* is a finite set of cost relations defined in terms of each other. That is, for every cost relation $C \subseteq CRS$, and for all its cost equations $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$. The cost equation c only contains references to other cost relations in the cost relation system $C_i \subseteq CRS$.

A cost relation system *CRS* can be seen as an abstract program defined in terms of a set of procedures that correspond to each cost relation. It can also be seen as a constraint logic program with a fixed resolution strategy (left to right) where each cost relation $C \subseteq CRS$ corresponds to a predicate and each cost equation $c \in C$ corresponds to a clause.

When we analyze a program, in general we are not interested in any possible execution of all of its fragments. Instead, we are only interested in the executions that start at one of the entry points of the program. Therefore, given a cost relation system *CRS*, we define a subset of cost relations $ES \subseteq CRS$ that represent its entries.

3.3 Semantics

There are multiple ways to define the semantics of cost relations. In [AAGP11] Albert et al. define a denotational semantics for cost relations in terms of evaluation trees. This semantics is very useful to reason about cost bounds in a compositional manner. However, it lacks the notion of sequential execution. This makes it inadequate to reason about non-terminating evaluations or about different notions of cost such as *peak cost*. The peak cost of a program, also known as *high-water mark*, is the maximum amount of resources consumed at any point of an evaluation. On the other hand, the papers [AGGZ13] and [ABG12] provide a small step operational semantics which can easily capture non-terminating evaluations and peak costs but is less adequate to reason about bound computation techniques compositionally.

This work defines a denotational evaluation semantics that maps a term $C(\boldsymbol{a} : \boldsymbol{b})$ where C is a cost relation within a cost relation system $C \subseteq CRS$ and $\boldsymbol{a} \in \mathbb{Q}^n$ $\boldsymbol{b} \in \mathbb{Q}^m$ and some input and output parameters to a set of evaluations. A term $C(\boldsymbol{a} : \boldsymbol{b})$ is mapped to a set of evaluations instead of a single evaluation because of the cost relations' non-determinism. In constrast to the semantics defined in [AAGP11], this semantics distinguishes finite and infinite evaluations. Finite evaluations are defined inductively and infinite evaluations co-inductively [LG09]. The semantics only considers evaluations that do not fail so an evaluation can be either complete or infinite.

A complete evaluation of CR C with respect to some parameters $\boldsymbol{a}$ and $\boldsymbol{b}$ is represented with a tree $t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$. The node of the tree $c(\boldsymbol{a} : \boldsymbol{b})$ contains the label of the selected CE $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$ and the parameters $\boldsymbol{a}$ and $\boldsymbol{b}$. Besides, there is an assignment

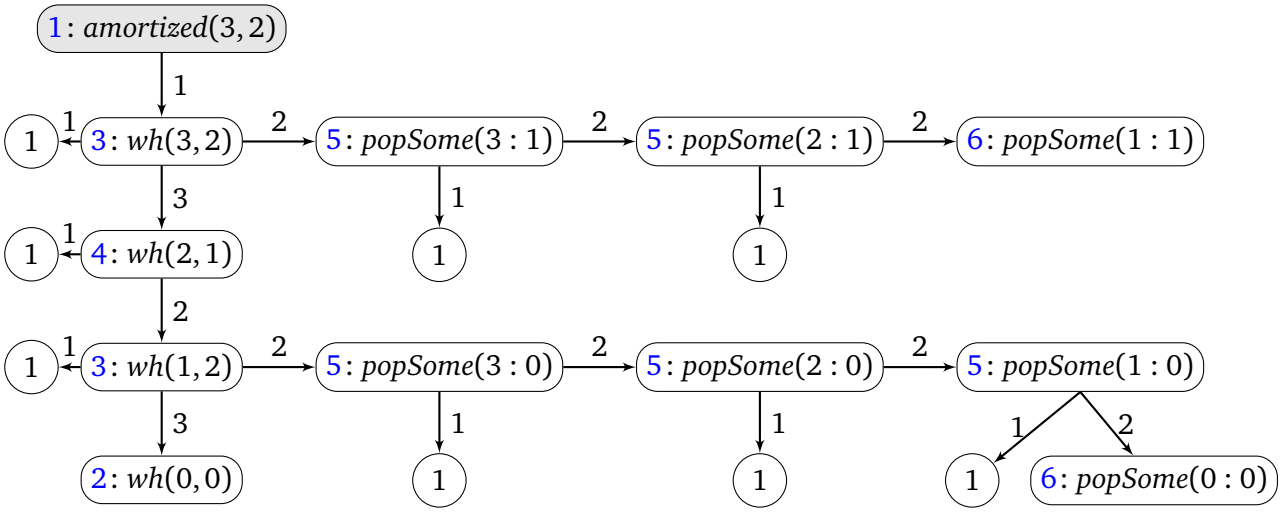


Figure 3.1.: Evaluation of Program 3

σ such that $\boldsymbol{ab} = \boldsymbol{xy}\sigma$, the children T_i are evaluations of $b_i\sigma$, and the constraints of the CE are satisfied $\models (\varphi_0 \wedge \cdots \wedge \varphi_n)\sigma$. Note that if b_i is a linear expression l, then $b_i\sigma$ is a rational number $l\sigma \in \mathbb{Q}$. An infinite evaluation of CR C is represented similarly with a tree $t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$ where CE $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$. The difference is that the evaluation might have diverged before reaching the last b_n. That is, we have $m \leq n$ and the last child T_m is also an infinite evaluation. Note that for infinite evaluations the output values $\boldsymbol{b}$ are meaningless but they are kept in the evaluations to maintain a uniform format.

Definition 3.3 (Cost Relation Semantics)**.** Let *CRS* be a cost relation system. First, we define (as a base case) the evaluation of linear expressions. A linear expression l has only one evaluation with respect to some parameters $\boldsymbol{a}$ which is its numeric value:

$$[\![CRS|l(\boldsymbol{a})]\!]_c := \{l(\boldsymbol{a})\}$$

Let $C \in CRS$ be a cost relation and let $\boldsymbol{a}$ and $\boldsymbol{b}$ be input and output parameters. The set of complete evaluations induced by C and $\boldsymbol{a} : \boldsymbol{b}$ is defined as:

$$[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c := \left\{ T \;\middle|\; \begin{array}{l} 1.\ c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C \\ 2.\ T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \\ 3.\ \exists \sigma \text{ such that } \mathrm{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma,\ \models (\bigwedge_{i=0}^{n} \varphi_i)\sigma \\ 4.\ \bigwedge_{i=1}^{n} T_i \in [\![CRS|b_i\sigma]\!]_c \end{array} \right\}$$

Conversely, the set of infinite evaluations is defined as:

$$[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega := \left\{ T \;\middle|\; \begin{array}{l} 1.\ c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C \\ 2.\ \exists m \leq n \text{ such that } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m]) \\ 3.\ \exists \sigma \text{ such that } \mathrm{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma,\ \models (\bigwedge_{i=0}^{m-1} \varphi_i)\sigma \\ 4.\ \bigwedge_{i=1}^{m-1} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_m \in [\![CRS|b_m\sigma]\!]_\omega \end{array} \right\}$$

Program 9

```
def Unit f(Int n)=
 if (n==0) Unit
 else  if (n>0){ tick(2); tick(-1); f(n-1); tick(-1);}
       else{ tick(1); tick(-1); f(n-1); tick(2);}
```

Cost relations

$1: f(n) = \{n = 0\}, 0$
$2: f(n) = \{n \geq 1, n' = n-1\}, 2, -1, f(n'), -1$
$3: f(n) = \{n \leq -1, n' = n-1\}, 1, -1, f(n'), 2$

Figure 3.2.: Program 9: Possibly non-terminating program with non-cumulative cost

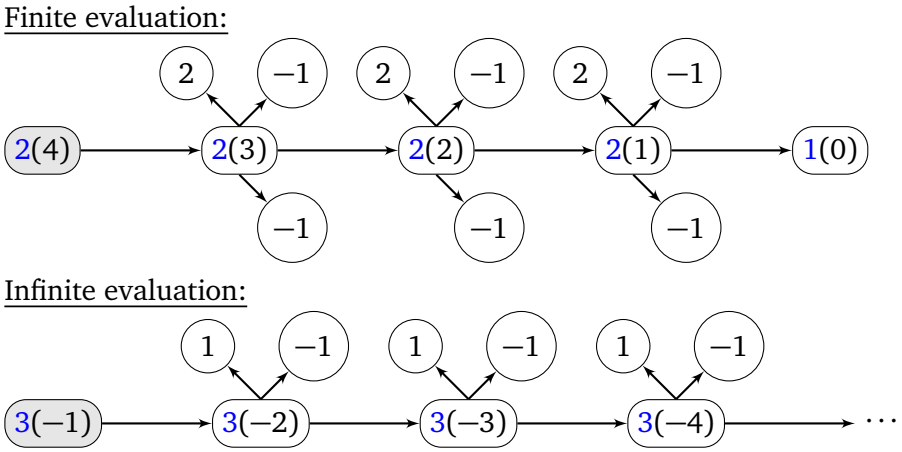


Figure 3.3.: Finite and infinite evaluation of Program 9

Both complete and infinite evaluations are very similar but infinite evaluations contain one infinite branch in the last place. The union of all possible evaluation sets $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ (respectively $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$) for every possible parameters $\boldsymbol{a}$ and $\boldsymbol{b}$ is denoted $[\![CRS|C]\!]_c$ (respectively $[\![CRS|C]\!]_\omega$) . The set $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!] := [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c \cup [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ represents all the evaluations (complete and infinite) of C in CRS with respect to some parameters $\boldsymbol{a} : \boldsymbol{b}$. In the cases where the cost relation system in which the evaluation takes place is clear, the simplified notation $[\![C]\!]_c$ and $[\![C]\!]_\omega$ is used.

Example 3.4. Figure 3.1 contains one of the possible complete evaluations of Program 3 (its cost relations are in Figure 2.1 in Page 21). The nodes that represent the evaluation of a cost relations are rounded rectangles and the ones that represent the evaluation of a linear expression are circles. The labels on the edges indicate the order of the sub-evaluations. The starting point of the evaluation is shadowed. To ease readability, the corresponding cost relation symbols have been included in the nodes in addition to the CE identifiers.

Example 3.5. Figure 3.2 contains a recursive function f (Program 9) that iterates until n is 0. If $n > 0$, function f allocates two resource units first and deallocates them later. It deallocates one unit before the recursive call and the other one after the recursive call has been completed. If $n < 0$, the program is non-terminating and it allocates and deallocates one resource unit in each iteration. Function f would also allocate 2 resource units after the recursive call is completed but this never happens. Note how the cost equations maintain the order of the resource consumption which is essential in this case. Figure 3.3 contains a complete and an infinite evaluation of Program 9. In Figure 3.3, edges are not labeled but the children of each tree are displayed in clockwise order.

Let us define some notation for evaluation trees. The expression $T' \preceq T$ denotes that T' is a descendant of T. Let $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$, the root of the tree is $rt(T) = c(\boldsymbol{a} : \boldsymbol{b})$, its label is $label(T) = c$, and

Program 10	Cost relations	
	Translation 1	Translation 2

`void f(int i){`
`g(i);`
`assert(i>=0);`
`tick(10);`
`}`
	$1: f(i) = g(i), \{i \geq 0\}, 10$	$1: f(i) = g(i), assert(i,b), \{b = 1\}, 10$
		$2: f(i) = g(i), assert(i,b), \{b = 0\}$
		$3: assert(i,b) = \{i \geq 0, b = 1\}$
		$4: assert(i,b) = \{i < 0, b = 0\}$

Figure 3.4.: Program 10: Different interpretations of program assertions

its parameters $param(T) = (\boldsymbol{a} : \boldsymbol{b})$. A *position* π is a sequence of numbers of length $|\pi|$ that specify a sub-tree $T_{|\pi}$ of an evaluation T. The position can be empty, denoted ϵ, or have the form $\pi = k \cdot \pi'$ where k is a number and π' another position. We have that $T_{|\epsilon} = T$ and $T_{|k\cdot\pi} = (T_k)_{|\pi}$.

The sets of evaluations can be restricted with respect to an entry set to consider only the evaluations that can take place starting from one of the entries.

Definition 3.6 (Evaluations from Entries)**.** Let CRS be a cost relation system, let $ES \subseteq CRS$ be an entry set and let $C \in CRS$ be a cost relation. The set of evaluations of C from the entries ES is denoted:

$$[\![CRS_{ES}|C]\!] := \left\{ \; T \in [\![CRS|C]\!] \;\middle|\; \text{there is a } T' \in [\![CRS|E]\!] \text{ for } E \in ES \text{ such that } T \preceq T' \; \right\}$$

In general $[\![CRS_{ES}|C]\!] \subseteq [\![CRS|C]\!]$ and if $C \in ES$, we have that $[\![CRS_{ES}|C]\!] = [\![CRS|C]\!]$. This definition can also be extended for finite $[\![CRS_{ES}|C]\!]_c$ or infinite evaluations $[\![CRS_{ES}|C]\!]_\omega$ and for evaluations with specific parameters $[\![CRS_{ES}|C(\boldsymbol{a} : \boldsymbol{b})]\!]$.

3.4 Failed Evaluations vs Runtime Failure

Note that failed evaluations, that is, evaluations that cannot be completed because there is a constraint set φ that is incompatible with our variable assignment σ, are not included in any of the evaluation sets. This is because a failed evaluation does not necessarily represent an actual execution of the original program.

If the execution of a program can fail, for instance, throwing an exception or having a runtime error and we want to consider this behavior, it should be encoded in the cost relations explicitly. This has the advantage that the user can decide which errors to consider and which to ignore in the cost relation extraction phase.

Example 3.7. Figure 3.4 contains two alternative translations of Program 10 into cost relations. In the first translation, only the executions of the program that satisfy the assertion `assert(i>=0)` are considered. That means that $[\![f(-1)]\!]_c = \emptyset$. This translation enables the user to reason about programs annotated with preconditions, postconditions or to analyze the cost of a program such that certain properties are maintained.

In the second translation, the possibility that the assertion fails is explicitly considered. If we obtain a bound of this translation, it is also valid for executions that finish abruptly because the assertion is not satisfied. In particular, we can have an evaluation

$$t(2(-1), [T_g, t(4(-1,0), [\,])]) \in [\![f(-1)]\!]_c$$

where T_g is an evaluation of $[\![g(-1)]\!]$.

Even with the first translation, the set of infinite evaluations of $[\![f(-1)]\!]_\omega$ might not be empty if g is non-terminating, because in such a case the condition $i \geq 0$ is not reached.

3.5 Costs

Now that the evaluations of a cost relation have been defined, we define their costs. The cost of a complete evaluation is defined recursively over the tree structure:

Definition 3.8 (Evaluation Cost)**.** Let T be a complete evaluation, its cost is defined as:

$$Cost(T) = \begin{cases} \sum_{i=1}^{n} Cost(T_i) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \\ T & \text{if } T \in \mathbb{Q} \end{cases}$$

Upper and lower bounds of complete evaluations are defined as follows:

Definition 3.9 (Net Evaluation Bound)**.** Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ be a complete evaluation. The function $f(\boldsymbol{x}) : \mathbb{Q}^n \to \mathbb{Q} \cup \{\omega, -\omega\}$ is an *net upper bound* (respectively a *net lower bound*) of T if $f(\boldsymbol{a}) \geq Cost(T)$ (respectively $f(\boldsymbol{a}) \leq Cost(T)$).

This definition of cost corresponds to the notion of *net cost* which considers the total amount of resources consumed or released during a complete execution of a program. This definition is problematic for infinite evaluations. In an infinite evaluation, the cost might oscillate indefinitely and the function *Cost* might not be well defined.

Example 3.10. This is the case for the infinite evaluations of Program 9. Resources are allocated and deallocated indefinitely (see infinite evaluation in Figure 3.3) and the sum does not converge to any specific amount.

An alternative notion of cost is the *peak cost* (which in case of upper bounds is also referred to as *high water mark*) which also considers the amount of resources consumed at any intermediate state of the execution. The peak cost bound of an evaluation can be defined in terms of its partial evaluations, that is, evaluations that are truncated at some position.

Definition 3.11 (Partial Evaluation)**.** Let $T \in [\![CRS|C]\!]$ be an evaluation and π a valid position of T. A partial evaluation is defined as:

$$\downarrow_\pi (T) = \begin{cases} t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, T_2, \cdots, \downarrow_{\pi'} (T_k)]) & \text{if } \pi = k \cdot \pi' \ \wedge \ T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \\ t(\bot(\boldsymbol{a} : \boldsymbol{b}), []) & \text{if } \pi = \epsilon \ \wedge \ T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \\ 0 & \text{if } \pi = \epsilon \ \wedge \ T \in \mathbb{Q} \end{cases}$$

Note that $Cost(t(\bot(\boldsymbol{a} : \boldsymbol{b}), [])) = 0$ and thus the distinction between the second and third case is not necessary. However, keeping the parameters of the original evaluation ($\boldsymbol{a}$ and $\boldsymbol{b}$) and marking the evaluation with a special symbol $\bot$ is useful to reason about partial evaluations in the bound computation procedure.

Definition 3.12 (Peak Evaluation Bound)**.** Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ be an evaluation. The function $f(\boldsymbol{x}) : \mathbb{Q}^n \to \mathbb{Q} \cup \{\omega, -\omega\}$ is a *peak upper bound* (respectively a *peak lower bound*) of T if $f(\boldsymbol{a}) \geq Cost(\downarrow_\pi (T))$ (respectively $f(\boldsymbol{a}) \leq Cost(\downarrow_\pi (T))$) for every valid position π. Moreover, if T is finite, f has to be a net evaluation bound of T as well.

A function f is a cost relation bound if it is a bound of all its evaluations. This definition is applicable for both peak and net bounds.

Definition 3.13 (Cost Relation Bound)**.** Let *CRS* be a cost relation system and $C \in CRS$ a cost relation. The function $f(\boldsymbol{x}) : \mathbb{Q}^n \to \mathbb{Q} \cup \{\omega, -\omega\}$ is a peak/net upper/lower bound of C in *CRS* if f is a peak/net upper/lower bound of every $T \in [\![CRS|C]\!]$.

Note that every evaluation T and every cost relation C has a trivial upper bound ω and a trivial lower bound $-\omega$. This definition can be easily extended for bounds of cost relations with respect to some entry set ES. The function f is a bound of C in CRS with respect to ES if it is a bound of every $T \in [\![CRS_{ES}|C]\!]$.

Example 3.14. The cost of the evaluation of Program 3 in Figure 3.1 is 8 which corresponds to the sum of the cost of its nodes. As mentioned in the previous chapter, the function $\|l\| + \|l + s\|$ is a sound and precise net upper bound of CR *amortized* (Program 3). This function is also a valid peak upper bound.

The function $\|l\|$ is a net lower bound. This is because *popSome* might not be called at all in some evaluations, but l has to be completely consumed in the main loop.

Example 3.15. As mentioned in Example 3.10, the net costs of infinite evaluations of Program 9 are not well defined. However, if we consider only the finite evaluations (with $n \geq 0$), their net cost is 0 because all the allocated resources are eventually deallocated.

The peak upper bound of the finite evaluations is $n + 1$. This is because in each iteration (application of CE 2), the program allocates 2 resource units and deallocates only 1, that is, one extra resource unit remains allocated after each recursive call. The difference is accumulated until the base case is reached. The peak cost takes place at that point after the last allocation. Conversely, the peak upper bound of the infinite evaluations is 1 because at each iteration (application of CE 3) one resource unit is allocated and then deallocated so no resources are accumulated.

The techniques presented in this work focus on obtaining net bounds. Obtaining precise peak bounds is a challenging problem because, in principle, all possible locations where evaluations can stop have to be considered (all possible partial evaluations). This falls out of the scope of this work. However, if the resource being measured is cumulative, that is, the cost equations do not contain negative cost annotations, a net cost upper bound is also a peak upper bound [ABG12] and the lower peak bound is trivially 0 (which corresponds to the partial evaluation $\downarrow_{\epsilon}(T)$). That means that the upper bounds obtained by this approach are also valid peak upper bounds. Moreover, the CoFloCo approach can also obtain peak upper bounds for non-terminating cost relations if the CRS has only cumulative cost (see Chapter 6).

Despite not obtaining peak cost bounds for cost relation systems with negative costs, the preprocessing phase (Chapter 4) and the refinement phase (Chapter 5) are still valid and applicable to arbitrary cost relation systems (including negative costs), only the bound computation procedure needs to be adapted. It is also worth mentioning that there exist approaches based on cost relations that can obtain peak upper bounds without resorting to negative cost annotations [ACRD15, AGGZ13]. These works generate cost relation systems with only positive cost annotations that already represent the peak cost of the program. Consequently, they can directly benefit from the results presented here.

3.6 Cost Preserving Transformations

Finally, an important part of the approach consists of incrementally transforming and refining a cost relation system CRS into another CRS'. These transformations have to guarantee that a bound obtained in the new CRS' is also valid for CRS. It is also desirable (but not necessary) that the transformation does not lose precision, that is, if CRS has a bound, then CRS' has also the same bound.

Definition 3.16 (Sound and Precise Transformation)**.** Let CRS be a cost relation system. A transformation that generates a CRS' is *sound* if any bound of C in CRS' is also a bound of C in CRS. A transformation is also *precise* if any bound of C in CRS is also a bound of C in CRS'.

If a transformation substitutes a C by a C', then in order to be sound, a bound of C' in CRS' has to be a valid bound of C in CRS. A transformation can also be sound and precise with respect to an entry set if the bounds are maintained for the evaluations with respect to the entry set.

4 Preprocessing

The first step of the analysis of a cost relation system (CRS) is to reduce all indirect recursion to direct recursion. A similar preprocessing step is presented in [AAGP11] and is common to all the cost relation analyses [AAGP11, AGM13, ABAG13]. In this chapter, the preprocessing step is reformulated according to this thesis's notation and definitions and it is proven sound with respect to the semantics given in the previous chapter. This is important because it guarantees that the preprocessing step is also sound for non-terminating evaluations. In addition to that, this chapter provides additional transformations of CRS that can be used to overcome the limitations of the preprocessing presented in [AAGP11] and to simplify the cost relation system.

4.1 Call-graphs, Strongly Connected Components and Feedback Sets

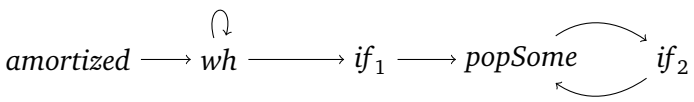
Let us first introduce some notation. A cost equation $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$ *calls* a cost relation D at position i if $b_i = D(\boldsymbol{x_i} : \boldsymbol{y_i})$. This is denoted $c \rightarrow_i D$. Then, a cost relation C *calls* another cost relation D, written $C \rightarrow D$, if there is a CE $c \in C$ such that $c \rightarrow_i D$.

This relation between cost relations induces a call-graph, i.e. a directed graph where the vertices are cost relation symbols and the edges represent calls. The strongly connected components (SCCs) of the call-graph are computed. The SCCs can be sorted in topological order $\langle S_1, \cdots, S_n \rangle$ such that any cost relation $C \in S_i$ can only contain calls to $C' \in S_j$ for $j \geq i$. Each SCC represents a set of mutually recursive cost relations or a single non-recursive cost relation. The objective is to transform each SCC with more than one cost relation into a single cost relation.

Given a SCC S, the preprocessing computes its minimal feedback vertex set $MFVS(S)$ which is the smallest set of vertices G such that the call-graph with vertices $S \setminus G$ is acyclic. This can be computed efficiently ($\mathcal{O}(n)$ where n is the number of cost relations in S) if the call graph is reducible [Sha79]. This is usually the case for CRS generated from programs. Otherwise, it can be checked whether there exists a feedback vertex set of size 1 in $\mathcal{O}(n^2)$ [AAGP11]. The problem is NP complete for general graphs but there is plenty of research for certain graph classes [FPR09].

If the $MFVS(S)$ of a SCC S contains only one element, unfolding (which corresponds to inlining) can be applied to all the other cost relations $S \setminus MFVS(S)$ repeatedly to reduce the SCC into a single cost relation [YK97]. Consider that if a CR $C \notin MFVS(S)$, we know that C does not have a call to itself. If all the calls to C in all the cost equations in S are unfolded, at the end of this sequence of transformations no CR in S contains any call to C. Therefore, C does not belong to the SCC S anymore. Moreover, if C is not an entry ($C \notin ES$) nor it is reachable from any other SCC, it can be safely removed.

Example 4.1. Consider the cost relations from Program 3 (Figure 4.1). Its call-graph is:



The sequence of topologically sorted SCCs is $\langle \{amortized\}, \{wh\}, \{if_1\}, \{popSome, if_2\} \rangle$. The SCCs $\{wh\}$ and $\{popSome, if_2\}$ are recursive and only the latter has indirect recursion. The set $\{popSome\}$ is a minimal feedback vertex set (in fact $\{if_2\}$ is also a minimal feedback vertex set).

Cost relations of Program 3

$$
\begin{array}{lcl}
1\colon amortized(l,s) & = & wh(l,s:lo,so) \\
2\colon wh(l,s:lo,so) & = & \{l=0, l=lo, s=so\}, \\
3\colon wh(l,s:lo,so) & = & \{l>0, s'=s+1, l'=l-1\},\ 1,\ if_1(l',s':l'^2,s'^2),\ wh(l'^2,s'^2:lo,so) \\
4\colon if_1(l,s:lo,so) & = & \{l=lo\},\ popSome(s:so) \\
5\colon if_1(l,s:lo,so) & = & \{l=lo, s=so\} \\
6\colon popSome(s:so) & = & if_2(s:so) \\
7\colon if_2(s:so) & = & \{s=so\} \\
8\colon if_2(s:so) & = & \{s>0, tmp=s-1\},\ 1,\ popSome(tmp:so)
\end{array}
$$

Figure 4.1.: Cost relations of Program 3 with output variables

4.2 Unfolding Cost Relations

Unfolding a CR D in a cost equation $c \in C$ is equivalent to performing one evaluation step. It accounts for substituting a call to D in c by its body (its right-hand side). If D contains several cost equations, the unfolding of D in c generates one version of c for each CE $d \in D$. The formal definition is as follows:

Definition 4.2 (Unfold)**.** Let CRS be a cost relation system with non-empty cost relations $C, D \subseteq CRS$ and let $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$ such that there is a call $b_j = D(\boldsymbol{x}_j:\boldsymbol{y}_j)$ (we have $c \to_j D$). The unfolded set of cost equations is:

$$U(c,D,j) := \left\{\ c.d_j\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{j-1}, body(d)[\boldsymbol{x}\boldsymbol{y}/\boldsymbol{x}_j\boldsymbol{y}_j], \varphi_j, \cdots, b_n, \varphi_n \ \middle|\ d \in D\ \right\}$$

The Unfold transformation defines a new cost relation system $CRS' = CRS \setminus \{c\} \cup U(c,D,j)$

This definition assumes that for each $d \in D$ the variables on the head are $head(d) = D(\boldsymbol{x}:\boldsymbol{y})$ and c and d do not have other variables in common $vars(body(d)) \cap vars(body(c)) \subseteq vars(\boldsymbol{x}\boldsymbol{y})$. The variables of each CE d can be renamed to guarantee these assumptions.

Theorem 4.3. *The Unfold transformation is sound and precise.*

If a CE c has several calls to D the unfolding transformation can be applied also multiple times, once per call.

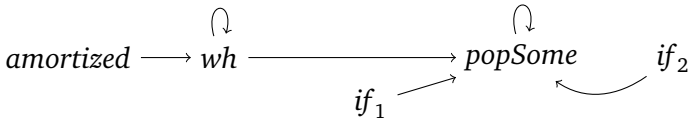
Example 4.4. In Example 4.1 we saw that Program 3 has a SCC $\{popSome, if_2\}$ with indirect recursion and a minimal feedback vertex set $\{popSome\}$. Therefore, we can unfold the call to if_2 is CE 6 obtaining the CEs:

$$
\begin{array}{lll}
6.7_1\colon & popSome(s:so) = & \{s=so\} \\
6.8_1\colon & popSome(s:so) = & \{s>0, tmp=s-1\},\ 1,\ popSome(tmp:so)
\end{array}
$$

CE 6.7_1 results from the combination of CE 6 and 7 and CE 6.8_1 from CE 6 and 8. Once performed this transformation, CR *popSome* only has direct recursion. Unfolding can also be applied to cost equations and cost relations in different SCCs. This is not useful for obtaining direct recursion, but it can help to simplify the CRS. For instance, we can unfold the call to if_1 in CE 3 obtaining:

$$
\begin{array}{lll}
3.4_2\colon wh(l,s:lo,so) = & \{l>0, s'=s+1, l'=l-1\},\ 1,\ \{l'=l'^2\},\ popSome(s':s'^2), \\
& wh(l'^2,s'^2:lo,so) \\
3.5_2\colon wh(l,s:lo,so) = & \{l>0, s'=s+1, l'=l-1\},\ 1,\ \{l'=l'^2, s'=s'^2\},\ wh(l'^2,s'^2:lo,so)
\end{array}
$$

The resulting call-graph is the following:



If we consider CR *amortized* to be the entry point, CRs if_1 and if_2 are not longer reachable and we can discard them. The sequence of SCCs, which now is a sequence of CRs (all SCCs have a single element) is $\langle amortized, wh, popSome\rangle$.

4.3 Dealing with Non-unitary Feedback Sets

One of the limitations of the approach presented in [AAGP11] is that whenever a SCC does not have a feedback vertex set of size one, the analysis fails.

Example 4.5. Consider the following *CRS* example:

$$\begin{array}{ll} 1\colon f(x) = \{x = 0\},\ 0 & 2\colon f(x) = \{x \geq 1, x' = x-1\},\ g(x'),\ f(x') \\ 3\colon g(x) = \{x = 0\},\ 0 & 4\colon g(x) = \{x \geq 1, x' = x-1\},\ f(x'),\ g(x') \end{array}$$

It contains a single *SCC* with a *MFVS* $\{f, g\}$. We need a transformation to reduce the size of the *MFVS* to 1 in order to be able to obtain direct recursion using unfolding.

If the feedback vertex set of a SCC contains several cost relations, a simple solution is to merge them into a single cost relation that simulates all of them.

Definition 4.6 (Cost Relation Merging)**.** Let $C_1, C_2 \subseteq CRS$ and let $\boldsymbol{x_1 y_1}$ and $\boldsymbol{x_2 y_2}$ be in input and output variables of C_1 and C_2 respectively. The merged cost relation C_m has the maximum number of input and output variables. If $|\boldsymbol{x_1}| \geq |\boldsymbol{x_2}|$, the input variables of C_m are defined $\boldsymbol{x_m} := \boldsymbol{x_1} = \boldsymbol{x_2 z}$ where $\boldsymbol{z}$ a sequence of fresh variables used to complete $\boldsymbol{x_2}$ to the right length. Conversely, if $|\boldsymbol{x_1}| < |\boldsymbol{x_2}|$, the input variables of C_m are defined $\boldsymbol{x_m} = \boldsymbol{x_1 z} = \boldsymbol{x_2}$. The output variables $\boldsymbol{y_m}$ are defined the same way. Then, for all cost equations in the *CRS*, the calls to C_1 and C_2 and the heads of the equations in C_1 and C_2 are substituted by terms $C_m(\boldsymbol{x_m} : \boldsymbol{y_m})$. The new cost relation system is:

$$CRS' = CRS[C_1(\boldsymbol{x_1} : \boldsymbol{y_1})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m}),\quad C_2(\boldsymbol{x_2} : \boldsymbol{y_2})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m})]$$

Theorem 4.7. *Cost relation merging is sound.*

Example 4.8. The merged cost relation *fg* from the cost relations f and g from Example 4.5 is:

$$\begin{array}{ll} 1\colon fg(x) = \{x = 0\},\ 0 & 2\colon fg(x) = \{x \geq 1, x' = x-1\},\ fg(x'),\ fg(x') \\ 3\colon fg(x) = \{x = 0\},\ 0 & 4\colon fg(x) = \{x \geq 1, x' = x-1\},\ fg(x'),\ fg(x') \end{array}$$

In this particular case, CEs 1 and 3 are equal and 2 and 4 are also equal so they can be simplified.

Example 4.9. Consider an example of merging cost relations with different number of variables (all input variables):

$$\begin{array}{ll} 1\colon f(x,y,z) = \{x = 0\} & 4\colon g(y,z) = \{y \leq 0, y' = 0, z' = z-1\},\ f(z',y',z') \\ 2\colon f(x,y,z) = \{x > 0, x' = x-1\},\ 1,\ f(x',y,z) & 5\colon g(y,z) = \{y > 0, y' = y-1\},\ g(y',z) \\ 3\colon f(x,y,z) = \{x = z, y > 0\},\ 1,\ g(y,z) & \end{array}$$

Although this example is different from Example 4.5, its call-graph is identical. It has a single SCC $\{f, g\}$ which is itself the *MFVS*. The merged cost relations are:

$1: fg(x,y,z) = \{x = 0\}$
$2: fg(x,y,z) = \{x > 0, x' = x - 1\}, 1, fg(x',y,z)$
$3: fg(x,y,z) = \{x = z, y > 0\}, 1, fg(y,z,r)$
$4: fg(y,z,r) = \{y \leq 0, y' = 0, z' = z - 1\}, fg(z',y',z')$
$5: fg(y,z,r) = \{y > 0, y' = y - 1\}, fg(y',z,r')$

Note how the cost equations generated from g have and additional parameter r which is not used and it does not appear in any constraint set. This ensures that all the resulting equations have the same number of arguments.

Cost relation merging can be generalized to merge any number of cost relations. In addition, it can be made more precise if an extra argument fl (a flag) is added to the new cost relation. The value of fl in each CE encodes from which cost relation it originated.

Example 4.10. The CRS of Example 4.9 can be transformed as follows:

$1: fg(fl,x,y,z) = \{fl = 1, x = 0\}$
$2: fg(fl,x,y,z) = \{fl = 1, x > 0, x' = x - 1, fl' = 1\}, 1, fg(fl',x',y,z)$
$3: fg(fl,x,y,z) = \{fl = 1, x = z, y > 0, fl' = 2\}, 1, fg(fl',y,z,r)$
$4: fg(fl,y,z,r) = \{fl = 2, y \leq 0, z' = z - 1, y' = 0, fl' = 1\}, fg(fl',z',y',z')$
$5: fg(fl,y,z,r) = \{fl = 2, y > 0, y' = y - 1, fl' = 2\}, fg(fl',y',z,r')$

In this transformed CRS all the CEs and calls have an extra argument that is 1 or 2 to indicate whether they originated from CR f or g. The calls to fg also contain the parameter fl' that indicates which CR was originally called.

4.4 Simplifying Transformations

The following transformations are used to simplify the cost relation systems both during the preprocessing and later in the analysis.

Definition 4.11 (Unfeasible Constraints Simplification)**.** Let c: $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$. If a subsequence of the constraint sets is unsatisfiable $\bigwedge_{0 \leq i \leq j} \varphi_i = \bot$, there cannot be any evaluation that reaches φ_j and the later part of the cost relation can be removed. The new CE is c': $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_j, \bot$ and the transformation generates $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Theorem 4.12. *Unfeasible Constraints Simplification is sound and precise.*

Note that the cost equation cannot be discarded altogether because some references in $b_1, \cdots, b_j$ might be non-terminating and thus CE c might still take part in infinite evaluations. In the special case where φ_0 in unsatisfiable, the cost equation can be safely discarded.

If a cost equation contains a call that is always terminating or always non-terminating, additional simplifications can be applied.

Definition 4.13 (Unreachable Code Elimination)**.** Let c: $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ such that b_j is guaranteed to *not terminate*. Then, all the elements after b_j can be eliminated from c because they are never reached. The new CE is c': $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_j$ and the transformation generates $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Theorem 4.14. *Unreachable Code Elimination is sound and precise.*

Definition 4.15 (Constraint Set Compression)**.** Let c: $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{i-1}, b_i, \varphi_i, \cdots$ such that b_i is terminating, then the constraint set φ_i can be moved before b_i. The new CE is c': $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{i-1} \wedge \varphi_i, b_i, \cdots$ and the transformation is $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Algorithm 1 Pre-processing of a cost relation system

```
1:  function PRE-PROCESS(CRS, ES)
2:      SCCs := DetectSCCs(CRS)
3:      for each S ∈ SCCs ∧ |S| > 1 do
4:          MFVS := feedbackSet(S)
5:          if |MFVS| > 1 then mergeCRs(MFVS, CRS)
6:          for each C ∈ S ∧ C ∉ MFVS do
7:              for each c ∈ CRS such that c →_j C do
8:                  CRS = CRS \ {c}
9:                  for each c' ∈ U(c, C, j) do
10:                     CRS = CRS ∪ {simplifyCE(c')}
11:                     CRS = checkSubsumtion(CRS, c')
12:             if C ⊈ ES then CRS = CRS \ C
13:     return CRS
```

Theorem 4.16. *Constraint set compression is sound and precise.*

This transformation can be combined with unfeasible constraints simplification to discard additional cost equations. This transformation is intensively used during the refinement (Chapter 5) where cost relations are proved terminating. At this point though, sufficient conditions can be considered. For instance, if b_i is a linear expression, it is always terminating. Also, if $b_i = C_i(\boldsymbol{x}_i : \boldsymbol{y}_i)$ and no recursive CR can be reached from C_i in the call-graph, then C_i must be terminating.

Definition 4.17 (Cost Equation Subsumption)**.** Let $c, c' \in CRS$ be two cost equations of the form $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_i, \cdots, b_n, \varphi_n$ and $c'\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi'_i, \cdots, b_n, \varphi_n$ (they are equal except in φ_i) such that $\varphi_i \Rightarrow \varphi'_i$. Then, CE c subsumes c' and c can be removed from the cost relation system. The transformation generates $CRS' = CRS \setminus \{c\}$.

Theorem 4.18. *Cost equation subsumption is sound and precise.*

Checking subsumption can be costly, but sufficient conditions can be checked instead. For example, if $\varphi_1 \subseteq \varphi_2$, we know that $\varphi_2 \Rightarrow \varphi_1$.

4.5 Algorithm

The preprocessing step is implemented in Algorithm 1. The algorithm receives a cost relation system *CRS* and a set of entries *ES* and returns a transformed CRS that does not contain indirect (mutual) recursion.

The algorithm starts by detecting the strongly connected components of the CRS (Line 2). For each SCC that contains indirect recursion i.e. $|S| > 1$, it computes the minimal feedback vertex set *MFVS* (Line 4). If *MFVS* contains more than one cost relation, it merges them (Line 5). Then, the algorithm unfolds all the cost relations not contained in *MFVS*. Each of those cost relations C is unfolded in all the cost equations c of the complete CRS that call it and for all the positions j where it is called (Line 7). For each newly generated cost equation c', the algorithm applies the function *simplifyCE* before adding it to the CRS (Line 10). The function *simplifyCE* applies constraint set compression (Definition 4.15), unreachable code elimination (Definition 4.13), and unfeasible constraints simplification (Definition 4.11) in that order. Once c' has been added to the CRS, the function *checkSubsumption* (Line 11) checks if it subsumes or if it is subsumed by any other cost equation in the CRS and eliminates the corresponding CEs (Definition 4.17). Finally, once the unfolding of a CR C is finished, if C is not an entry, it is no longer reachable from the entries and it is discarded (Line 12).

In general, applying unfolding repeatedly to the cost relations of a SCC can result in an exponential number of cost equations. The algorithm tries to reduce the number of generated cost relations by applying *simplifyCE* and checking for subsumption at every step. Several CEs that are initially different might become equivalent after simplifying them, and reduced to a single CE when checking for subsumption. In addition to that, the order in which the unfolding of CRs is done (Line 6) can also affect the efficiency of the preprocessing. The algorithm selects this order heuristically. From the cost relations $C \in S$ that have not been unfolded yet, it selects the one with the smallest number of cost equations to be unfolded next.

4.6 Proofs

Lemma 4.19. *A sufficient condition for a transformation from CRS to CRS′ to be sound is that there is a total function* mp *that transforms an evaluation into another such that:*

(a) $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ *implies* $\text{mp}(T) \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ *for every* $C \subseteq CRS$ *and for every* $\boldsymbol{a} : \boldsymbol{b}$.

(b) If T *is finite, then* $Cost(T) = Cost(\text{mp}(T))$

(c) For every partial evaluation $\downarrow_\pi (T)$ *there is a corresponding* $\downarrow_{\pi'} (\text{mp}(T))$ *such that* $Cost(\downarrow_\pi (T)) = Cost(\downarrow_{\pi'} (\text{mp}(T)))$

Proof. Let f be a net upper bound of C in CRS'. Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$, because of point (a) there is a $\text{mp}(T) \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ such that $Cost(T) = Cost(\text{mp}(T)) \leq f(\boldsymbol{a})$ (because of point (b)). Consequently, f is a net upper bound of T for every $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ and therefore it is a net upper bound of C in CRS. The same reasoning applies for net lower bounds.

Let f be a peak upper bound of C in CRS'. Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]$, because or point (a) there is a $\text{mp}(T) \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]$. If for any $\downarrow_\pi (T)$ there is a $\downarrow_{\pi'} (\text{mp}(T))$ such that $Cost(\downarrow_\pi (T)) = Cost(\downarrow_\pi (\text{mp}(T))) \leq f(\boldsymbol{a})$ (because of point (c)), f is a peak upper bound of C in CRS. The same reasoning applies for net lower bounds. As a consequence, the transformation is sound. □

All the proofs in this section are based on Lemma 4.19. For many instances of mp, points (b) and (c) are immediate. For others, they can be proved by induction of the height of the tree and the length of the position π respectively. Point (a) can be proven by induction for finite evaluations and by co-induction for infinite evaluations. In both cases, the proof amounts to guaranteeing that if the conditions of the semantics are satisfied for a T, then they are also satisfied for $\text{mp}(T)$. To prove that a transformation from CRS to CRS' is precise, is equivalent to prove that the inverse transformation from CRS' to CRS is sound. In most cases, the soundness proof can simply be reversed to prove that a transformation is precise.

4.6.1 Proof of Theorem 4.3

Given an Unfold transformation with $U(c, D, j)$, the function mp is defined as follows:

$$\text{mp}(T) = \begin{cases} t(c'(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)]) & \text{if } T = t(c'(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \wedge c' \neq c \\ t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)]) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \\ & \quad \wedge n < j \wedge d \in D \\ t\Big(c.d_j(\boldsymbol{a} : \boldsymbol{b}), \begin{bmatrix} \text{mp}(T_1), \cdots, \text{mp}(T_{j-1}), \\ \text{mp}(T'_1), \cdots, \text{mp}(T'_m), \\ \text{mp}(T_{j+1}), \cdots, \text{mp}(T_n) \end{bmatrix}\Big) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_j, \cdots, T_n]) \\ & \quad \wedge T_j = t(d(\boldsymbol{a}_j : \boldsymbol{b}_j), [T'_1, \cdots, T'_m]) \\ T & \text{if } T \in \mathbb{Q} \end{cases}$$

This definition mimics the behavior of the transformation. The first case corresponds to an evaluation of a c that has not been modified. The second case corresponds to an evaluation of c that diverges before reaching the unfolded call to CR D. In that case, the function mp chooses any $d \in D$. The third case corresponds to an evaluation c such that the unfolded call is reached. Finally, the last case is the base case for linear expressions.

Proof of Point (a) for Finite evaluations

Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ for any C with $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$, we can assume all the conditions of the semantics are satisfied for T:

1. $e : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$
3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(e), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{n} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{n} T_i \in [\![CRS|b_i\sigma]\!]_c$

and we have to prove that the corresponding evaluation $\mathrm{mp}(T) \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$. This can be proven by induction on the height of the tree. In the base case, every T_i in T is a linear expression evaluation and $e \neq c$ (because c has a call to D). Therefore, $\mathrm{mp}(T) = T \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$.

For the inductive step, we distinguish two cases that correspond to the first and third cases of mp (the second case of mp is not possible for finite evaluations):

- If $e \neq c$, we have $\mathrm{mp}(T) = t(e(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_n)])$. Condition (1) and (2) of the semantics are satisfied with the same CE $e \in CRS'$. For condition (3), the assignment σ used in T is also valid for $\mathrm{mp}(T)$. For condition (4), we can apply the induction hypothesis. If $T_i \in [\![CRS|b_i\sigma]\!]_c$ then $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $1 \leq i \leq n$.

- If $e = c$ and $T_j = t(d(\boldsymbol{a'} : \boldsymbol{b'}), [T'_1, \cdots, T'_m]) \in [\![CRS|C(\boldsymbol{a'} : \boldsymbol{b'})]\!]_c$, we can also assume the conditions of the semantics are satisfied for T_j with a σ'. We have

  $$\mathrm{mp}(T) = t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_{j-1}), \mathrm{mp}(T'_1), \cdots, \mathrm{mp}(T'_m), \mathrm{mp}(T_{j+1}), \cdots, \mathrm{mp}(T_n)])$$

  and conditions (1) and (2) are satisfied with the unfolded CE

  $$c.d_j : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{j-1}, body(d)[\boldsymbol{xy}/\boldsymbol{x_j y_j}], \varphi_j, b_{j+1}, \cdots, b_n, \varphi_n \in CRS'$$

  where $body(d) = \varphi'_0, b'_1, \cdots, b'_m \varphi'_m$. Let σ be the assignment used for T and σ' the one used for T_j. We can have σ'' that extends σ and $[\boldsymbol{x_j y_j}/\boldsymbol{xy}]\sigma'$ such that it satisfies $\varphi_0 \wedge \cdots \wedge \varphi_n$ and $(\varphi'_0 \wedge \cdots \wedge \varphi'_m)[\boldsymbol{xy}/\boldsymbol{x_j y_j}]$ and such that $b_i\sigma = b_i\sigma''$ and $b'_i\sigma' = (b'_i[\boldsymbol{xy}/\boldsymbol{x_j y_j}])\sigma''$. This is because σ and $[\boldsymbol{x_j y_j}/\boldsymbol{xy}]\sigma'$ only have the variables $\boldsymbol{x_j y_j}$ in common and they coincide for those variables $\boldsymbol{x_j y_j}\sigma = (\boldsymbol{x_j y_j}[\boldsymbol{x_j y_j}/\boldsymbol{xy}])\sigma' = \boldsymbol{a'b'}$. Consequently, condition (3) is satisfied with σ''. Condition (4) holds for $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma'']\!]_c$ for $1 \leq i \leq n$ except $i = j$ and $\mathrm{mp}(T'_i) \in [\![CRS'|(b'_i[\boldsymbol{xy}/\boldsymbol{x_j y_j}])\sigma'']\!]_c$ for $1 \leq i \leq m$ by the induction hypothesis.

Proof of Point (a) for Infinite evaluations

For infinite evaluations, we prove that the sets $\mathrm{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega)$ (we lift mp to sets of evaluations) are a fixpoint of the infinite evaluations of in CRS'. The infinite evaluations correspond to the greatest fixpoint that satisfies the condition of the semantics (note that infinite evaluations are defined coinductively) so if the sets $\mathrm{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega)$ are a fixpoint, then $\mathrm{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega) \subseteq [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ and point (a) holds.

Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ with the form $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$. We assume the conditions of the semantics hold for T:

1. $e : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $\exists m \leq n$ such that $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$
3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(e), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{m-1} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{m-1} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_m \in [\![CRS|b_m\sigma]\!]_\omega$

and we distinguish cases:

- If $e \neq c$, we have the first case of mp: $\mathrm{mp}(T) = t(e(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_m)])$. Conditions (1) and (2) are satisfied with $e \in CRS'$. For condition (3), the assignment σ used in T is valid for $\mathrm{mp}(T)$. For condition (4), $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $i < m$ has been proved above and $\mathrm{mp}(T_n) \in [\![CRS'|b_n\sigma]\!]_\omega$ holds by co-induction hypothesis.

- If $e = c$ and $m < j$, we have the second case of mp and it corresponds to an evaluation that diverges before reaching the call to D. In this case, we have $\mathrm{mp}(T) = t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_m)])$ and there is a $d \in D$ (D is not empty) in CRS and conditions (1) and (2) are satisfied with

  $$c.d_j : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{j-1}, body(d)[\boldsymbol{xy}/\boldsymbol{x_j y_j}], \varphi_j, b_{j+1}, \cdots, b_n, \varphi_n \in CRS'$$

  For condition (3), we can extend the σ used for T arbitrarily to cover all the variables in $c.d_j$. The extended assignment σ' also satisfies the constraints φ_i and yields the same terms $b_i\sigma' = b_i\sigma$ for every $i \leq n$ because $n < j$. For condition (4) $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma']\!]_c$ for $i < n$ has been proved above and $\mathrm{mp}(T_n) \in [\![CRS'|b_n\sigma']\!]_\omega$ holds by co-induction hypothesis.

- If $e = c$ and $n \geq j$, then the sub-evaluation T_j has the form $T_j = t(d(\boldsymbol{a_j} : \boldsymbol{b_j}), [T'_1, \cdots, T'_p])$ and we can also assume the conditions of the semantics hold for CE d and an assignment σ'. There are also two possibilities (I) $T_j \in [\![CRS|D(\boldsymbol{a_j} : \boldsymbol{b_j})]\!]_c$ or (II) $T_j \in [\![CRS|D(\boldsymbol{a_j} : \boldsymbol{b_j})]\!]_\omega$ (in which case $j = m$). The corresponding $\mathrm{mp}(T)$ are:

  (I) $t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_{j-1}), \mathrm{mp}(T'_1), \cdots, \mathrm{mp}(T'_p), \mathrm{mp}(T_{j+1}), \cdots, \mathrm{mp}(T_m)])$
  (II) $t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_{j-1}), \mathrm{mp}(T'_1), \cdots, \mathrm{mp}(T'_p)])$

  In both cases, conditions (1) and (2) are satisfied with

  $$c.d_j : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{j-1}, body(d)[\boldsymbol{xy}/\boldsymbol{x_j y_j}], \varphi_j, b_{j+1}, \cdots, b_n, \varphi_n \in CRS'$$

  where $body(d) = \varphi'_0, b'_1, \cdots, b'_k\varphi'_k$, $p \leq k$, and $m \leq n$. Let σ the assignment used for T and σ' the one used for T_j. In this case we can also extend σ and $[\boldsymbol{x_j y_j}/\boldsymbol{xy}]\sigma'$ to σ'' as in the proof for finite evaluations and σ'' satisfies condition (3). For condition (4):

  (I) $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma'']\!]_c$ for $1 \leq i < m$ except $i = j$ and $\mathrm{mp}(T'_i) \in [\![CRS'|(b'_i[\boldsymbol{xy}/\boldsymbol{x_j y_j}])\sigma'']\!]_c$ for $1 \leq i \leq p$ have been proved above; and $\mathrm{mp}(T_m) \in [\![CRS'|b_i\sigma'']\!]_\omega$ by co-induction hypothesis.

  (II) $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma'']\!]_c$ for $1 \leq i < j$ and $\mathrm{mp}(T'_i) \in [\![CRS'|(b'_i[\boldsymbol{xy}/\boldsymbol{x_j y_j}])\sigma'']\!]_c$ for $1 \leq i < p$ have been proved above; and $\mathrm{mp}(T'_p) \in [\![CRS'|(b'_p[\boldsymbol{xy}/\boldsymbol{x_j y_j}])\sigma'']\!]_\omega$ by co-induction hypothesis.

Proof of Point (b)

We prove that $Cost(T) = Cost(\text{mp}(T))$ for every finite T by induction on the height of T. In the base case, T is a linear expression evaluation $T \in \mathbb{Q}$ and $Cost(\text{mp}(T)) = Cost(T) = T$. In the inductive case, we distinguish cases according to the definition of mp (only the first and third cases are applicable):

- If $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ with $e \neq c$.

$$\begin{aligned} Cost(\text{mp}(T)) &=^{(1)} Cost(t(e(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)])) \\ &=^{(2)} \sum_{i=1}^{n} Cost(\text{mp}(T_i)) \\ &=^{(3)} \sum_{i=1}^{n} Cost(T_i) =^{(4)} Cost(T) \end{aligned}$$

  1. definition of mp
  2. and 4. definition of *Cost*
  3. induction hypothesis

- If $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_j, \cdots, T_n])$ with $T_j = t(d(\boldsymbol{a}_j : \boldsymbol{b}_j), [T'_1, \cdots, T'_m])$.

$$\begin{aligned} Cost(\text{mp}(T)) =^{(1)} \; & Cost\Big(t\big(c(\boldsymbol{a} : \boldsymbol{b}), \left[\begin{array}{l} \text{mp}(T_1), \cdots, \text{mp}(T_{j-1}), \text{mp}(T'_1), \cdots, \text{mp}(T'_m), \\ \text{mp}(T_{j+1}), \cdots, \text{mp}(T_n) \end{array}\right]\big)\Big) \\ =^{(2)} \; & \sum_{i=1}^{j-1} Cost(\text{mp}(T_i)) + \sum_{i=1}^{m} Cost(\text{mp}(T'_i)) + \sum_{i=j+1}^{n} Cost(\text{mp}(T_i)) \\ =^{(3)} \; & \sum_{i=1}^{j-1} Cost(T_i) + \sum_{i=1}^{m} Cost(T'_i) + \sum_{i=j+1}^{n} Cost(T_i) \\ =^{(4)} \; & \sum_{i=1}^{n} Cost(T_i) =^{(5)} Cost(T) \end{aligned}$$

  1. definition of mp
  2. definition of *Cost*
  3. induction hypothesis
  4. and 5. definition of *Cost* of T_j and T

Proof of Point (c)

We prove that for every π there is a π' such that $Cost(\downarrow_\pi (T)) = Cost(\downarrow_{\pi'} (\text{mp}(T)))$ by induction over the length of π. In the base case $\pi = \epsilon$, we take $\pi' = \epsilon$ so we have $Cost(\downarrow_\epsilon (T)) = 0 = Cost(\downarrow_\epsilon (\text{mp}(T)))$. For the inductive case, we consider a $\pi = k \cdot \pi_2$ and distinguish cases on the definition of mp (the fourth case is not applicable):

- Consider the first case $T = t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ with $e \neq c$.

$$\begin{aligned} Cost(\downarrow_\pi (T)) =^{(1)} \; & Cost(t(e(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, \downarrow_{\pi_2} (T_k)]) \\ =^{(2)} \; & \sum_{i=1}^{k-1} Cost(T_i) + Cost(\downarrow_{\pi_2} (T_k)) \\ =^{(3)} \; & \sum_{i=1}^{k-1} Cost(\text{mp}(T_i)) + Cost(\downarrow_{\pi_2} (T_k)) \\ =^{(4)} \; & \sum_{i=1}^{k-1} Cost(\text{mp}(T_i)) + Cost(\downarrow_{\pi'_2} (\text{mp}(T_k))) \\ =^{(5)} \; & Cost(t(e(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \downarrow_{\pi'_2} (\text{mp}(T_k))]) =^{(6)} \; Cost(\downarrow_{k \cdot \pi'_2} (\text{mp}(T))) \end{aligned}$$

1. and 6. definition of partial evaluation
2. and 5. definition of *Cost*
3. point (b) for finite evaluations
4. induction hypothesis

- The second case $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ with $n < j$ is equivalent to the previous one.
- In the third case $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_j, \cdots, T_n])$ with $T_j = t(d(\boldsymbol{a_j} : \boldsymbol{b_j}), [T'_1, \cdots, T'_m])$ and the transformed evaluation is $\text{mp}(T) = t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_{j-1}), \text{mp}(T'_1), \cdots, \text{mp}(T'_m), \text{mp}(T_{j+1}), \cdots, \text{mp}(T_n)])$. We distinguish cases depending on whether $k < j$, $k = j$ or $k > j$ (remember that we are considering a position $\pi = k \cdot \pi_2$).
  - The case where $k < j$ is equivalent to the previous ones.
  - If $k = j$ and $\pi_2 = \epsilon$, we take $\pi' = k$:

$$
\begin{aligned}
\mathit{Cost}(\downarrow_{\pi}(T)) &=^{(1)} \mathit{Cost}(t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, \downarrow_{\epsilon}(T_k)]) \\
&=^{(2)} \sum_{i=1}^{k-1} \mathit{Cost}(T_i) \\
&=^{(3)} \sum_{i=1}^{k-1} \mathit{Cost}(\text{mp}(T_i)) \\
&=^{(4)} \sum_{i=1}^{k-1} \mathit{Cost}(\text{mp}(T_i)) + \mathit{Cost}(\downarrow_{\epsilon}(\text{mp}(T_k))) \\
&=^{(5)} \mathit{Cost}(t(c.d_j(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \downarrow_{\epsilon}(\text{mp}(T_k))]) =^{(6)} \mathit{Cost}(\downarrow_k(\text{mp}(T)))
\end{aligned}
$$

1. and 6. definition of partial evaluation
2. and 5. definition of *Cost*
3. point (b) for finite evaluations
4. $\mathit{Cost}(\downarrow_{\epsilon}(T)) = 0$

If $\pi_2 = k_2 \cdot \pi_3$:

$$
\begin{aligned}
\mathit{Cost}(\downarrow_{\pi}(T)) &=^{(1)} \mathit{Cost}(t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, \downarrow_{\pi_2}(T_j)]) \\
&=^{(2)} \sum_{i=1}^{k-1} \mathit{Cost}(T_i) + \mathit{Cost}(\downarrow_{\pi_2}(t(d(\boldsymbol{a_j} : \boldsymbol{b_j}), [T'_1, \cdots, T'_m]))) \\
&=^{(3)} \sum_{i=1}^{k-1} \mathit{Cost}(T_i) + \mathit{Cost}(t(d(\boldsymbol{a_j} : \boldsymbol{b_j}), [T'_1, \cdots, \downarrow_{\pi_3}(T'_{k_2})])) \\
&=^{(4)} \sum_{i=1}^{k-1} \mathit{Cost}(T_i) + \sum_{i=1}^{k_2-1} \mathit{Cost}(T'_i) + \mathit{Cost}(\downarrow_{\pi_3}(T'_{k_2})) \\
&=^{(5)} \sum_{i=1}^{k-1} \mathit{Cost}(\text{mp}(T_i)) + \sum_{i=1}^{k_2-1} \mathit{Cost}(\text{mp}(T'_i)) + \mathit{Cost}(\downarrow_{\pi'_3}(\text{mp}(T'_{k_2}))) \\
&=^{(6)} \mathit{Cost}\left(t\left(c.d_j(\boldsymbol{a} : \boldsymbol{b}), \left[\begin{array}{l} \text{mp}(T_1), \cdots, \text{mp}(T_{j-1}), \\ \text{mp}(T'_1), \cdots, \text{mp}(T'_{k_2-1}), \downarrow_{\pi'_3}(\text{mp}(T'_{k_2})) \end{array} \right] \right) \right) \\
&=^{(7)} \mathit{Cost}(\downarrow_{(k-1+k_2)\cdot\pi'_3}(\text{mp}(T)))
\end{aligned}
$$

1. , 3., and 7. definition of partial evaluation.
2. , 4., and 6. definition of *Cost*.
5. point (b) for finite evaluations and induction hypothesis for π_3.

– If $k > j$:

$$
\begin{aligned}
& Cost(\downarrow_{\pi}(T)) \\
=^{(1)}\ & Cost(t(c(\boldsymbol{a}:\boldsymbol{b}),[T_1,\cdots,\downarrow_{\pi_2}(T_k)]) \\
=^{(2)}\ & \sum_{i=1}^{k-1} Cost(T_i) + Cost(\downarrow_{\pi_2}(T_k)) \\
=^{(3)}\ & \sum_{i=1}^{j-1} Cost(T_i) + \sum_{i=1}^{m} Cost(T'_i) + \sum_{i=j+1}^{k-1} Cost(T_i) + Cost(\downarrow_{\pi_2}(T_k)) \\
=^{(4)}\ & \sum_{i=1}^{j-1} Cost(\mathrm{mp}(T_i)) + \sum_{i=1}^{m} Cost(\mathrm{mp}(T'_i)) + \sum_{i=j+1}^{k-1} Cost(\mathrm{mp}(T_i)) + Cost(\downarrow_{\pi'_2}(\mathrm{mp}(T_k))) \\
=^{(5)}\ & Cost\left(t\left(c.d_j(\boldsymbol{a}:\boldsymbol{b}),\left[\begin{array}{l}\mathrm{mp}(T_1),\cdots,\mathrm{mp}(T_{j-1}),\mathrm{mp}(T'_1),\cdots,\mathrm{mp}(T'_m),\\ \mathrm{mp}(T_{j+1}),\cdots,\downarrow_{\pi'_2}(\mathrm{mp}(T_k))\end{array}\right]\right)\right) \\
=^{(6)}\ & Cost(\downarrow_{(m+k-1)\cdot\pi'_2}(\mathrm{mp}(T)))
\end{aligned}
$$

1. and 6. definition of partial evaluation.
2. and 5. definition of *Cost*.
3. cost of T_j.
4. point (b) for finite evaluations and induction hypothesis for π_2.

In order to prove that the transformation is precise, the inverse of the selected mp can be considered and the proof for point (a) can be considered backwards to prove that for every $T' \in [\![CRS'|C(\boldsymbol{a}:\boldsymbol{b})]\!]$, the evaluation $\mathrm{mp}^{-1}(T) \in [\![CRS|C(\boldsymbol{a}:\boldsymbol{b})]\!]$. The proofs for points (b) and (c) can also be reversed. In particular, for point (c), note that given a position $\pi = q \cdot \pi'$ for an evaluation of a CE $c.d_j$, q can be expressed either as:

- $n \cdot \pi'$ with $n < j$
- $j \cdot \pi'$
- $(k-1+k_2) \cdot \pi'$ with $k = j$ and $1 < k_2 \leq m$
- $(k-1+m) \cdot \pi'$ with $k > j$

which fall into the cases considered in the proof of point (c).

4.6.2 Proof of Theorem 4.7

Let $C_1, C_2 \subseteq CRS$ and let $\boldsymbol{x_1 y_1}$ and $\boldsymbol{x_2 y_2}$ be in input and output variables of C_1 and C_2 respectively. The variables of C_m are $\boldsymbol{x_m} = \boldsymbol{x_1 z_1} = \boldsymbol{x_2 z_2}$ where either $\boldsymbol{z_1}$ or $\boldsymbol{z_2}$ is an empty sequence of variables and $\boldsymbol{y_m} = \boldsymbol{y_1 w_1} = \boldsymbol{y_2 w_2}$ where either $\boldsymbol{w_1}$ or $\boldsymbol{w_2}$ is empty. The transformation substitutes C_1 and C_2 by C_m. It can be proved sound by using a small variant of Lemma 4.19.

Corollary 4.20. *Let* mp *be a function that satisfies points (b) and (c) of Lemma 4.19. If*

- $\mathrm{mp}([\![CRS|C_1(\boldsymbol{a}:\boldsymbol{b})]\!]) \subseteq [\![CRS'|C_m(\boldsymbol{a0}:\boldsymbol{b0})]\!]$
- $\mathrm{mp}([\![CRS|C_2(\boldsymbol{a}:\boldsymbol{b})]\!]) \subseteq [\![CRS'|C_m(\boldsymbol{a0}:\boldsymbol{b0})]\!]$
- $\mathrm{mp}([\![CRS|C(\boldsymbol{a}:\boldsymbol{b})]\!]) \subseteq [\![CRS'|C(\boldsymbol{a}:\boldsymbol{b})]\!]$ *for every* $C \neq C_1$ *and* $C \neq C_2$

where $\mathbf{0}$ *are sequences of zeros that complete the parameters to the right length. Then, the transformation is sound. That is, let* $f(\boldsymbol{x_m})$ *be a bound for* C_m *in* CRS' *then* $f(\boldsymbol{x_1 0})$ *is a bound for* C_1 *in* CRS *and* $f(\boldsymbol{x_2 0})$ *is a bound for* C_2 *in* CRS.

The function mp simply maps the tree nodes from C_1 and C_2 to nodes of C_m with the extra variables set to zero.

$$\text{mp}(T) = \begin{cases} t(c(\boldsymbol{a0} : \boldsymbol{b0}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)]) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \wedge c \in C_1 \cup C_2 \\ t(c(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)]) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \wedge\ c \notin C_1 \cup C_2 \\ T & \text{if } T \in \mathbb{Q} \end{cases}$$

This definition of mp trivially satisfies points (b) and (c) (only the parameters in the nodes are changed but not its structure or numeric nodes). Let us prove the premises of Corollary 4.20.

Finite Evaluations

We prove it by induction on the height of the evaluation tree. In the base case, T is a linear expression $T \in [\![CRS|l\sigma]\!]_c$ and $\text{mp}(T) = T \in [\![CRS'|l\sigma]\!]_c$. In the inductive case, we have $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ and we can assume the conditions of the semantics hold for T:

1. $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$
3. $\exists\sigma$ such that $\text{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma,\ \models (\bigwedge_{i=0}^{n} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{n} T_i \in [\![CRS|b_i\sigma]\!]_c$

and we distinguish cases depending on whether $c \in C_1$, $c \in C_2$ or $c \notin (C_1 \cup C_2)$.

- If $c \notin (C_1 \cup C_2)$, we have $\text{mp}(T) = t(c(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)])$. The corresponding cost equation in CRS' is $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b'_1, \varphi_1, \cdots, b'_n, \varphi_n \in CRS'$ where each b'_i is the result of the substitution $b'_i = b_i[C_1(\boldsymbol{x_1} : \boldsymbol{y_1})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m}), C_2(\boldsymbol{x_2} : \boldsymbol{y_2})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m})]$. Consequently, conditions (1) and (2) are satisfied. For condition (3), the assignment σ can be extended to assign zeros to the extra parameters of the calls to C_1 and C_2. We refer to such an extension as σ' and we have that if $b_i\sigma = C_1(\boldsymbol{a_i} : \boldsymbol{b_i})$ then $b'_i\sigma' = C_m(\boldsymbol{a_i 0} : \boldsymbol{b_i 0})$ and the same applies to $b_i\sigma = C_2(\boldsymbol{a_i} : \boldsymbol{b_i})$. Therefore, we can apply the induction hypothesis for every $b'_i\sigma'$ to guarantee condition (4).

- If $c \in C_1$, the reasoning is similar. We have $\text{mp}(T) = t(c(\boldsymbol{a0} : \boldsymbol{b0}), [\text{mp}(T_1), \cdots, \text{mp}(T_n)])$. The cost equation in CRS' is $c\colon C(\boldsymbol{x_m} : \boldsymbol{y_m}) = \varphi_0, b'_1, \varphi_1, \cdots, b'_n, \varphi_n \in CRS'$ where each b'_i is the result of the substitution $b'_i = b_i[C_1(\boldsymbol{x_1} : \boldsymbol{y_1})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m}), C_2(\boldsymbol{x_2} : \boldsymbol{y_2})/C_m(\boldsymbol{x_m} : \boldsymbol{y_m})]$. Consequently, condition (1) and (2) are satisfied. For condition (3), the assignment σ can be extended to assign zeros to the extra variables in the calls to C_1 and C_2 and in the head of c. As before, We refer to such an extension as σ' and we have that if $b_i\sigma = C_1(\boldsymbol{a_i} : \boldsymbol{b_i})$ then $b'_i\sigma' = C_m(\boldsymbol{a_i 0} : \boldsymbol{b_i 0})$. Therefore, we can apply the induction hypothesis for every $b'_i\sigma'$ to guarantee condition (4).

- The case where $c \in C_2$ is equivalent to the case of $c \in C_1$.

Infinite evaluations

For infinite evaluations we prove that the sets $\text{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega)$ for $C \neq C_1$ and $C \neq C_2$ and $\text{mp}([\![CRS|C_1(\boldsymbol{a_1} : \boldsymbol{b_1})]\!]_\omega) \cup \text{mp}([\![CRS|C_2(\boldsymbol{a_2} : \boldsymbol{b_2})]\!]_\omega)$ constitute a fixpoint. Let $T \in [\![CRS|C(\boldsymbol{x} : \boldsymbol{y})]\!]_\omega)$ with the form $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$, the reasoning is the same as with the induction proof i.e. the assignment σ is simply extended to map the extra variables to zero, but the co-induction hypothesis is applied for the infinite branch T_n.

4.6.3 Proof of Theorem 4.12

For proving Theorem 4.12, we use Lemma 4.19 to prove soundness and precision. For soundness, we use a function mp that simply changes the label of the evaluation nodes of c by c': $\text{mp}(T) := T[c/c']$. This substitution does not change the cost of T or of its partial evaluations. Therefore, points (b) and (c) are trivially satisfied. For precision, we choose the inverse substitution $\text{mp}'(T) := T[c'/c]$.

Let us prove point (a) for soundness. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ such that $\bigwedge_{0\leq i\leq j} \varphi_i = \bot$. There cannot be any complete evaluation that contains a node with c in CRS (there is no assignment σ that satisfies all the constraints) and the same happens with c' in CRS'. Therefore, $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c = \text{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c) = [\![CRS \setminus \{c\}|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c = [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ for every C and every $\boldsymbol{a} : \boldsymbol{b}$. That is, finite evaluations are not affected by the transformation. This is also valid for precision.

Consider now infinite evaluations. We take the sets $\text{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega)$ for every $\boldsymbol{a} : \boldsymbol{b}$ and C and show they correspond to a fixpoint of the infinite evaluations in CRS'. Let $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m]) \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$, we can assume that the conditions of the semantics hold for T:

1. $d : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $\exists m \leq n$ such that $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$
3. $\exists \sigma$ such that $\text{Dm}(\sigma) = vars(d), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{m-1} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{m-1} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_m \in [\![CRS|b_m\sigma]\!]_\omega$

We have $\text{mp}(T) = t(d[c/c'](\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_m)])$ and we distinguish cases:

- If $d \neq c$, we have $\text{mp}(T) = t(d(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_m)])$. Conditions (1) and (2) are satisfied with $d \in CRS'$. For condition (3), the assignment σ used in T is valid for $\text{mp}(T)$. For condition (4), $\text{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $i < m$ has been proved above and $\text{mp}(T_m) \in [\![CRS'|b_m\sigma]\!]_\omega$ holds by co-induction hypothesis.

- If $d = c$, we have $\text{mp}(T) = t(c'(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_m)])$ and the number of sub-evaluations m has to be smaller than j (from $\bigwedge_{0\leq i\leq j} \varphi_i = \bot$) so the conditions (1) and (2) are satisfied with $c' : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_j, \bot \in CRS'$. For condition (3), we can restrict σ to the variables in c' and such a σ' still satisfies the constraints ($\bigwedge_{i=0}^{m-1} \varphi_i$). For condition (4), we have that $b_i\sigma' = b_i\sigma$ for every $1 \leq i \leq m$, $\text{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $i < m$ has been proved above, and $\text{mp}(T_m) \in [\![CRS'|b_m\sigma]\!]_\omega$ holds by co-induction hypothesis.

The proof for infinite evaluations can also be reversed to prove precision. In the case $d = c'$ the assignment σ' can be extended arbitrarily to the variables of c.

4.6.4 Proof of Theorem 4.14

Similarly to the previous proof, we use Lemma 4.19 with $\text{mp}(T) := T[c/c']$ (so points (b) and (c) are trivial) for soundness and the inverse substitution $\text{mp}'(T) := T[c'/c]$ for precision.

We prove point (a) (soundness) for finite evaluations as before. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$, if we know b_j is always non-terminating ($[\![CRS|b_j]\!]_c = \emptyset$), there cannot be any a complete evaluation that contains c. Therefore, $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c = [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ for every C and all $\boldsymbol{a} : \boldsymbol{b}$. This also guarantees point (a) for the inverse transformation and thus guarantees precision for finite evaluations.

The case for infinite evaluations is also similar to the previous proof (the proof of Theorem 4.12). Consider an infinite evaluation $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m]) \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ that satisfies the conditions of the semantics:

1. $d : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $\exists m \leq n$ such that $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$
3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(d), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{m-1} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{m-1} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_m \in [\![CRS|b_m\sigma]\!]_\omega$

and the corresponding $T' = t(d[c/c'](\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_m)]) \in \mathrm{mp}([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega)$. The case where $d \neq c$ is equal to the same case in the previous proof so we only have to consider the case where $d = c$. In such a case, if b_j is always non-terminating, we have $m \leq j$ so the conditions (1) and (2) are satisfied with $c' : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_j \in CRS'$. For condition (3), we can restrict σ to the variables in c' and such a σ' still satisfies the constraints $(\bigwedge_{i=0}^{m-1} \varphi_i)$. For condition (4), we have that $b_i\sigma' = b_i\sigma$ for every $1 \leq i \leq m$, $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $i < m$ has been proved above, and $\mathrm{mp}(T_m) \in [\![CRS'|b_m\sigma]\!]_\omega$ holds by co-induction hypothesis.

In this case, the reasoning can also be reversed to prove precision.

4.6.5 Proof of Theorem 4.16

We use Lemma 4.19 with $\mathrm{mp}(T) := T[c/c']$ for soundness and $\mathrm{mp}'(T) := T[c'/c]$ for precision. We have a cost equation $c : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ where b_j is always terminating ($[\![CRS|b_j]\!]_\omega = \emptyset$).

Proof of Point (a) for Finite evaluations

Let T be a finite evalutation ($T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ or $T \in [\![CRS|l\sigma]\!]_c$), and we have to prove that the corresponding evaluation $\mathrm{mp}(T)$ is in $[\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ or $[\![CRS|l\sigma]\!]_c$. This can be proven by induction on the structure of finite evaluations. In the base case, T is a linear expression evaluation. Therefore, $\mathrm{mp}(T) = T \in [\![CRS'|l\sigma]\!]_c$.

For the inductive step, $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ and has the form $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ so we can assume all the conditions of the semantics are satisfied for T:

1. $d : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$
3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(d), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{n} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{n} T_i \in [\![CRS|b_i\sigma]\!]_c$

- If $d \neq c$, we have $\mathrm{mp}(T) = t(d(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_n)])$. Conditions (1) and (2) of the semantics are satisfied with the same CE $d \in CRS'$. For condition (3), the assignment σ used in T is also valid for $\mathrm{mp}(T)$. For condition (4), we can apply the induction hypothesis. If $T_i \in [\![CRS|b_i\sigma]\!]_c$ then $\mathrm{mp}(T_i) \in [\![CRS'|b_i\sigma]\!]_c$ for $1 \leq i \leq n$.

- If $d = c$, then $\mathrm{mp}(T) = t(c'(\boldsymbol{a} : \boldsymbol{b}), [\mathrm{mp}(T_1), \cdots, \mathrm{mp}(T_n)])$. Conditions (1) and (2) hold with $c' : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{i-1} \wedge \varphi_i, b_i, \cdots, b_n, \varphi_n \in CRS'$. The σ used for T is also valid for $\mathrm{mp}(T)$ (note that the conjunction of all the constraints is equivalent). For condition (4), we can apply the induction hypothesis as in the previous case.

Proof of Point (a) for Infinite evaluations

For infinite evaluations, we prove that the sets $\text{mp}(\llbracket CRS|C(\boldsymbol{a} : \boldsymbol{b})\rrbracket_\omega)$ (we lift mp to sets of evaluations) are a fixpoint of the infinite evaluations of in CRS'.

Let $T \in \llbracket CRS|C(\boldsymbol{a} : \boldsymbol{b})\rrbracket_\omega$ with the form $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$. We assume the conditions of the semantics hold for T:

1. $d : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C$
2. $\exists m \leq n$ such that $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$
3. $\exists \sigma$ such that $\text{Dm}(\sigma) = vars(d), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\bigwedge_{i=0}^{m-1} \varphi_i)\sigma$
4. $\bigwedge_{i=1}^{m-1} (T_i \in \llbracket CRS|b_i\sigma\rrbracket_c) \wedge T_m \in \llbracket CRS|b_m\sigma\rrbracket_\omega$

and we distinguish cases:

- If $d \neq c$, this corresponds to the cases where $d \neq c$ in the previous proofs.
- If $d = c$, then $\text{mp}(T) = t(c'(\boldsymbol{a} : \boldsymbol{b}), [\text{mp}(T_1), \cdots, \text{mp}(T_m)])$. Conditions (1) and (2) hold with $c' : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_{i-1} \wedge \varphi_i, b_i, \cdots, b_n, \varphi_n \in CRS'$. In this case, we know that b_j is terminating so the infinite evaluation T_m cannot be of b_j ($m \neq j$). In both cases $m < j$ or $m > j$, the conjunction of constraints is the same for c and c' and is satisfied by the same σ so condition (3) is satisfied. For condition (4) we have that $\text{mp}(T_i) \in \llbracket CRS'|b_i\sigma\rrbracket_c$ for $i < m$ (thanks to the proof for finite evaluations above) and $\text{mp}(T_m) \in \llbracket CRS'|b_i\sigma\rrbracket_\omega$ because of the co-induction hypothesis.

In this case, the reasoning can also be reversed to prove precision with mp'.

4.6.6 Proof of Theorem 4.18

In this case, we also use Lemma 4.19 with $\text{mp}(T) := T[c/c']$. The proof for point (a) for soundness follows closely the structure of the previous proof. We have that $c : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_i, \cdots, b_n, \varphi_n$ and $c' : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_i', \cdots, b_n, \varphi_n$ such that $\varphi_i \Rightarrow \varphi_i'$. So the only difference is when considering the case is $T = t(d(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$ where $d = c$. In such a case, conditions (1) and (2) are satisfied for $\text{mp}(T)$ with c' and condition (3) is also satisfied with the same σ used for T (both in the cases where T is finite or infinite). Condition (4) is guaranteed as in previous proofs by applying induction hypothesis or co-induction hypothesis to the corresponding sub-evaluations.

The proof for precision is inmediate. Let $\text{mp}'(T) := T$ be the identity function, we have that if $T \in \llbracket CRS'|C(\boldsymbol{a} : \boldsymbol{b})\rrbracket$, then $\text{mp}'(T) = T \in \llbracket CRS|C(\boldsymbol{a} : \boldsymbol{b})\rrbracket$. This is because $CRS' = CRS \setminus \{c\}$.

5 Refinement

The ultimate objective of the analysis is to obtain bounds of cost relations in a cost relation system. That is, obtain a function f that bounds the cost of every evaluation in $[\![CRS|C]\!]$ for CR C. A cost relation is usually composed of several cost equations that can interact in subtle ways. This can lead to many evaluations with distinct costs. If we partition the set of all evaluations $[\![CRS|C]\!]$ into sets of evaluations that are similar, we can then obtain bounds for each of these partitions. The range of possible behaviors of the evaluations in each of these partitions is more restricted, only a subset of the evaluations is considered. Therefore, finding a bound for a partition is easier than finding a bound for the complete cost relation and the bound can be (hopefully) more precise.

The key underlying assumption of the refinement is that the cost of an evaluation is closely related to the cost equations that are used in the evaluation and the order in which they are used. The refinement uses this information (the evaluation patterns) to partition the set of all possible evaluations $[\![CRS|C]\!]$. In addition, the refinement transforms the original cost equations into a simplified format called *refined cost equations*. A refined cost equation has a single constraint set at the beginning of its body, all the elements in its body are evaluated, and it is used only in complete evaluations or in infinite evaluations (but not in both). This refined format simplifies the later bound computation.

After the preprocessing step, a CRS can be represented as a sequence of cost relations $\langle C_1, C_2, \cdots, C_n \rangle$ in which each cost relation C_i can only contain calls to C_j with $j \geq i$. The refinement for each cost relation in $\langle C_1, C_2, \cdots, C_n \rangle$ is performed incrementally following a bottom-up approach, that is, starting from C_n and finishing with C_1.

Given a cost relation C_i, the refinement has the following steps:

1. A control-flow refinement over the cost relation is performed (Section 5.2). First, the control-flow refinement detects and enumerates a set of *chains* (evaluation patterns) that reflect all possible behaviors of C_i. Second, it attempts to discard non-terminating patterns by proving termination (Section 5.2.2). Third, it transforms the cost equations of C_i into refined cost equations (Section 5.2.3) and last, it generates a set of *refined chains* over the refined CEs. These refined chains represent set of evaluations that are either all terminating or all non-terminating (Section 5.2.4).

2. For each of the chains, summaries and calling contexts are computed and they are used for discarding unfeasible patterns and strengthen the CEs' constraints (Section 5.3).

3. Finally, once the cost relation C_i has been completely refined, the refinement is propagated to previous cost relations C_j for $j < i$. Any cost equation that calls C_i is specialized to call specific chains of C_i instead (Section 5.4).

First, the refinement is presented for cost relations with linear recursion, i.e. cost relations where the cost equations $c \in C$ can have at most one recursive call to C. This is later generalized to CRS with non-linear recursion in Section 5.5.

5.1 Partially Refined Cost Equations

Given a cost relation that is going to be refined next, its cost equations can be represented in a restricted format in which there are at most two constraint sets, one at the beginning of the CE and one after the recursive call (if there is such a call).

Claim 5.1. *Let CRS* $:= \langle C_1, C_2, \cdots, C_m \rangle$ *such that* $C_{i+1}, \cdots, C_m$ *have been refined. Then for every* $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n \in C_i$*, every* b_j *that is not a direct recursive call i.e.* $b_j \neq C_i(\boldsymbol{x} : \boldsymbol{y})$ *is either always terminating* $[\![CRS|b_j]\!]_\omega = \emptyset$ *or non-terminating* $[\![CRS|b_j]\!]_c = \emptyset$*.*

Consider a CE $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \varphi_1, \cdots, b_n, \varphi_n$ of the last cost relation C_m in the sequence $\langle C_1, C_2, \cdots, C_m \rangle$. CE c contains at most one call $b_i = C(\boldsymbol{x_i} : \boldsymbol{y_i})$. All the other b_j for $j \neq i$ have to be linear expressions. Linear expressions are guaranteed to terminate so Constraint Compression (Definition 4.15) can be applied to obtain a CE where all the constraints are grouped either at the beginning of the CE or after the recursive call:

$$C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_a, b_1, \cdots, b_{i-1}, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \varphi_b, b_i, \cdots, b_n \tag{5.1}$$

where $\varphi_a = \varphi_0 \wedge \cdots \wedge \varphi_{i-1}$ are the constraints before the call and $\varphi_b = \varphi_i \wedge \cdots \wedge \varphi_n$ are the constraints after the call. If the CE does not contain a recursive call, it has the form:

$$C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_a, b_1, \cdots, b_n \tag{5.2}$$

In the other cost relations C_j with $j < m$, there are can be calls to other CR and do not terminate. However, as already mentioned, terminating and non-terminating calls are explicitly distinguished. Given a call to a terminating chain, Constraint Compression can be applied as before. If on the contrary, there is a call to a non-terminating chain, Unreachable Code Elimination (Definition 4.13) can be applied to remove everything after the call. Therefore, the CEs in C_j with $j < m$ also have one of the formats above with the difference that the last b_n can be a call to a non-terminating chain.

A partially refined cost equation can be *recursive* if it has the shape of Equation 5.1, or *non-recursive* if it has the shape of Equation 5.2. A cost equation is *tail-divergent* if b_n is a non-terminating call.

5.2 Control-flow Refinement of a Cost Relation

In order to perform the refinement of a cost relation, a call-graph is computed (similarly to what is done in Chapter 4), but this time at the level of cost equations within a cost relation.

Definition 5.2 (CE Call Relation)**.** Let C be a cost relation and let $c, d \in C$. CE c calls d, written $c \to d$, if $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi, b_1, \cdots, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \cdots$ and $d\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi', \cdots$ and $\varphi \wedge (\varphi'[\boldsymbol{xy}/\boldsymbol{x_i y_i}])$ is satisfiable.

The CE call relation induces a call-graph $\mathcal{G}(C)$ within the cost relation C. In principle, an evaluation of C is a tree $t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$. But if we consider the nodes of the tree that result from evaluating cost equations in C, they are all contained in a single branch (remember that we are considering linear recursion). The order in which nodes $c(\boldsymbol{a} : \boldsymbol{b})$ appear in such a branch is determined by the CE call-graph. In particular, any branch formed by its cost equations corresponds to a valid path of the CE call-graph.

The refinement partitions the (possibly infinite) paths in the call-graph into a finite number of patterns, called chains. For that purpose, it first detects the strongly connected components (SCCs) of the call-graph. The SCCs give rise to *phases* and *chains* are formed by sequences of phases.

Definition 5.3 (Phase)**.** Let S be the set of vertices of a SCC in $\mathcal{G}(C)$. If $S = \{c_1, \cdots, c_m\}$ is recursive, it generates an *iterative phase* $ph := (c_1 \vee \cdots \vee c_m)^+$ where the cost equations in S are evaluated a positive finite number of times and a *divergent phase* $ph := (c_1 \vee \cdots \vee c_m)^\omega$ where the cost equations in S are evaluated an infinite number of times. If $S = \{c\}$ cannot iterate $(c \not\to c)$, it generates a *non-iterative phase* $ph := (c)$ in which c is evaluated once.

Note that non-recursive cost equations will always form non-iterative phases. However, it is also possible to have a non-iterative phase (c) where c is recursive as long as $c \not\to c$.

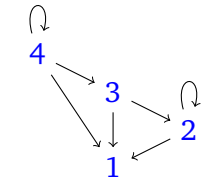
Program 11

```
while(i<x){
 if(i=j)
  tick(j);
 else
  tick(1);
 i++;
}
```

Cost relations

$1: wh(i,j,x) = \{i \geq x\},\ 0$
$2: wh(i,j,x) = \{i < x, i > j, i' = i+1\},\ 1,\ wh(i',j,x)$
$3: wh(i,j,x) = \{i < x, i = j, i' = i+1\},\ j,\ wh(i',j,x)$
$4: wh(i,j,x) = \{i < x, i < j, i' = i+1\},\ 1,\ wh(i',j,x)$

Call-graph



Phases

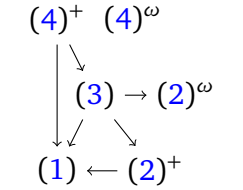


Figure 5.1.: Program 11: Example with different kinds of phases

The CE call relation can be lifted to the level of phases. Let ph_1 and ph_2 be the phases generated from two different SCCs S_1 and S_2 in $\mathcal{G}(C)$. We have $ph_1 \to ph_2$ if and only if ph_1 is not a divergent phase and there are $c_1 \in S_1$, $c_2 \in S_2$ such that $c_1 \to c_2$.

Example 5.4. Figure 5.1 contains an example program with multiple phases. The program has a loop in which variable i is incremented until it reaches x. In each iteration the loop consumes some resources. If $i = j$, it consumes j resource units. Otherwise, it consumes one resource unit. The former case is represented by CE 3 and the latter is represented by CEs 2 and 4. Figure 5.1 also contains the cost relation call-graph. In this call-graph we can see for example that CE 2 cannot be evaluated directly after CE 4 or that CE 4 can never be evaluated after CE 2.

The SCCs of the call-graph give rise to the phases. For instance, SCC {4} generates the iterative phase $(4)^+$ and the divergent phase $(4)^\omega$. The SCCs {3} and {1} generate the non-iterative phases (3) and (1) respectively. Note that the phase (3) is non-iterative but it contains a recursive CE. Figure 5.1 also contains a graph with the call relations among phases. Note that this graph is acyclic.

Definition 5.5 (Chain)**.** A *chain* is a sequence of phases defined recursively:

$$ch := \begin{array}{ll} ph & \text{if } ph = (c) \text{ and } c \text{ is not recursive, or } ph = (c_1 \vee \cdots \vee c_m)^\omega \\ \mid ph \cdot ch' & \text{if } ch' = ph'\_ \text{ and } ph \to ph' \end{array}$$

A *chain* is a sequence of phases $ch = ph_1 \cdot ph_2 \cdots ph_n$ such that for every $1 \leq i < n$ we have $ph_i \to ph_{i+1}$ and the last phase is either a base case $ph_n := (c)$ (c is of the form of equation 5.2) or divergent. The relation $\to$ between phases is acyclic so the number of possible chains is finite. Note that a chain cannot end with a non-iterative phase that is not a base case neither with an iterative phase.

Example 5.6. The chains of Program 11 (Figure 5.1) are the following:

Terminating:	$(4)^+(3)(2)^+(1)$	$(4)^+(3)(1)$	$(4)^+(1)$	$(3)(2)^+(1)$	$(3)(1)$	$(2)^+(1)$	(1)
Non-Terminating:	$(4)^+(3)(2)^\omega$		$(4)^\omega$	$(3)(2)^\omega$		$(2)^\omega$	

5.2.1 Chain Evaluations

Now we can define chain evaluations. Intuitively, a chain evaluation is an evaluation of a cost relation C in which the cost equations of C are evaluated following the pattern described by the chain. This concept

can be formally defined. First, when we refer to the pattern in which the CEs of C are evaluated, we consider paths from the root of the evaluation that only visit nodes of C and are as long as possible.

Definition 5.7 (CR-paths)**.** Let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ be an evaluation tree, its CR-paths are:

$$paths_C(T) = \begin{cases} c(\boldsymbol{a} : \boldsymbol{b}) \cdot p' & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [\cdots T' \cdots]) \ \wedge label(T') \in C \wedge p' \in paths_C(T') \\ c(\boldsymbol{a} : \boldsymbol{b}) & \text{if } T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \ \wedge (\forall 1 \leq i \leq n : label(T_i) \notin C) \end{cases}$$

A CR-path p of T can be finite and end at a node that does not contain more recursive calls in C or it can be infinite. The pattern of CEs followed by these paths is determined by its labels so the function *label* is extended to CR-paths. Let $p = c_1(\boldsymbol{a}_1 : \boldsymbol{b}_1) \cdot c_2(\boldsymbol{a}_2 : \boldsymbol{b}_2) \cdots$ be a CR-path of an evaluation tree, its evaluation pattern is $label(p) := c_1 \cdot c_2 \cdots$. Similarly, the function *label* is lifted to sets of paths.

Finally, a chain induces either a regular language, if its last phase is not recursive, or an ω-regular language, if its last phase is divergent. The language defined by a chain ch is denoted $\mathscr{L}_{ch}$. Given these definitions, the evaluations of a chain are defined as follows:

Definition 5.8 (Chain evaluations)**.** Let ch be a chain of CR C and $\boldsymbol{a} : \boldsymbol{b}$ some parameters, its chain evaluations are:

$$[\![CRS|C[ch](\boldsymbol{a} : \boldsymbol{b})]\!] := \left\{ \ T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!] \ \middle| \ label(paths_C(T)) \subseteq \mathscr{L}_{ch} \ \right\}$$

Theorem 5.9 (Chain Completeness)**.** *Let $ch_1, \cdots, ch_n$ be the chains for C. Any possible evaluation of C for some parameters $\boldsymbol{a} : \boldsymbol{b}$ is covered by a chain evaluation:*

$$[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!] = \bigcup_{i=1}^{n} [\![CRS|C[ch_i](\boldsymbol{a} : \boldsymbol{b})]\!]$$

Proof of Theorem 5.9. The direction $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!] \supseteq \bigcup_{i=1}^{n} [\![CRS|C[ch_i](\boldsymbol{a} : \boldsymbol{b})]\!]$ holds directly by Definition 5.8. Let us prove $[\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!] \subseteq \bigcup_{i=1}^{n} [\![CRS|C[ch_i](\boldsymbol{a} : \boldsymbol{b})]\!]$. That is, let $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ there is a chain ch_i such that $T \in [\![CRS|C[ch_i](\boldsymbol{a} : \boldsymbol{b})]\!]$.

In the case of linear recursion there is only one CR-path per evaluation $|paths_C(T))| = 1$. We consider this path p and distinguish cases on according to whether p is finite or infinite:

- For finite paths p, we prove by induction on the length of the path that there is a ch such that $label(p) \in \mathscr{L}_{ch}$.

  Base case We have $p = c(\boldsymbol{a} : \boldsymbol{b})$. Given the definition of $paths_C$, we know that the evaluation corresponding to the node $c(\boldsymbol{a} : \boldsymbol{b})$ does not have any sub-evaluation that represents a recursive call in C. Therefore, c is not recursive (remember that c is partially refined as described in Section 5.1). Consequently, there is a non-iterative phase and a chain (c) and $label(p) \in \mathscr{L}_{(c)}$.

  Inductive step We assume that there is a chain ch such that $label(p) \in \mathscr{L}_{ch}$ holds. Let $p' = c(\boldsymbol{a} : \boldsymbol{b}) \cdot p$, we prove that there is a chain ch' such that $label(p') \in \mathscr{L}_{ch'}$.

  Let ph be the first phase of ch, if $c \in ph$ then the phase has to be iterative and $label(p') \in \mathscr{L}_{ch}$. Otherwise, $c \in ph'$ such that ph' is not divergent (all CEs belong to one non-divergent phase). Besides, let c' be the CE of the first element of p, we have that $c \to c'$ and $c' \in ph$. Consequently, $ph' \to ph$, the chain $ch' = ph' \cdot ch$ is valid, and $label(p') \in \mathscr{L}_{ch'}$.

- Let p be infinite, because the number of CEs is finite, there has to be some CE c that appears infinitely often in p. We can divide p into two sub-paths $p = p_1 \cdot p_2$ such that $p_2 = c(\boldsymbol{a} : \boldsymbol{b}) \cdots$ starts with the first occurrence of c. The sub-path p_1 is finite and p_2 is infinite. Because c appears infinitely often in p_2, every other c' that appears in p_2 belongs to the same SCC S in the call-graph. Let ph be the divergent phase generated from S, it generates a valid chain ph and $label(p_2) \in \mathscr{L}_{ph}$.

Now we reason about the finite prefix p_1 by induction as in the previous case. We conclude that there is a chain ch such that $label(p) \in \mathscr{L}_{ch}$.

□

5.2.2 Discarding Divergent Phases

Let $ph = (c_1 \vee c_2 \vee \cdots \vee c_m)^\omega$ be a divergent phase, if termination of ph is proven, the phase and all the chains that end with it can be discarded.

Consider a phase $(c_1 \vee c_2 \vee \cdots \vee c_m)^\omega$, and let $c_i : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_i, \cdots, C(\boldsymbol{x'} : \boldsymbol{y'}), \varphi'_i, \cdots$ for $1 \leq i \leq m$, proving termination of the phase is equivalent to proving termination of a multi-path linear constraint loop where each path of the loop is $\varphi_i \downarrow \boldsymbol{x}\boldsymbol{x'}$ (the projection of φ_i onto the input variables of the head and the recursive call). We assume the standard notation in which a multi-path linear constraint loop is represented as a disjunction of linear constraint sets over the variables $\boldsymbol{x}$ and $\boldsymbol{x'}$ (see [BAG14]).

Definition 5.10 (Phase Loop)**.** Let $(c_1 \vee \cdots \vee c_m)^\omega$ be a phase where each cost equation has the form $c_i : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_i, \cdots, C(\boldsymbol{x'} : \boldsymbol{y'}), \varphi'_i, \cdots$ for $1 \leq i \leq m$, its phase loop is:

$$\bigvee_{i=1}^{m} \varphi_i \downarrow \boldsymbol{x}\boldsymbol{x'}$$

where $\boldsymbol{x}$ and $\boldsymbol{x'}$ correspond to the variables before and after a "loop transition" and each disjunction corresponds to a loop path.

Theorem 5.11. *If the loop of a phase $ph = (c_1 \vee \cdots \vee c_m)^\omega$ is terminating, the phase is unfeasible i.e. the set of evaluations of the phase $[\![CRS|C[ph]]\!]$ is empty.*

Proof. Let us assume otherwise, there is $T \in [\![CRS|C[ph]]\!]$. We have that every evaluation in $[\![CRS|C]\!]$ must have at least one CR-path, that is $paths_C(T)$ cannot be empty. Any path in $p \in paths_C(T)$ corresponds to a valid execution of the phase loop of ph. If the phase loop is terminating, there is a well-founded ordering on the elements of p and it must be finite. However, by definition if $T \in [\![CRS|C[ph]]\!]$, then $label(p) \in \mathscr{L}_{ph}$ and p must be infinite. This leads to a contradiction. Consequently, there cannot be an evaluation $T \in [\![CRS|C[ph]]\!]$. □

There are many alternatives to prove termination of such a loop. The current implementation infers lexicographical linear ranking functions using a technique similar to the one in [BAG14].

Example 5.12. The chains of Program 11 (defined in Example 5.6) have two divergent phases $(2)^\omega$ and $(4)^\omega$. CE 2 corresponds to a loop $\varphi = \{i < x, i > j, i' = i + 1, j' = j, x' = x\}$ and the expression $x - i$ is a valid ranking function of such a loop. Therefore, we can discard the phase $(2)^\omega$. Similarly, CE 4 corresponds to a loop $\varphi = \{i < x, i < j, i' = i + 1, j' = j, x' = x\}$. The expression $x - i$ is also a valid ranking function of that loop and the phase $(4)^\omega$ can also be discarded. The remaining chains of Program 11 are all terminating:

$$(4)^+(3)(2)^+(1) \qquad (4)^+(3)(1) \qquad (4)^+(1) \qquad (3)(2)^+(1) \qquad (3)(1) \qquad (2)^+(1) \qquad (1)$$

5.2.3 Refined Cost Equations

So far, the refinement has generated a set of chains that represent all the possible evaluations of a cost relation C and it has attempted to discard non-terminating evaluations that end up in divergent phases.

However, two issues remain. First, the cost equations that compose the chains are still partially refined (see Section 5.1). That is, they can contain several constraint sets and their recursive calls might terminate or not. And second, the chains do not separate terminating and non-terminating evaluations in all cases (there is an example of this in the next sub-section).

At this point, the refinement performs a case distinction for each recursive CE depending on whether the recursive call terminates or not. The result of this case distinction are *refined cost equations* which have a single constraint set and all its calls are evaluated (only the last one might diverge). These cost equations are simpler and are the ones considered for the rest of the analysis.

Definition 5.13 (CE Case Distinction)**.** Let $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_j:\boldsymbol{y}_j), \varphi_1, b_{j+1}, \cdots, b_n$ be a recursive CE, the case distinction generates two refined cost equations. A partial CE:

$$c^p : C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_j:\boldsymbol{y}_j)$$

and a tail-divergent CE or a complete CE depending on whether c is tail-divergent or not:

$$\begin{array}{ll} c^{td}\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0 \wedge \varphi_1, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_j:\boldsymbol{y}_j), \varphi_1, b_{j+1}, \cdots, b_n & \text{if } c \text{ is tail-divergent} \\ c^{c}\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0 \wedge \varphi_1, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_j:\boldsymbol{y}_j), \varphi_1, b_{j+1}, \cdots, b_n & \text{otherwise} \end{array}$$

For non-recursive CEs of the form $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, b_n$, the case distinction generates a tail-divergent CE $c^{td}\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, b_n$ or a complete CE $c^c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi_0, b_1, \cdots, b_n$ depending on whether c is tail-divergent or not. The case distinction is performed for all CEs in the cost relation.

The *partial* CE c^p represents the case where the recursive call $C_j(\boldsymbol{x}_j:\boldsymbol{y}_j)$ does not finish and thus the calls (or costs) in $b_{j+1}, \cdots, b_n$ are never evaluated. The *complete* CE c^c is the case where $C_j(\boldsymbol{x}_j:\boldsymbol{y}_j)$ finishes and all calls are completely evaluated. Finally, a *tail-divergent* CE c^{td} is the case where $C_j(\boldsymbol{x}_j:\boldsymbol{y}_j)$ finishes and all calls are evaluated but b_n diverges.

Definition 5.14 (Refined Cost Equation)**.** A *refined cost equation* has the form:

$$c^X : C(\boldsymbol{x}:\boldsymbol{y}) = \varphi, b_1, b_2, \cdots, b_n$$

where c^X can be c^c, c^p, or c^{td}. A refined CE contains one constraint set at the beginning φ and all b_i are either calls to specific chains $C_j[ch](\boldsymbol{x}_i:\boldsymbol{y}_i)$, direct recursive calls $C_i(\boldsymbol{x}_i:\boldsymbol{y}_i)$, or linear expressions. In addition, we establish the requirement that all b_i have to be evaluated (the evaluation cannot diverge in a b_i for $i < n$) and b_n has to be completely evaluated if the CE is complete c^c.

The additional requirement can be seen as an extra condition on the semantics for refined CEs. This condition guarantees that finite evaluations contain only complete CEs and the infinite branches in infinite evaluations are formed by partial or tail-divergent CEs.

Theorem 5.15. *The CE case distinction is sound and precise.*

Proof of Theorem 5.15. Soundness can be proved using Lemma 4.19 in a similar way to the proofs in Section 4.6. Remember that this lemma establishes three sufficient conditions for the soundness of a transformation. It is based on a cost preserving function that maps evaluations in the original *CRS* to evaluations in the transformed *CRS*′. We define a function *sp* that satisfies the tree points of the lemma.

$$sp(T) = \begin{cases} t(c^{td}(\boldsymbol{a}:\boldsymbol{b}), [sp(T_1), \cdots, sp(T_n)]) & \text{if } T = t(c(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_n]) \wedge T_n \in [\![CRS|C']\!]_\omega \wedge C' \neq C \\ t(c^{p}(\boldsymbol{a}:\boldsymbol{b}), [sp(T_1), \cdots, sp(T_n)]) & \text{if } T = t(c(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_n]) \wedge T_n \in [\![CRS|C]\!]_\omega \\ t(c^{c}(\boldsymbol{a}:\boldsymbol{b}), [sp(T_1), \cdots, sp(T_n)]) & \text{if } T = t(c(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_n]) \wedge T_n \text{ is finite} \\ t(d(\boldsymbol{a}:\boldsymbol{b}), [sp(T_1), \cdots, sp(T_n)]) & \text{if } T = t(d(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_n]) \wedge d \notin C \\ T & \text{if } T \in \mathbb{Q} \end{cases}$$

The function sp maps the labels of the evaluation nodes of C from c to c^p, c^{td}, or c^c according to whether the evaluation diverges in the recursive call, diverges in other call or it does not diverge respectively. No other changes are made to the evaluations so points (b) and (c) of the Lemma 4.19 (cost preservation) are trivial.

Proof of Point (a) for Finite evaluations

Let T be a finite evaluation ($T \in [\![CRS|C'(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ for some C' or $T \in [\![CRS|l\sigma]\!]_c$), we have to prove that the corresponding evaluation $sp(T)$ is in $[\![CRS'|C'(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ or $[\![CRS'|l\sigma]\!]_c$. This can be proven by induction on the structure of finite evaluations. In the base case, T is a linear expression evaluation and $sp(T) = T \in [\![CRS'|l\sigma]\!]_c$.

For the inductive step, we only have to consider the case $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$ with $c \in C$. The case where $c \notin C$ is immediate as in the proofs of Section 4.6. We assume c is recursive and all the conditions of the semantics are satisfied for T:

1. $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \varphi_1, b_{j+1}, \cdots, b_n \in C$
2. $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$
3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models (\varphi_0 \wedge \varphi_1)\sigma$
4. $\bigwedge_{1 \le i \le n \wedge i \neq j} T_i \in [\![CRS|b_i\sigma]\!]_c \wedge T_j \in [\![CRS|C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma]\!]_c$

We have $sp(T) = t(c^c(\boldsymbol{a} : \boldsymbol{b}), [sp(T_1), \cdots, sp(T_n)])$. Conditions (1) and (2) hold with $c^c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0 \wedge \varphi_1, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), b_{j+1}, \cdots, b_n \in CRS'$ (note that c cannot be tail-divergent). The σ used for T is also valid for $sp(T)$ (note that the conjunction of all the constraints is equivalent). For condition (4), we can apply the induction hypothesis. The additional requirement for refined CEs that all b_i are completely evaluated is also satisfied. The case where c is not recursive is analogous.

Proof of Point (a) for Infinite evaluations

For infinite evaluations, again we only have to consider the case $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$ with $c \in C$. The case where $c \notin C$ is immediate. We assume c is recursive and the conditions of the semantics hold for T:

1. $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \varphi_1, b_{j+1}, \cdots, b_n \in C$
2. $\exists m \le n$ such that $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_m])$

For conditions (3) and (4), there are two possibilities (all the b_i except b_n have to be terminating):

- if $m = j$:

  3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models \varphi_0\sigma$
  4. $\bigwedge_{i=1}^{m-1} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_m \in [\![CRS|C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma]\!]_\omega$

  then $sp(T) = t(c^p(\boldsymbol{a} : \boldsymbol{b}), [sp(T_1), \cdots, sp(T_m)])$. Conditions (1) and (2) hold with $c^p\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_1 : \boldsymbol{y}_1) \in CRS'$. We can restrict σ to the variables in c^p. The restricted assignment σ' satisfies condition (3) and we have $b_i\sigma = b_i\sigma'$ and $C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma = C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma'$ so we have that $sp(T_i) \in [\![CRS'|b_i\sigma']\!]_c$ for $i < m$ (thanks to the proof for finite evaluations above) and $sp(T_m) \in [\![CRS'|C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma']\!]_\omega$ because of the co-induction hypothesis. The additional condition that all b_i in the refined CE have to be evaluated is also satisfied.

- Otherwise $m = n$ (c has to be tail-divergent):

  3. $\exists \sigma$ such that $\mathrm{Dm}(\sigma) = vars(c), \boldsymbol{ab} = \boldsymbol{xy}\sigma, \models \varphi_0\sigma$
  4. $\bigwedge_{1 \le i < m \wedge i \neq j} (T_i \in [\![CRS|b_i\sigma]\!]_c) \wedge T_j \in [\![CRS|C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma]\!]_c \wedge T_m \in [\![CRS|b_m\sigma]\!]_\omega$

then $sp(T) = t(c^{td}(\boldsymbol{a} : \boldsymbol{b}), [sp(T_1), \cdots, sp(T_m)])$. Conditions (1) and (2) hold with c^{td}: $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0 \wedge \varphi_1, b_1, \cdots, b_{j-1}, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), b_{j+1}, \cdots, b_n \in CRS'$. The same assignment σ satisfies condition (3) and we have that $sp(T_i) \in [\![CRS' | b_i\sigma]\!]_c$ for $1 \leq i < m$ and $i \neq j$ and $sp(T_j) \in [\![CRS' | C(\boldsymbol{x}_1 : \boldsymbol{y}_1)\sigma]\!]_c$ (thanks to the proof for finite evaluations above), and $sp(T_m) \in [\![CRS' | b_m\sigma]\!]_\omega$ because of the co-induction hypothesis. The additional condition that all b_i in the refined CE have to be evaluated is also satisfied.

The case where c is not recursive is analogous to the latter where c is tail-divergent.

CE case distinction is precise

Point (a) basically proves that $sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c) \subseteq [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ and $sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega) \subseteq [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ for every C and every $\boldsymbol{a} : \boldsymbol{b}$. In order to prove precision, it is enough to show the other direction, that is, $sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c) \supseteq [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ and $sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega) \supseteq [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$. This, in turn, is equivalent to showing that any evaluation $T \in [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]$ is the result applying sp to an evaluation $T' \in [\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]$.

In this case, it is easy to see that any evaluation $T \in [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]$ matches one of the cases obtained by applying sp in the previous proof. In particular, the additional requirements on the evaluations of refined CEs prevent us from having finite evaluations that contain partial CEs and from having infinite evaluations that diverge before their last call.

□

This result is useful later on and thus we establish it in a lemma.

Lemma 5.16. *Let CRS and CRS′ be the cost relation systems before and after CE case distinction. For any C and any parameters* $\boldsymbol{a} : \boldsymbol{b}$*, we have:*

$$sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c) = [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$$
$$sp([\![CRS | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega) = [\![CRS' | C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$$

5.2.4 Refined Chains

Next, the refinement procedure generates chains formed by refined cost equations. These refined chains represent either only terminating evaluations or only non-terminating evaluations. These chains are not computed by examining the call-graph of the refined cost equations but rather by distinguishing cases for the already existing chains.

In the case of tail recursive cost relations (which are the ones originated from loops) the distinction between terminating and non-terminating evaluations is already explicit. If a CR is tail recursive, the partially refined CEs have the form: $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_a, b_1, \cdots, b_{i-1}, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \varphi_b$ or $C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_a, b_1, \cdots, b_n$. All the calls except b_n and $C(\boldsymbol{x}_i : \boldsymbol{y}_i)$ have to be terminating, otherwise the CEs could be simplified further. Therefore, given a chain $ch = ph_1 \cdot ph_2 \cdots ph_m$, it is non-terminating if and only if $ph_m = (c_1 \vee \cdots \vee c_k)^\omega$ or $ph_m = (c)$ and c is tail-divergent (because its last call b_n is non-terminating). Consequently, each original chain over partially refined CEs generates one chain (terminating or non-terminating) over the refined CEs.

Example 5.17. Consider the following tail recursive cost relation:

$$1: tail(l, m) = \{l = 0\}, 0$$
$$2: tail(l, m) = \{l \geq 1, l' = l - 1, m \leq x, x < 0\}, inf(x), tail(l', m)$$
$$3: tail(l, m) = \{l \geq 1, l' = l - 1, m \leq x, x \geq 0\}, tail(l', m)$$

The cost relation decreases l until it reaches 0. In each iteration a non-deterministic value x (and bounded from below by m) is selected and if it is negative a CR inf is called. If we assume that inf is

always non-terminating, we can apply unreachable code elimination (Definition 4.13) to CE 2 and obtain a tail-divergent CE

$$2\colon tail(l,m) = \{l \geq 1, l' = l-1, m \leq x, x < 0\},\ inf(x)$$

Once this simplification has been performed, CE 2 is no longer recursive and the chains of CR *tail* can be clearly divided into terminating and non-terminating.

Terminating:	$(3)^+(1)$	(1)	
Non-terminating:	$(3)^+(2)$	(2)	$(3)^\omega$

Chains $(3)^+(2)$ and (2) are non-terminating because they end in a tail-divergent CE.

If some partially refined cost equations are non-tail recursive, that is, a CE with calls after the recursive call $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_a, b_1, \cdots, b_{i-1}, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \varphi_b, b_i, \cdots, b_n$, there can still be chains that might terminate or not. If c is tail divergent, it has a non-terminating call b_n after the recursive call $C(\boldsymbol{x}_i : \boldsymbol{y}_i)$ has been completely evaluated. Therefore, c does not have to appear at the end of the chain and can be part of an iterative phase $ph := (c_1 \vee \cdots \vee c_m)^+$. In such a case, the termination of ph will depend on whether CE c is ever evaluated and we have to distinguish cases.

Example 5.18. Consider a non tail recursive variant of the cost relation from Example 5.17:

$$1\colon nonTail(l,m) = \{l = 0\},\ 0$$
$$2\colon nonTail(l,m) = \{l \geq 1, l' = l-1, m \leq x, x < 0\},\ nonTail(l',m),\ inf(x)$$
$$3\colon nonTail(l,m) = \{l \geq 1, l' = l-1, m \leq x, x \geq 0\},\ nonTail(l',m)$$

In this case, CE 2 cannot be simplified further and the chains of *nonTail* are:

$$(2 \vee 3)^+(1) \qquad (1) \qquad (2 \vee 3)^\omega$$

While chains (1) and $(2 \vee 3)^\omega$ are respectively terminating and non-terminating, chain $(2 \vee 3)^+(1)$ can terminate or not depending on whether CE 3 is evaluated in the phase $(2 \vee 3)^+$.

In order to generate chains over refined cost equations, the refinement distinguishes cases for phases and for chains.

Definition 5.19 (Phase Case Distinction)**.** Let $ph := (c_1 \vee \cdots \vee c_m)^+$ be an iterative phase, such that $c_1, \cdots, c_i$ for $0 \leq i \leq m$ are tail-divergent, the following phases are generated:

1. A *complete phase* $ph^c := (c^c_{i+1} \vee \cdots \vee c^c_m)^+$ where the phase is completely evaluated. Not all CEs can be tail-divergent ($i < m$).
2. A *partial phase* $ph^p := (c^p_1 \vee \cdots \vee c^p_m)^+$ where the recursive calls diverge but not within the phase. All the b_i after the recursive calls are not evaluated.
3. A *tail-divergent phase* $ph^{td} := (c^p_1 \vee \cdots \vee c^p_m \vee c^{td}_1 \vee \cdots \vee c^{td}_i \vee c^c_{i+1} \vee \cdots \vee c^c_m)^+$ where a tail-divergent CE is evaluated at some point in the phase. The evaluation diverges within the phase. The phase must contain at least one tail-divergent CE ($i \geq 1$).

If none of the CE in the phase is tail-divergent (i=0), only cases (1) and (2) are generated. If on the contrary, all CE in the phase are tail-divergent (i=m) only cases (2) and (3) are generated.

Non-iterative phases $ph := (c)$ are a particular case of the above. If c is not tail-divergent, a partial phase $ph^p := (c^p)$ and a complete phase $ph^c := (c^c)$ are generated. If c is tail-divergent, a partial phase $ph^p := (c^p)$ and tail-divergent phase $ph^{td} := (c^{td})$ are generated.

Finally, a divergent phase $ph := (c_1 \vee \cdots \vee c_m)^\omega$ generates $ph^d := (c^p_1 \vee \cdots \vee c^p_m)^\omega$.

The pattern of tail-divergent phases is not precise, in fact it can match any of the other patterns (complete and partial phases). A more precise pattern would be:

$$(c_1^p \vee \cdots \vee c_m^p)^*(c_1^{td} \vee \cdots \vee c_i^{td})(c_{i+1}^c \vee \cdots \vee c_m^c)^*$$

However, keeping the simpler pattern does not affect the soundness of the analysis and it allows us to maintain a uniform format for the phases.

Example 5.20. Continuing with Example 5.18, we generate the following phases for CR *nonTail*:

Complete	Partial	Tail-divergent	Divergent
$(3^c)^+$ (1^c)	$(2^p \vee 3^p)^+$	$(2^{td} \vee 3^c \vee 2^p \vee 3^p)^+$	$(2^p \vee 3^p)^\omega$

Finally, chains can also be refined into terminating and non-terminating chains.

Definition 5.21 (Refined Chains)**.** Let ch be a chain, the functions *term* and *nterm* receive a chain and return its terminating and non-terminating refinements. A terminating chain is written ch^c and a non-terminating chain is written ch^ω.

$$term(ch) := \begin{cases} ph^c & \text{if } ch = ph \\ ph^c \cdot ch^{c\prime} & \text{if } ch = ph \cdot ch' \text{ and } ch^{c\prime} = term(ch') \end{cases}$$

$$nterm(ch) := \begin{cases} \{ph^d\} & \text{if } ch = ph \wedge\ ph \text{ is divergent} \\ \{ph^{td}\} & \text{if } ch = ph \wedge ph = (c) \\ \{ph^{td} \cdot ch^{c\prime} \mid ch^{c\prime} = term(ch')\} & \\ \cup\{ph^p \cdot ch^{\omega\prime} \mid ch^{\omega\prime} \in nterm(ch')\} & \text{if } ch = ph \cdot ch' \end{cases}$$

The function *term* returns a single terminating chain whereas *nterm* returns a (possibly empty) set of non-terminating chains. Note though that *term* is not defined for all chains. For instance, it is not defined for chains whose last phase is divergent.

Example 5.22. The resulting chains generated from the phases in Example 5.20 are the following[1]:

Terminating chains:	$(3^c)^+(1^c)$	(1^c)
Non-terminating chains:	$(2^{td} \vee 3^c \vee 2^p \vee 3^p)^+(1^c)$	$(2^p \vee 3^p)^\omega$

The evaluations of refined chains are defined as the evaluations of chains (Definition 5.8) but we only define finite evaluations for terminating chains and infinite evaluations for non-terminating chains:

Definition 5.23 (Refined Chain Evaluations)**.** Let ch^c and ch^ω be respectively a terminating and non-terminating chain of CR C and $\boldsymbol{a} : \boldsymbol{b}$ some parameters, their evaluations are:

$$[\![CRS|C[ch^c](\boldsymbol{a} : \boldsymbol{b})]\!]_c := \left\{\ T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c \ \middle|\ label(paths_C(T)) \subseteq \mathscr{L}_{ch^c}\ \right\}$$

$$[\![CRS|C[ch^\omega](\boldsymbol{a} : \boldsymbol{b})]\!]_\omega := \left\{\ T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega \ \middle|\ label(paths_C(T)) \subseteq \mathscr{L}_{ch^\omega}\ \right\}$$

Thanks to this case distinction, terminating and non-terminating evaluations are separated. This is important to guarantee Claim 5.1 for the remaining cost relations.

In the common case where everything is terminating (all divergent phases have been discarded and there are no tail-divergent CEs), the case distinction generates only a terminating chain for each original

[1] The chain $(2^p \vee 3^p)^\omega$ can be proved terminating and be discarded.

chain. On the other hand, if some CEs are tail-divergent, the case distinction generates different non-terminating chains depending on where non-termination originates.

Example 5.24. Consider a chain $ch := ph_1 \cdot ph_2 \cdot ph_3 \cdot ph_4$ where ph_1 and ph_3 contain tail-divergent CEs and ph_4 is a base case. The generated non-terminating chains are:

$$ch^{\omega}{}_1 := ph_1^p \cdot ph_2^p \cdot ph_3^{td} \cdot ph_4^c$$
$$ch^{\omega}{}_2 := ph_1^{td} \cdot ph_2^c \cdot ph_3^c \cdot ph_4^c$$

However, if ph_4 was divergent, the only refined chain would be $ch^{\omega} := ph_1^p \cdot ph_2^p \cdot ph_3^p \cdot ph_4^d$.

The following theorem is an updated version of Theorem 5.9 (Chains Completeness) for the refined chains.

Theorem 5.25 (Refined Chains Completeness)**.** *Let CRS' be the cost relation system after the CE case distinction (Definition 5.13). Let $ch^c{}_1, \cdots, ch^c{}_n$ and $ch^{\omega}{}_1, \cdots, ch^{\omega}{}_m$ be the terminating and non-terminating refined chains of C. We have that for all parameters $\boldsymbol{a} : \boldsymbol{b}$:*

$$[\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c = \bigcup_{i=1}^{n} [\![CRS'|C[ch^c{}_i](\boldsymbol{a} : \boldsymbol{b})]\!]_c \tag{5.3}$$

$$[\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega} = \bigcup_{i=1}^{m} [\![CRS'|C[ch^{\omega}{}_i](\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega} \tag{5.4}$$

Proof of Theorem 5.25. The direction $\supseteq$ holds by Definition 5.23. The other direction $\subseteq$ can be proved based on Lemma 5.16 ($sp([\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]) = [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]$) and based on Theorem 5.9 (Chain completeness before the case distinction).

Consider Equation 5.3. Let $T' \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$, we know that there is a $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_c$ such that $sp(T) = T'$ (Lemma 5.16). For T there is a chain ch such that $T \in [\![CRS|C[ch](\boldsymbol{a} : \boldsymbol{b})]\!]_c$ (Theorem 5.9). It is enough to prove that every $p' \in paths_C(sp(T))$ matches $term(ch) = ch^c$. Because T is finite, sp maps every c to c^c. Additionally, we know that T and any of its paths $p \in paths_C(T)$ do not contain any node with a tail-divergent CE. Such paths match the original chain $label(p) \in \mathscr{L}_{ch}$ and $ch^c = term(ch)$ corresponds to ch in which every CE c has been substituted by c^c and the tail-divergent CEs have been excluded. Therefore, for every $p' \in paths_c(sp(T))$ we have $label(p') \in \mathscr{L}_{ch^c}$ and $sp(T) = T' \in [\![CRS'|C[ch^c](\boldsymbol{a} : \boldsymbol{b})]\!]_c$.

Consider Equation 5.4. Let $T' \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega}$, we know that there is a $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega}$ such that $sp(T) = T'$. For T there is a chain ch such that $T \in [\![CRS|C[ch](\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega}$. Therefore, it is enough to prove that all $p' \in paths_C(sp(T))$ match a chain in $nterm(ch)$. We prove it by induction on the length of the chain (number of phases).

In the base case $ch = ph$, there are two possibilities.

- If ph is divergent, ph^d is well-defined and so is $ch^{\omega} = ph^d$. In addition, any sub-evaluation T'' of a CE in the phase is infinite and therefore $sp(T'')$ maps c to c^p and for any path $p' \in paths_C(T')$ we have $label(p') \in \mathscr{L}_{ch^{\omega}}$.
- If the phase is non-recursive $ph = (c)$, $label(sp(T)) = c^{td}$ (c has to be tail divergent so T is infinite). Therefore, we have that $ph^{td} = (c^c)$ and $ch^{\omega} = ph^{td}$ are well-defined. This phase contains all the possible cases of the CEs in ph so $label(p) \in \mathscr{L}_{ch^{\omega}}$.

Let us consider now the inductive case. The chain has the form $ch = ph \cdot ch'$. Every path $p \in paths_C(T)$ and can be split into two parts $p = p_1 \cdot p_2$ where $p_2 \in \mathscr{L}_{ch'}$ where p_1 is common for all the paths p. The corresponding path in T' is $p' = p'_1 \cdot p'_2$. Let T_2 be the evaluation whose root starts with the first node of p_2 i.e. $p_2 \in paths_C(T_2)$. The evaluation T_2 might be finite or infinite.

- If T_2 is finite, then $T_2 \in [\![CRS|C[ch']](\boldsymbol{a} : \boldsymbol{b})]\!]_c$ and $sp(T_2) \in [\![CRS'|C[ch^{c'}]](\boldsymbol{a} : \boldsymbol{b})]\!]_c$ (this is the finite case which we have already proved). Because T is infinite, there must be a call to a non-terminating chain so there must be a $c_i \in ph$ that is tail-divergent. This means that the phase ph^{td} is well-defined and p'_1 matches ph^{td} (ph^{td} contains both the complete, the tail-divergent, and the partial versions of the CEs in ph), $ch^{\omega} = ph^{td} \cdot ch^{c'}$ is defined, and every path p' matches ch^{ω}.
- If T_2 is infinite, then $T_2 \in [\![CRS|C[ch']](\boldsymbol{a} : \boldsymbol{b})]\!]_{\omega}$ and there is a chain $ch^{\omega'}$ such that $sp(T_2) \in [\![CRS'|C_r[ch^{\omega'}](\boldsymbol{x}\boldsymbol{y})]\!]_{\omega}$ by induction hypothesis and p'_2 matches $ch^{\omega'}$. Moreover, we have that every node in p_1 is mapped to c^p (because they contain an infinite evaluation on C) and thus p'_1 matches ph^p. The chain $ch^{\omega} = ph^p \cdot ch^{\omega'}$ is defined and every p' matches it.

□

5.3 Invariants

Once the CR evaluations have been split into a set of chains, the refinement procedure infers two types of invariants for each chain: summaries and calling contexts. This inference is done over refined chains. However, we use the generic symbols *ch* and *ph* if we do not want to distinguish between terminating (ch^c) and non-terminating chains (ch^{ω}) or between different kinds of phases.

5.3.1 Chain Summaries

A summary of a chain *ch* is a constraint set that over-approximates the behavior of *ch* and relates the input and output variables of the cost relation.

Definition 5.26 (Chain Summary)**.** Let *ch* be a chain of CR C, the constraint set $\varphi(\boldsymbol{x}\boldsymbol{y})$ is a summary of chain *ch* if for every evaluation of the chain $T \in [\![CRS|C[ch]]\!]$ with $rt(T) = c(\boldsymbol{a} : \boldsymbol{b})$, the constraint set φ is valid with respect to $\boldsymbol{a} : \boldsymbol{b}$, i.e. $\models \varphi(\boldsymbol{a}\boldsymbol{b})$.

Example 5.27. The chain summaries of Program 11 (in Page 53) are the following[2]:

Chain	Summary	Chain	Summary
$(4)^+(3)(2)^+(1)$	$\{x \geq j+2, j \geq i+1\}$	$(4)^+(3)(1)$	$\{j+1 = x, j \geq i+1\}$
$(4)^+(1)$	$\{j \geq x, x \geq i+1\}$	$(3)(2)^+(1)$	$\{j = i, x \geq j+2\}$
$(3)(1)$	$\{j+1 = x, j = i\}$	$(2)^+(1)$	$\{i \geq j+1, x \geq i+1\}$
(1)	$\{i \geq x\}$		

In this program, the final values of the variables have not been included in the cost relation representation so the summaries correspond to necessary preconditions on the chains.

Example 5.28. Let us consider Program 8 (in Figure 1.14 in Page 8). The program implements the function *take* that takes n elements from a list l. The cost relation *take* has the following terminating chains with the corresponding summaries:

Chain	Summary	Chain	Summary
$(3)^+(1)$	$\{n \geq 1, l \geq n, n = ret\}$	(1)	$\{0 \geq n, ret = 0\}$
$(3)^+(2)$	$\{l \geq 1, n > l, l = ret\}$	(2)	$\{l = 0, n \geq 1, ret = 0\}$

[2] For terminating cost relations the CEs are referred by c instead of c^c, e.g $(3)(1)$ instead of $(3^c)(1^c)$, to keep the notation simple.

Here we can see how a summary provides conditions for the applicability of the chain, i.e. a precondition, and input-output relations. For instance, in order to evaluate the chain $(3)^+(2)$ the initial value of n must be higher than the size of the list l ($n > l$). In that case the returned list has the size of l ($ret = l$).

Chain summaries are computed by propagating information backward from the end of a chain to the beginning using the polyhedra abstract domain [CH78] (although other domains such as the octagon domain [Min06] could be considered). Let us introduce some notation. A constraint set φ corresponds to a polyhedron. The operation $\varphi \downarrow \boldsymbol{x}$ corresponds to the projection of φ over the variables $\boldsymbol{x}$ and the operation $\varphi_1 \sqcup \varphi_2$ is the convex hull (least upper bound) of φ_1 and φ_2.

Next, we define the operations necessary to compute chain summaries. The operation $\tau^{-1}(c, \varphi)$ receives a CE c and a constraint set φ that represents a state and abstractly executes the CE inversely (hence the -1 super-script) with respect to such state. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1$[3] be a refined CE:

$$\tau^{-1}(c, \varphi) = (\varphi(\boldsymbol{x_1 y_1}) \wedge \varphi_c) \downarrow \boldsymbol{xy}$$

The operation can be extended to disjunctions of cost equations:

$$\tau^{-1}((c_1 \vee c_2 \vee \cdots \vee c_n), \varphi) = \bigsqcup_{i=1}^{n} \tau^{-1}(c_i, \varphi)$$

and to a pattern E (e.g. a disjunction of cost equations) applied one or zero times:

$$\tau^{-1}(E^{0-1}, \varphi) = \tau^{-1}(E, \varphi) \sqcup \varphi$$

Finally, the operation is defined for a pattern E applied and unknown number of times and for a positive number of times. The latter corresponds to the pattern of iterative phases. This is defined as a fixpoint iteration of the previous pattern:

$$\tau^{-1}(E^*, \varphi) = \text{fix}(\tau^{-1}(E^{0-1}))(\varphi)$$
$$\tau^{-1}(E^+, \varphi) = \tau^{-1}(E, \tau^{-1}(E^*, \varphi))$$

The refinement procedure uses widening to ensure termination of the fixpoint computation.

Once the operation τ^{-1} has been defined for all possible patterns corresponding to iterative and non-iterative phases, it can be used to compute chain summaries. The summary of a chain with a single $ch := ph$ is defined by the convex hull of all the constraint sets φ_i of the CEs in the phase $c_i \in ph$ projected onto the input and output variables. This is the case even if ph is divergent. This is equivalent to assume that some CE in the divergent phase is evaluated at least once which is sound. The summary of a chain formed by several phases is computed recursively. Let $ch := ph_1 \cdot ch'$, its summary is computed by applying the operation τ^{-1} with the pattern ph to the summary of ch'. The formal definition is:

$$summary(ch) = \begin{cases} \bigsqcup \{\ \varphi \downarrow \boldsymbol{xy} \mid c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi \cdots \in ph\ \} & \text{if } ch = ph \\ \tau^{-1}(ph, summary(ch')) & \text{if } ch = ph \cdot ch' \end{cases}$$

Observation 5.29. $summary(ch)$ returns a summary of ch.

[3] We use $\boldsymbol{b}$ in cost equations to denote a sequence of calls or linear expressions $b_1, \cdots, b_n$.

5.3.2 Calling Contexts

The refinement also considers calling contexts, which are invariants that are valid at a certain point of the evaluation with respect to some entry set. In contrast to summaries, calling contexts are propagated forward along the chains and they are defined in terms of finite chain prefixes.

Definition 5.30 (Chain Prefix)**.** Let $ch = ph_1 \cdots ph_n$, a prefix of ch is defined as $pfx := ph_i \cdot ph_{i-1} \cdots ph_1$ for $0 \leq i < n$. The empty prefix ($i = 0$) is written ϵ. If $ph_n = (c_1 \vee \cdots \vee c_n)^\omega$, then $pfx = (c_1 \vee \cdots \vee c_n)^+ \cdot ph_{n-1} \cdots ph_1$ is also a valid prefix.

In order to define calling contexts in terms of evaluations, we need an auxiliary notion to identify where the evaluation of a given cost relation starts.

Definition 5.31 (Maximal Evaluation of C in T)**.** Let T be an evaluation, T' is maximal in T with respect to C if it is a sub-evaluation $T' \preceq T$, it is an evaluation $T' \in [\![CRS|C]\!]$ and no ancestor T'' of T' ($T' \preceq T'' \preceq T$) is in $[\![CRS|C]\!]$.

Definition 5.32 (Calling Context)**.** Let CRS be a cost relation system with an entry set ES. A constraint set $\varphi(\boldsymbol{x})$ is a calling context of a prefix pfx if it is valid $\models \varphi(\boldsymbol{a})$ for any evaluation $T \in [\![CRS_{ES}|C(\boldsymbol{a} : \boldsymbol{b})]\!]$ that satisfies the following conditions:

- T belongs to an evaluation $T_e \in [\![CRS|E]\!]$ for $E \in ES$ ($T \preceq T_e$)
- T_i is a maximal evaluation of C in T_e with $T \preceq T_i \preceq T_e$
- The path from T to T_i matches the prefix pfx (without including T)

Note that these calling contexts are only "context sensitive" with respect to the path inside the cost relation C but they do not distinguish between the different paths that can reach C from the entries.

Example 5.33. Let us consider a variant of Program 11 in which the while loop is defined inside a function `void foo(int i,int x){ int j=i+10; while(i<x){...}}` which corresponds to the following cost relation $foo(i,x) = \{j = i + 10\}, wh(i,j,x)$. In this example CR wh is called only at one point under the condition $j = i + 10$. This is the initial calling context of CR wh, that is, the calling context of its empty prefix. The remaining calling contexts of CR wh are:

Chain prefix	Calling context	Chain prefix	Calling context
ϵ	$\{j = i + 10\}$	$(4)^+$	$\{x \geq i, j \geq i \geq j - 9\}$
$(2)^+$	$\bot$	$(3)(4)^+$	$\{j = i - 1, x \geq j + 1\}$
(3)	$\bot$	$(2)^+(3)(4)^+$	$\{i \geq j + 2, x \geq i\}$

Note that the calling context for ϵ is incompatible with the constraint sets of CEs 3 and 2 of wh. Consequently, the calling contexts for the prefixes $(2)^+$ and (3) are $\bot$.

The inference of calling contexts is very similar to the inference of chain summaries but it is defined recursively over chain prefixes. Let C be a cost relation, the calling context of the empty prefix ϵ for C is an empty constraint set $\top$ if C is an entry ($C \in ES$), or an over-approximation (convex hull) of all the calling contexts in which C is called from other CRs otherwise. For the recursive case, we define the operation $\tau^1(c, \varphi)$ that receives a CE c and a constraint set φ and abstractly executes the CE. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c, \boldsymbol{b}_0, C(\boldsymbol{x'} : \boldsymbol{y'}), \boldsymbol{b}_1$ be a refined CE:

$$\tau^1(c, \varphi) = (\varphi(\boldsymbol{x}) \wedge \varphi_c) \downarrow \boldsymbol{x'}$$

We lift the operation for τ^1 to the patterns of iterative phases as in the case of τ^{-1} (see previous section). The computation of calling contexts is:

$$callCxt(pfx) = \begin{cases} \top & \text{if } pfx = \epsilon \wedge C \in ES \\ \bigsqcup \left\{ \begin{array}{l|l} \varphi \downarrow \boldsymbol{x}' & \begin{array}{l} c\colon C'(\boldsymbol{x}:\boldsymbol{y}) = \varphi, \cdots, C(\boldsymbol{x}':\boldsymbol{y}') \cdots \in CRS \\ \wedge\, C' \neq C \end{array} \end{array} \right\} & \text{if } pfx = \epsilon \wedge C \notin ES \\ \tau^1(ph, callCxt(pfx')) & \text{if } pfx = ph \cdot pfx' \end{cases}$$

Observation 5.34. *callCxt*(*pfx*) returns a calling context of *pfx*.

5.3.3 Discarding Unfeasible Chains

Both chain summaries and calling contexts can be used to detect and discard unfeasible chains.

Observation 5.35. If *summary*(*ch*) is unsatisfiable, then $[\![CRS|C[ch]]\!]$ is empty and *ch* can be discarded. Any chain *ch*′ that has *ch* as a suffix can also be discarded.

Observation 5.36. If *callCxt*(*pfx*) is unsatisfiable, then for any chain *ch* such that *pfx* is its prefix $[\![CRS_{ES}|C[ch]]\!]$ is empty and *ch* can be discarded.

These observations rely on Observations 5.29 and 5.34 and the chain summary and calling context definitions (Definition 5.26 and Definition 5.32).

Example 5.37. Continuing with Example 5.33, we can discard the chains (3)(2)$^+$(1), (3)(1) and (2)$^+$(1) because they contain the prefixes (2)$^+$ and (3) whose calling contexts are unsatisfiable.

5.3.4 Strengthening Cost Equations

The information contained in calling contexts and chain summaries can be useful for its bound computation and even for other steps in the analysis such as proving termination.

The refinement procedure strengthens cost equations' constraints using the inferred summaries and calling contexts. It does so in a context insensitive manner. For a cost equation *c*, it computes a summary for its recursive calls by joining all the summaries of the chains that can be evaluated in such recursive call. These are the chains where *c* is in the first phase or chains that come immediately after a phase that contains *c* (the recursive call of a CE can already belong to a different phase).

Definition 5.38 (Cost Equation Summary)**.** Let *C* be a cost relation with a set of chains *CH*. Let $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi, \boldsymbol{b}, C(\boldsymbol{x}':\boldsymbol{y}'), \boldsymbol{b}$ be a recursive CE, its summary is:

$$summary(c) = \bigsqcup \left\{ summary(ch)(\boldsymbol{x}'\boldsymbol{y}') \mid (ph \cdot ch \in CH \wedge c \in ph) \vee (ch = ph\_ \in CH \wedge c \in ph) \right\}$$

Lemma 5.39. *For every* $t(c(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_j, \cdots, T_n]) \in [\![CRS|C]\!]$ *where* T_j *corresponds to a recursive call* $C(\boldsymbol{x}':\boldsymbol{y}')$ *and* $param(T_j) = \boldsymbol{a}'\boldsymbol{b}'$ *then* $\models summary(c)(\boldsymbol{a}'\boldsymbol{b}')$.

Proof. The lemma is a consequence of the completeness of refined chains (Theorem 5.25), every T_j has to belong to one of the chains considered for computing the summary of *c*, and of Observation 5.29 that guarantees that *summary*(*ch*) generates a summary of *ch*. □

Definition 5.40 (Call Summary Strengthening)**.** Let $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi, \boldsymbol{b}_0, C(\boldsymbol{x}':\boldsymbol{y}'), \boldsymbol{b}_1$ be a recursive CE, the strengthened CE is $c'\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi \wedge summary(c), \boldsymbol{b}_0, C(\boldsymbol{x}':\boldsymbol{y}'), \boldsymbol{b}_1$. The transformation generates $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Corollary 5.41. *Call Summary Strengthening is sound and precise.*

Program 12	Cost relations
`void strengthen(int x){`	
`2 int s=1;`	$1: strengthen(x) = \{s = 1\}, 1, wh(x,s)$
`3 while(x>0)`	$2: wh(x,s) = \{x \leq 0\}$
`4 x=x-s;`	$3: wh(x,s) = \{x > 0, x' = x - s\},\ 1,\ wh(x',s)$
`5 }`	

Figure 5.2.: Program 12: Example where strengthening is important

Similarly, for a cost equation c, the refinement computes a calling context for its head by considering the calling contexts of all the prefixes that can lead to c.

Definition 5.42 (Cost Equation Calling Context)**.** Let C be a cost relation with a set of chain prefixes PFX. Let c be a cost equation in C, its calling context is:

$$callCxt(c) = \bigsqcup \{\ callCxt(pfx) \mid (ph \cdot pfx \in PFX \wedge c \in ph) \vee (pfx = ph\_ \in CH \wedge c \in ph)\ \}$$

Lemma 5.43. *For every* $t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![CRS_{ES}|C]\!]$, *we have* $\models callCxt(c)(\boldsymbol{a}\boldsymbol{b})$.

Proof. The lemma is a consequence of the completeness of the chain case distinction (Theorem 5.25), every T_j has to belong to one of the chains and its corresponding prefixes are considered for computing the calling contexts of c. For each of the prefixes pfx, $callCxt(pfx)$ is a valid calling context because of Observation 5.34. □

Definition 5.44 (Calling Context Strengthening)**.** Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi, \boldsymbol{b}$ be a CE, the strengthened CE is $c'\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi \wedge callCxt(c), \boldsymbol{b}$. The transformation generates $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Corollary 5.45. *Calling Context Strengthening is sound and precise with respect to ES.*

Example 5.46. Consider the Program 12 in Figure 5.2. The cost relation wh is not necessarily terminating if considered without a context. In particular, wh diverges if variable s is not strictly positive. However, if we consider that CR *strengthen* is its only entry, wh is always called with $s = 1$ and wh is guaranteed to terminate.

The chains of CR wh are $(3)^+(2)$, $(3)^\omega$ and (2). As it is, the phase $(3)^\omega$ cannot be proved terminating. The calling contexts of wh are $callCxt(\epsilon) = \{s = 1\}$ and $callCxt((3)^+) = \{s = 1, x + s > 0\}$. Therefore, the calling context of CE 3 is $callCxt(3) = \{s = 1, x + s > 0\} \sqcup \{s = 1\} = \{s = 1\}$ and if we apply Calling Context Strengthening to CE 3, we obtain:

$$3' : wh(x,s) = \{x > 0, x' = x - s, s = 1\},\ 1,\ wh(x',s)$$

Now the phase loop of $(3')^\omega$ has a ranking function x and the non-terminating chain can be discarded.

5.4 Refinement Propagation

Given a CR C, the refinement has split all its possible evaluations into a set of chains $ch_1, ch_2, \cdots, ch_n$. Each of the chains is either terminating or non-terminating and has a summary $summary(ch)$. Next, the refinement procedure uses these chains and summaries to specialize the calls to CR C from all the other cost relations that have not been refined yet.

Extension of Program 8

```
def List<A> drop(List<A> l, Int n)=
 if (n<=0) l else
  case l {
   | Nil => Nil;
   | Cons(h,t)=> drop(t,n-1);
  };
def Pair<List<A>,List<A>> split(List<A> l,Int n)=
 Pair(take(l,n),drop(l,n))
```

Cost relations

4: $drop(l, n : ret) = \{n \leq 0, ret = l\}$
5: $drop(l, n : ret) = \{n \geq 1, l = 0, ret = 0\}$
6: $drop(l, n : ret) = \{n > 0, l > 0, t = l - 1, n' = n - 1\}$, 1, $drop(t, n' : ret)$
7: $split(l, n : r1, r2) = take(l, n : r1), drop(l, n : r2)$

Figure 5.3.: Extension of Program 8: Example of refinement propagation

Definition 5.47 (Cost Equation Specialization)**.** Let C be a cost relation with a set of chains $CH = \{ch_1, ch_2, \cdots, ch_n\}$. Let $c\colon C'(\boldsymbol{x} : \boldsymbol{y}) = \varphi_1, b_1, \cdots, \varphi_i, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \cdots, b_n, \varphi_n \in C'$ be a cost equation that contains a call to CR C, the set of specialized CEs of c with respect to the call $C(\boldsymbol{x}_i : \boldsymbol{y}_i)$ is:

$$Z(c, C, i) := \left\{ \; c'\colon C'(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, b_1, \cdots, \varphi_i \wedge summary(ch), C[ch](\boldsymbol{x}_i : \boldsymbol{y}_i), \cdots, b_n, \varphi_n \;\middle|\; ch \in CH \; \right\}$$

where c' is a fresh identifier. The transformation generates $CRS' := CRS \setminus \{c\} \cup Z(c, C, i)$

Theorem 5.48. *The cost equation specialization is a sound and precise.*

Proof. The soundness and precision of this transformation derive directly from the completeness of refined chains (Theorem 5.25), the chain summary definition (Definition 5.26), and the fact that $summary(ch)$ returns a valid chain summary (Observation 5.29). □

The refinement specializes all the calls to CR C. This guarantees that when the refinement process reaches another C', all its non-recursive calls have been specialized and therefore Claim 5.1 holds.

Example 5.49. Figure 5.3 contains an extension of Program 8 where the power of the refinement propagation becomes apparent. Function *split* splits a list l in two parts in which the first part has length n (as long as l has n elements). It uses the function *drop* and *take* (in Figure 1.14, Chapter 1). The chains and summaries of *drop* are:

Chain	Summary	Chain	Summary
$(6)^+(4)$	$\{n \geq 1, l \geq n, n + ret = l\}$	(4)	$\{0 \geq n, ret = l\}$
$(6)^+(5)$	$\{l \geq 1, n > l, 0 = ret\}$	(5)	$\{l = 0, n \geq 1, ret = 0\}$

The chains and summaries of *take* are in Example 5.28. Given these chains and summaries, the complete CE specialization of CR *split* is the following:

7.1: $split(l, n : r1, r2) = \{n = r1 \geq 1, 0 \leq r2 = l - n\}, take[(3)^+(1)](l, n : r1), drop[(6)^+(4)](l, n : r2)$
7.2: $split(l, n : r1, r2) = \{n > l = r1 \geq 1, r2 = 0\}, take[(3)^+(2)](l, n : r1), drop[(6)^+(5)](l, n : r2)$
7.3: $split(l, n : r1, r2) = \{0 \geq n, r1 = 0, r2 = l\}, take[(1)](l, n : r1), drop[(4)](l, n : r2)$
7.4: $split(l, n : r1, r2) = \{n \geq 1, r1 = r2 = l = 0\}, take[(2)](l, n : r1), drop[(5)](l, n : r2)$

Each of the specialized CEs corresponds to a specific situation. For example CE 7.1 corresponds to the case where l has at least n elements and $n > 0$. In such a case we have that $r1 + r2 = l$ and $r1$ has n elements. On the other hand, CE 7.2 is the case where l has less than n elements. The result $r1$ contains all the elements of l and $r2$ is empty.

Many combinations of specialized calls have been discarded because they are incompatible (the conjunction of their summaries is unsatisfiable). For instance, the calls $take[(3)^+(1)]$ and $drop[(6)^+(5)]$ are not compatible because they contain the constraints $l \geq n$ and $n > l$ in their respective summaries.

5.5 Extension to Multiple Recursion

This section generalizes the refinement procedure to cost relations with multiple recursion. This section focuses on the differences. The refinement maintains the same structure but the control-flow refinement and the invariants need to be adapted. The refinement propagation remains unchanged.

5.5.1 Partially Refined Cost Equations

If a partially refined cost equation with one recursive call could have two constraint sets, a cost equation with n recursive calls can have $n + 1$ constraint sets. Partially refined CEs have the form:

$$c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \varphi_1, \boldsymbol{b}_1, C(\boldsymbol{x}_2 : \boldsymbol{y}_2), \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \varphi_n, \boldsymbol{b}_n$$

or

$$c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, \boldsymbol{b}_0$$

Note that in the first case all the calls in $\boldsymbol{b}_i$ (each $\boldsymbol{b}_i$ is a sequence of calls or linear expressions) are terminating except the last call of $\boldsymbol{b}_n$ which might be non-terminating and in the second case only the last call of $\boldsymbol{b}_0$ can be non-terminating. If a different call is non-terminating, Dead Code Elimination can be applied to truncate the cost equation at that point. As before, a CE whose last call in non-terminating is called tail-divergent.

5.5.2 Control-flow Refinement of a Cost Relation

The call relation is generalized for CEs with several recursive calls. A CE c calls d, written $c \to d$, if the joint constraints of c and d are satisfiable. The relation does not distinguish the position in which d can be called.

Definition 5.50 (CE Call Relation)**.** Let C be a cost relation and let $c, d \in C$, we have $c \to d$ if and only if given $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \varphi_1, \boldsymbol{b}_1, C(\boldsymbol{x}_2 : \boldsymbol{y}_2), \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \varphi_n, \boldsymbol{b}_n$ and $d\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi'_0, \cdots$, there is a k such that $\bigwedge_{i=0}^{k} \varphi_i \wedge \varphi'_0[\boldsymbol{xy}/\boldsymbol{x_i y_i}]$ is satisfiable.

As in the linear case, phases also correspond to SCCs of the call-graph but if a SCC contains CEs with more than one recursive call, a different kind of phase is generated.

Definition 5.51 (Multiple Phase)**.** Let S be the vertices of a SCC in $\mathcal{G}(C)$. If $S = \{c_1, \cdots, c_m\}$ is recursive, and there is some c_i with more than one recursive call, it generates an *iterative multiple phase* $ph := M(c_1 \vee \cdots \vee c_m)^{+/\omega}$. In this case, there is no distinction between iterative and divergent phases at this point. A phase $M(c_1 \vee \cdots \vee c_m)^{+/\omega}$ represents an evaluation in which every branch of the tree matches the pattern $(c_1 \vee \cdots \vee c_m)^+$ or $(c_1 \vee \cdots \vee c_m)^\omega$.

If $S = \{c\}$ cannot iterate ($c \not\to c$) and has more than one recursive call, it generates a *non-iterative multiple phase* $ph := M(c)$ in which c is evaluated once.

$1\colon append(l1, l2 : l) = \{l1 = 0, l = l2\}$
$2\colon append(l1, l2 : l) = \{l1 \geq 1, l1 = l1' + 1, l = l' + 1\},\ 1,\ append(l1', l2 : l')$

$3\colon subtr(t : l) = \{t = 0 = l\}$
$4\colon subtr(t : l) = \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0, l = l3 + 1\},\ 1,\ subtr(t1 : l1),\ subtr(t2 : l2),$
$\{l1 \geq 1, l1 + l2 = l3\},\ append[(2)^+(1)](l1, l2 : l3)$
$5\colon subtr(t : l) = \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0, l = l3 + 1\},\ 1,\ subtr(t1 : l1),\ subtr(t2 : l2),$
$\{l1 = 0, l2 = l3\},\ append[(1)](l1, l2 : l3)$

Figure 5.4.: Cost Relations of Program 4: Partially refined multiple recursion

Whereas a simple phase corresponds to a sequence of CE evaluations, a multiple phase represents a tree. In an evaluation, a simple phase can only be followed by another phase but a multiple phase can be followed by several other phases as each leaf of the tree can continue with calls to different phases.

The call relation $\rightarrow$ is extended for multiple phases similarly as with simple phases. Let ph_1 and ph_2 be phases generated from SCCs S_1 and S_2. We have $ph_1 \rightarrow ph_2$ if there is $c_1 \in ph_1$ and $c_2 \in ph_2$ and $c_1 \rightarrow c_2$ and $ph_1 \neq (c_1 \vee \cdots \vee c_m)^\omega$. Note that a multiple phase can call other phases even if some of its branches might diverge. Once the call relation has been lifted to phases, the notion of chain is extended taking into account multiple phases.

Definition 5.52 (Chain)**.** A *chain* is a *tree* of phases defined recursively:

$$\begin{array}{lll} ch := & ph & \text{if } ph = (c) \text{ and } c \text{ is not recursive, or } ph = (c_1 \vee \cdots \vee c_m)^\omega \\ & \mid ph \cdot ch' & \text{if } ph \text{ is not multiple, } ch' = ph'\_ \text{ and } ph \rightarrow ph' \\ & \mid ph \cdot CH & ph \text{ is multiple, and } CH := \{ch' \mid ch' = ph'\_ \wedge ph \rightarrow ph'\} \end{array}$$

Note that in the third case $ph \cdot CH$, the phase ph might have a branch that diverges.

Example 5.53. Figure 5.4 contains the cost relations of Program 4 (in Page 11) where the CEs of *subtr* are partially refined. Note for example that the constraint sets before the call to *append* constitute the summaries of the chains in *append*. The chains of CR *subtr* are $M(4 \vee 5)^{+/\omega}\{(3)\}$ and (3).

Chain Evaluations

The definitions of CR-paths (Definition 5.7) and chain evaluations (Definition 5.8) do not need to be adapted. The only difference is in the definition of the language $\mathscr{L}_{ch}$ that paths have to match. The language of a chain $ch = ph \cdot CH$) that starts with a multiple phase $ph = M(c_1 \vee \cdots \vee c_n)^{+/\omega}$ is:

$$\mathscr{L}_{ch} := \mathscr{L}_{(c_1 \vee \cdots \vee c_n)^\omega} \vee \left(\mathscr{L}_{(c_1 \vee \cdots \vee c_n)^+} \cdot \left(\bigvee_{ch \in CH} \mathscr{L}_{ch}\right)\right)$$

A CR-path in ch can be infinite over the CEs of ph or it can have a finite prefix that matches ph and a suffix that matches the language of one of the chains in CH.

Theorem 5.9 (Chain Refinement Completeness) is also valid. Below the proof is adapted to the case of multiple recursion.

Proof of Theorem 5.9 for Multiple Recursion. The same reasoning as in the original proof can be applied to show that given an evaluation T, for every $p \in paths_C(T)$ there is a chain ch such that $label(p) \in \mathscr{L}_{ch}$. However, in the case of a cost relation with multiple recursion, the number of CR-paths in T can be more than one. Therefore, we have to prove that given two different paths $p_1, p_2 \in paths_C(T)$, they match the same chain. That is, if $p_1 \in \mathscr{L}_{ch}$ and $p_2 \in \mathscr{L}_{ch'}$, then $ch = ch'$.

The paths p_1 and p_2 have a non-empty prefix in common because they both start at the root of the evaluation. Let b be the maximal common prefix, that is, $p_1 = b \cdot p_1'$ and $p_2 = b \cdot p_2'$. We have that

$p_1' \neq p_2'$ so there must be a node in $b = b_1 \cdot c(\boldsymbol{a} : \boldsymbol{b}) \cdot b_2$ whose CE c has multiple recursion, otherwise p_1' and p_2' cannot differ. A CE with multiple recursion generates only one multiple phase ph and there is only one chain that starts with such a phase $ph \cdot CH$. This is because CH is uniquely determined (see Definition 5.52). Therefore, the paths $c(\boldsymbol{a} : \boldsymbol{b}) \cdot b_2 \cdot p_1'$ and $c(\boldsymbol{a} : \boldsymbol{b}) \cdot b_2 \cdot p_2'$ belong to the same chain $ph \cdot CH$. Finally, the rest of the path b_1 is common to p_1 and p_2 so if $p_1 \in \mathscr{L}_{ch}$ then $p_2 \in \mathscr{L}_{ch}$ and vice-versa.

□

Discarding Divergent Phases

The evaluation of a multiple phase $M(c_1 \vee \cdots \vee c_m)^{+/\omega}$ might have one branch that follows the pattern $(c_1 \vee \cdots \vee c_m)^{\omega}$. If we prove such a pattern is unfeasible, we can transform the phase into $M(c_1 \vee \cdots \vee c_m)^{+}$.

This can be done similarly to how it is done for divergent phases. That is, a multi-path linear constraint loop is extracted from the phase. If that loop, denoted phase loop, is proved terminating, the divergent pattern $(c_1 \vee \cdots \vee c_m)^{\omega}$ is unfeasible. The only difference is that a CE in a multiple phase can have several recursive calls and consequently generate multiple paths. One path for each recursive call.

Theorem 5.54. *If the phase loop of a phase $ph = M(c_1 \vee \cdots \vee c_m)^{+/\omega}$ is terminating, then the phase is equivalent to $ph' = M(c_1 \vee \cdots \vee c_m)^{+}$. That is, let $ch = ph \cdot CH$ and $ch' = ph' \cdot CH$ be chains that start with ph and ph', their evaluations coincide $[\![CRS|C[ch]]\!] = [\![CRS|C[ch']]\!]$.*

Proof. We prove that if the phase loop of ph is terminating, then $[\![CRS|C[ch]]\!] = [\![CRS|C[ch']]\!]$. It is clear that $[\![CRS|C[ch']]\!] \subseteq [\![CRS|C[ch]]\!]$. Assume $[\![CRS|C[ch]]\!] \not\subseteq [\![CRS|C[ch']]\!]$, that is, there is an evaluation $T \in [\![CRS|C[ch]]\!]$ such that $T \notin [\![CRS|C[ch']]\!]$. The evaluation T has to have a path $p \in paths_C(T)$ such that $label(p) \in \mathscr{L}_{(c_1 \vee \cdots \vee c_m)^{\omega}}$. Such a path has to be infinite and formed only by nodes of $c_1, \cdots, c_m$. However, if the phase loop of ph is terminating, such an infinite path cannot exist. Consequently, there is no evaluation T such that $T \in [\![CRS|C[ch]]\!]$ and $T \notin [\![CRS|C[ch']]\!]$ and therefore $[\![CRS|C[ch]]\!] \subseteq [\![CRS|C[ch']]\!]$. □

Example 5.55. Continuing with Example 5.53, consider phase $M(4 \vee 5)^{+/\omega}$ of Program 4. CE 4 generates two loop paths, one for each recursive call. The loop paths are obtained by projecting its first constraint set $\{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0, l = l3 + 1\}$ onto $t, t1$ and $t, t2$, which generate $\{t \geq 1 + t1, t1 \geq 0\}$ and $\{t \geq 1 + t2, t2 \geq 0\}$. Then, we rename the variables of the recursive calls to primed versions of the head variables and obtain $\{t \geq 1 + t', t' \geq 0\}$ in both cases. CE 5 generates another two paths that are equal. Finally, the expression t is a valid ranking function of $\{t \geq 1 + t', t' \geq 0\}$ and $M(4 \vee 5)^{+/\omega}$ can be simplified to $M(4 \vee 5)^{+}$.

Refined Cost Equations

The CE Case Distinction is a generalization of the linear case. A CE with multiple recursive calls generates more than one partial CE, one per recursive call.

Definition 5.56 (Multiple Recursive CE Case Distinction)**.** Let a cost equation $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_0, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \varphi_1, \boldsymbol{b}_1, C(\boldsymbol{x}_2 : \boldsymbol{y}_2), \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \varphi_n, \boldsymbol{b}_n$ with n recursive calls, the case distinction generates n partial CEs c^{p_j} for $1 \leq j \leq n$:

$$c^{p_i}\colon C(\boldsymbol{x} : \boldsymbol{y}) = \bigwedge_{i=0}^{j-1} \varphi_i, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \cdots, C(\boldsymbol{x}_i : \boldsymbol{y}_i)$$

and a complete CE c^c or a tail-divergent CE c^{td}:

$$c^{td}\colon C(\boldsymbol{x} : \boldsymbol{y}) = \bigwedge_{i=0}^{n} \varphi_i, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, C(\boldsymbol{x}_2 : \boldsymbol{y}_2), \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n \quad \text{if } c \text{ is tail-divergent}$$

$$c^{c}\colon C(\boldsymbol{x} : \boldsymbol{y}) = \bigwedge_{i=0}^{n} \varphi_i, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, C(\boldsymbol{x}_2 : \boldsymbol{y}_2), \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n \quad \text{otherwise}$$

Theorem 5.57. *The multiple recursive CE case distinction is sound and precise.*

Proof. The proof is analogous to the one of Theorem 5.15. The function CE *sp* can be generalized to map each c to c^c, c^{td}, or some c^{p_i}. □

Example 5.58. Consider CE 4 of Program 4 (Figure 5.4):

$$4\colon subtr(t:l) = \{t = 1+t1+t2, t1 \geq 0, t2 \geq 0, l = l3+1\},\ 1,\ subtr(t1:l1),\ subtr(t2:l2),\\ \{l1 \geq 1, l1+l2 = l3\},\ append[(2)^+(1)](l1,l2:l3)$$

Its complete and partial versions are:

$$4^c\colon subtr(t:l) = \{t = 1+t1+t2, t1 \geq 0, t2 \geq 0, l = l3+1, l1 \geq 1, l1+l2 = l3\},\ 1,\\ subtr(t1:l1),\ subtr(t2:l2),\ append[(2)^+(1)](l1,l2:l3)$$
$$4^{p_1}\colon subtr(t:l) = \{t = 1+t1+t2, t1 \geq 0, t2 \geq 0, l = l3+1\},\ 1,\ subtr(t1:l1)$$
$$4^{p_2}\colon subtr(t:l) = \{t = 1+t1+t2, t1 \geq 0, t2 \geq 0, l = l3+1\},\ 1,\ subtr(t1:l1),\ subtr(t2:l2)$$

Refined Chains

Similarly, the phase case distinction and the refined chains are extended for multiple phases. For a multiple phase, only two phases are generated, a complete phase and a divergent phase.

Definition 5.59 (Multiple Phase Case Distinction)**.** Let $ph = M(c_1 \vee \cdots \vee c_m)^{+/\omega}$ be a multiple phase, such that $c_1, \cdots, c_k$ are tail-divergent and each c_i has $\#calls(c_i)$ recursive calls, the following phases are generated:

1. A complete phase $ph^c := M(c^c_{k+1} \vee \cdots \vee c^c_m)^+$ where the phase is completely evaluated.

2. A divergent phase

$$ph^d := M\left(\bigvee_{i=1}^{k}\left(c_i^{td} \vee \bigvee_{j=1}^{\#calls(c_i)} c_i^{p_j}\right) \vee \bigvee_{i=k+1}^{m}\left(c_i^c \quad \vee \quad \bigvee_{j=1}^{\#calls(c_i)} c_i^{p_j}\right)\right)^{+/\omega}$$

   where the evaluation diverges. Basically, the divergent phase contains all the refined CEs of the original phase. If non-termination has been discarded within the phase, then ph^d has the superscript $+$ instead of $+/\omega$.

The non-iterative case is again a particular case. Let $ph = M(c)$ a phase $ph^d := (c^{p_j})$ is generated for each c^{p_j}. If c is tail-divergent $ph^d := (c^{td})$ is generated, otherwise $ph^c := (c^c)$ is generated.

Note that for multiple phases, no distinction is made between divergent and tail-divergent. Once the case distinction for multiple phases has been defined, the refined chains (Definition 5.21) can be extended to take these phases into account.

Definition 5.60 (Refined Chains)**.** The definitions of the functions *term* and *nterm* are extended to support chains with multiple phases:

$$term(ch) := \begin{cases} ph^c & \text{if } ch = ph \\ ph^c \cdot ch^{c'} & \text{if } ch = ph \cdot ch' \text{ and } ch^{c'} = term(ch') \\ ph^c \cdot CH' & \text{if } ch = ph \cdot CH \text{ and } CH' := \{term(ch') \mid ch' \in CH\} \end{cases}$$

$$nterm(ch) := \begin{cases} \{ph^d\} & \text{if } ch = ph \wedge\ ph \text{ is divergent} \\ \{ph^{td}\} & \text{if } ch = ph \wedge ph = (c) \\ \{ph^{td} \cdot ch^{c'} \mid ch^{c'} = term(ch')\} & \\ \cup\{ph^p \cdot ch^{\omega'} \mid ch^{\omega'} \in nterm(ch')\} & \text{if } ch = ph \cdot ch' \\ \{ph^d \cdot CH'\} & \text{if } ch = ph \cdot CH \wedge\ CH' = term(CH) \cup nterm(CH) \wedge (*) \end{cases}$$

where (*) is one of the following conditions:

1. $nterm(CH)$ is not empty.
2. ph is of the form $M(c_1 \vee \cdots \vee c_m)^{+/\omega}$.
3. or ph contains at least one tail-divergent CE.

Example 5.61. The initial chains of CR *subtr* are $M(4 \vee 5)^+\{(3)\}$ and (3) (see Examples 5.53 and 5.55). The terminating chains of *subtr* are $M(4^c \vee 5^c)^+\{(3^c)\}$ and (3^c). CR *subtr* does not have any non-terminating chain because we have discarded the only divergent path (Example 5.55) and *subtr* does not have tail-divergent CEs. Consequently, the partial versions of CEs do not appear in any chain and they do not need to be considered.

Theorem 5.25 (Refined Chains Completeness) remains unchanged but the proof has to be extended to consider chains with multiple recursion.

Proof of Theorem 5.25 for Multiple Recursion. The proof for finite evaluations (Equation 5.3) can be applied directly for chains with multiple recursion considering the updated version of the function *sp* because the reasoning does not rely on the structure of the chains.

For infinite evaluations (Equation 5.4). Let $T' \in [\![CRS'|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$, we know that there is a $T \in [\![CRS|C(\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ such that $sp(T) = T'$. For T, there is a chain ch such that $T \in [\![CRS|C[ch](\boldsymbol{a} : \boldsymbol{b})]\!]_\omega$ (because of Theorem 5.9). Therefore, it is enough to prove that all $p' \in paths_C(sp(T))$ match a chain in $nterm(ch)$. The original proof was done by induction on the structure of the chain and it can be reused. However, there is an additional case in the induction step, when $ch = ph \cdot CH$, that needs to be considered.

- If the phase is iterative, there is only one chain $ch^\omega = ph^d \cdot CH'$ that can be generated such that $CH' := term(CH) \cup nterm(CH)$. If T is infinite, at least one of the three conditions from *nterm*'s definition is guaranteed to hold so ch^ω is indeed generated.

  Let $p \in paths_C(T)$, we have to prove that the corresponding path $p' \in paths_C(sp(T))$ matches ch^ω. If p can be split into two sub-paths $p = p_1 \cdot p_2$ such that p_1 matches ph and p_2 matches $ch' \in CH$. The path p' can be split at the same point $p' = p'_1 \cdot p'_2$ and by induction hypothesis p'_2 matches a chain in CH'. The phase ph^d contains all the partial, complete, and tail-divergent CEs of ph so p'_1 has to match ph^d and $label(p') \in \mathscr{L}_{ch^\omega}$. If p cannot be split, then p matches ph completely and p' matches ph^d. Therefore, $label(p') \in \mathscr{L}_{ch^\omega}$.

- If the phase is not iterative $ph = (c)$, there is one refined chain $ch^\omega{}_i := c^{p_i} \cdot CH'$ for each partial CE c^{p_i} and one $ch^\omega := c^{td} \cdot CH'$ if c is tail-divergent. We prove that all the paths in $sp(T)$ match a single $ch^\omega{}_i$ or match ch^ω. Every path $p \in paths_C(T)$ has the form $p = c(\boldsymbol{a} : \boldsymbol{b}) \cdot p_1$ where $c(\boldsymbol{a} : \boldsymbol{b})$ is

the node of the root of the evaluation T (common to all paths) and p_1 matches a chain $ch' \in CH$. Similarly, every $p' \in paths_C(sp(T))$ has the form $p' = X(\boldsymbol{a} : \boldsymbol{b}) \cdot p'_1$ where X is c^{p_i} for some i or c^{td} but the same for all paths. By induction hypothesis p'_1 matches a chain in CH'. Therefore, all $p' \in paths_C(sp(T))$ match a single chain $X \cdot CH'$ which can be ${ch^\omega}_i$ or ch^ω.

□

5.5.3 Invariants

Chain Summaries

The definitions or chain summaries and calling contexts (Definition 5.26 and Definition 5.32) remain unaffected. The operations for computing chain summaries and calling contexts are extended to cost equations with multiple recursion. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n$ be a refined CE, the operation $\tau^{-1}(c, \varphi)$ is defined as follows:

$$\tau^{-1}(c, \varphi) = \left(\bigwedge_{i=1}^{n} \varphi(\boldsymbol{x}_i \boldsymbol{y}_i) \wedge \varphi_c \right) \downarrow \boldsymbol{x}\boldsymbol{y}$$

The extended summary computation is:

$$summary(ch) = \begin{cases} \bigsqcup \{\ \varphi \downarrow \boldsymbol{x}\boldsymbol{y} \mid c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi \cdots \in ph\ \} & ch = ph \text{ or } ch = ph^d \cdot CH \\ \tau^{-1}(ph, summary(ch')) & ch = ph \cdot ch' \\ \tau^{-1}(ph^c, \bigsqcup \{summary(ch') \mid ch' \in CH\}) & ch = ph^c \cdot CH \end{cases}$$

The first case includes the chains that have only one phase or the chains whose first phase is a divergent multiple phase. In these cases the summary only considers that one CE is evaluated which is a sound over-approximation. The second case corresponds to a previously existing case, the propagation of a summary through a single phase. Finally, the third case corresponds to the propagation of a summary through a multiple phase that can be iterative or not. It is computed by applying the pattern of the phase to the initial summary. Such initial summary is the convex hull of all the chain summaries that follow the phase ($ch' \in CH$).

Calling Contexts

For calling contexts, the definition of chain prefixes has to be extended for the new chains.

Definition 5.62 (Chain prefix)**.** Given a chain ch, ϵ is always a prefix of ch. In addition, we have:

- If $ch = (c_1 \vee \cdots \vee c_n)^\omega$, then $pfx := (c_1 \vee \cdots \vee c_n)^+$ is a prefix of ch.
- If $ch = ph \cdot ch'$ and pfx' is a prefix of ch', $pfx := pfx' \cdot ph$ is a prefix of ch.
- If $ch = M(c_1 \vee \cdots \vee c_n)^{+/\omega} \cdot CH'$ or $ch = M(c_1 \vee \cdots \vee c_n)^+ \cdot CH'$ and pfx' is a prefix of $ch' \in CH'$, then $pfx := pfx' \cdot (c_1 \vee \cdots \vee c_n)^+$ is a prefix of ch.

Example 5.63. Consider a chain of the form $M(a \vee b)^+\{\ (c)^+(e),\ \ (c)^+(d)^\omega\ \}$, its prefixes are:

$$\epsilon \qquad (a \vee b)^+ \qquad (c)^+(a \vee b)^+ \qquad (d)^+(c)^+(a \vee b)^+$$

The operation τ^1 is generalized to CEs with several recursive calls. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n$ be a refined CE:

$$\tau^1(c, \varphi) = \bigsqcup \{\ ((\varphi(\boldsymbol{x}) \wedge \varphi_c) \downarrow \boldsymbol{x}_i)[\boldsymbol{x}_i / \boldsymbol{x}] \mid 1 \leq i \leq n\ \}$$

This definition considers the convex hull of any of the paths from the head to a recursive call and returns a constraint set in terms of $\boldsymbol{x}$. The lifting to other patterns and the computation of calling contexts remains the same. Note that chain prefixes have the same shape in the case of chains with multiple phases. This is because a prefix always represents a single path and not a tree.

Discarding Unfeasible Chains

A chain ch that contains multiple phases can still be discarded if its summary is unsatisfiable. However, any chain ch' that has ch as a suffix may not be always discarded. Instead, it might be simplified. For example, consider a chain $ch := ph \cdot \{ch_1, ch_2\}$. If $summary(ch_1)$ is unsatisfiable, ch cannot be discarded but it can be simplified to $ph \cdot \{ch_2\}$. Even if $summary(ch_2)$ is unsatisfiable, the chain $ph \cdot \{\}$ might be feasible if ph is divergent.

Similarly, if a prefix of ch is unsatisfiable, it might not be possible to discard the chain, but it can be simplified. For example, consider a chain $ch := ph \cdot \{ph_2 \cdot ph_3, ch_2\}$. The sequence $ph_2 \cdot ph$ is a prefix of ch. If $callCxt(ph_2 \cdot ph)$ is unsatisfiable, the chain can be simplified to $ch := ph \cdot \{ch_2\}$.

Strengthening Cost Equations

Both cost equations strengthenings (Definition 5.40 and Definition 5.44) can be applied in cost relations with multiple recursion. The calling context strengthening can be directly applied. For the call summary strengthening, the definition of cost equation summary (Definition 5.38) has to be adapted to also consider the new type of chains.

Definition 5.64 (Cost Equation Summary)**.** Let C be a cost relation with a set of chains CH. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi, \boldsymbol{b}, C(\boldsymbol{x}' : \boldsymbol{y}'), \boldsymbol{b}$ be a recursive CE, its summary is:

$$summary(c) = \bigsqcup \left\{ summary(ch)(\boldsymbol{x'y'}) \;\middle|\; \begin{array}{l} (ph \cdot ch \in CH \wedge c \in ph) \vee (ch = ph\_ \in CH \wedge c \in ph) \\ \vee (ph \cdot CH' \in CH \wedge c \in ph \wedge ch \in CH') \end{array} \right\}$$

The strengthening incorporates the CE summary in each of the recursive calls.

Definition 5.65 (Call Summary Strengthening)**.** For a multiple recursive CE $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c, \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n$, the strengthened CE is

$$c'\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_c \wedge \bigwedge_{i=1}^{n} (summary(c)(\boldsymbol{x_i y_i})), \boldsymbol{b}_0, C(\boldsymbol{x}_1 : \boldsymbol{y}_1), \boldsymbol{b}_1, \cdots, C(\boldsymbol{x}_n : \boldsymbol{y}_n), \boldsymbol{b}_n$$

The transformation generates $CRS' = CRS \setminus \{c\} \cup \{c'\}$.

Example 5.66. Call summary strengthening is important to infer bounds in CRs that are not tail recursive. For example, in CR *subtr* from Program 4 we have the chain $M(4^c \vee 5^c)^+\{(3^c)\}$ where CE 4^c is:

$$\begin{array}{rl} 4^c\colon subtr(t : l) = & \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0, l = l3 + 1, l1 \geq 1, l1 + l2 = l3\},\ 1, \\ & subtr(t1 : l1),\ subtr(t2 : l2),\ append[(2)^+(1)](l1, l2 : l3) \end{array}$$

The cost of the call to *append* depends on the variables $l1$ and $l2$ but these are not related to the input variables of *subtr*. Call summary strengthening tackles this issue. The chain summaries of $M(4^c \vee 5^c)^+(\{(3^c)\})$ and (3^c) are respectively $\{t = l, t \geq 1\}$ and $\{t = 0 = l\}$. The CE summary of CE 3^c is the convex hull of both summaries $\{t = l, t \geq 0\}$ and the strengthened CE is:

$$\begin{array}{rl} 4^c\colon subtr(t : l) = & \{t = 1 + t1 + t2, t1 \geq 0, t2 \geq 0, l = l3 + 1, l1 \geq 1, l1 + l2 = l3, \underline{t1 = l1, t2 = l2}\}, 1, \\ & subtr(t1 : l1),\ subtr(t2 : l2),\ append[(2)^+(1)](l1, l2 : l3) \end{array}$$

In this CE, the bound of *append*, expressed in terms of $l1$ or $l2$, can be expressed in terms of t.

6 Bound Computation

In the previous chapter, it has been shown how to refine a cost relation system and split the evaluations of each cost relation into chains. Once the refinement is complete, a cost relation system is formed by a sequence of cost relations $\langle C_1, C_2, \cdots, C_n \rangle$ that are topologically sorted. Each cost relation is formed by a set of chains that are in turn formed by phases. A phase represents a pattern in which a set of cost equations is evaluated. The cost equations in this representation have also been refined.

A refined cost equation (Definition 5.14) has the following characteristics:

- All its body is evaluated, i.e. there is no intermediate call that diverges
- It contains a single constraint set at the beginning
- All its calls are either direct recursive calls or calls to chains of other cost relations

This chapter describes a sound algorithm to obtain net upper and lower cost bounds of cost relations in a refined cost relation system and peak cost bounds if the *CRS* is cumulative (with no negative annotations). Throughout the chapter, a fixed cost relation system *CRS* is assumed (it will not be modified further) and the simplified notation $[\![C[ch]]\!]$ is used to refer to evaluations $[\![CRS|C[ch]]\!]$ in such a cost relation system.

Example 6.1. Figure 6.1 contains Program 13 and its refined cost relations. There are 5 cost relations: p, $wh3$, $wh6$, $wh10$ and $wh13$. One for the function `p` and one for each while loop located at lines 3, 6, 10 and 13. The cost model of Program 13 is given by the `tick` annotations. The precondition in the first line of the code has been incorporated into CR p's constraint sets. This example is interesting because it contains several aspects that are challenging for bound computation analysis:

- It contains two pairs of nested loops. The first pair $wh3$ and $wh6$ presents amortized cost. Taken individually, the cost of entering loop $wh6$ once is at most $2(x+y)$ (in terms of $p1$'s input parameters). The loop can be entered x times and still its total cost is at most $2(x+y)$ and not $2(x+y)x$ as one might expect.
- The second pair of nested loops $wh10$ and $wh13$ presents a more typical pattern that gives rise to a quadratic cost $\|y\| \cdot \|z\|$.
- Finally, the sequential composition of the two outer loops $wh3$ and $wh10$ also presents amortized cost. This is especially important for lower bounds. Considered individually, the cost of $wh3$ can be 0 (if no iterations of the inner loop $wh6$ are executed) and the cost of $wh10$ can also be 0 (if the inner loop $wh6$ iterates until y reaches 0). However, the lower cost bound of $wh3$ followed by $wh10$ is $\min(2,z)(x+y)$.

The bound inference has two defining characteristics. First, the analysis is incremental at several levels. It starts form the last cost relation C_n and proceeds backward until it reaches the first cost relation C_1. In each cost relation, it uses the previously computed results and composes them to compute the bounds of the current CR. Within a cost relation, the analysis also works incrementally:

1. First, it computes costs for each cost equation without taking the recursive calls into account
2. Then, it composes those costs to form the cost for the phases
3. Finally, the cost of the phases are composed to obtain the overall cost of the chains

Program 13	Refined Cost relations
1 $\{x>0, y>0, z>0\}$	1.1: $p(x,y,z) = \{x>0, y>0, z>0, x_o=0, y_o\leq 0\}$, $wh3[(3.1 \vee 3.2)^+(2)](x, y:x_o, y_o)$, $wh10[(6)](y_o, z)$
2 `void p(int x,y,z){`	1.2: $p(x,y,z) = \{x>0, y>0, z>0, x_o=0, x+y\geq y_o>0\}$, $wh3[(3.1 \vee 3.2)^+(2)](x, y:x_o, y_o)$, $wh10[(7)^+(6)](y_o, z)$
3 `while(x>0) {`	
4 `x--;`	
5 `y++;`	2: $wh3(x, y:x_o, y_o) = \{x=x_o=0, y_o=y\}$
6 `while(y>0 && *){`	3.1: $wh3(x, y:x_o, y_o) = \{x'+1=x>0, y_2=y_1=y+1\}$, $wh6[(4)](y_1, y_2)$, $wh3(x', y_2:x_o, y_o)$
7 `y--;tick(2);`	
8 `}`	3.2: $wh3(x, y:x_o, y_o) = \{x'+1=x>0, y_2<y_1=y+1\}$, $wh6[(5)^+(4)](y_1:y_2)$, $wh3(x', y_2:x_o, y_o)$
9 `}`	
10 `while(y>0){`	4: $wh6(y:y_o) = \{y=y_o\}$
11 `y--;`	5: $wh6(y:y_o) = \{y\geq 1, y'=y-1\}$, 2, $wh6(y':y_o)$
12 `int i=0;`	
13 `while(i<z){`	6: $wh10(y,z) = \{y\leq 0\}$
14 `i++;tick(1);`	7: $wh10(y,z) = \{y\geq 1, y'=y-1, z>0, i=0\}$, $wh13[(9)^+(8)](i,z)$, $wh10(y',z)$
15 `}`	
16 `}`	8: $wh13(i,z) = \{i\geq z\}$
17 `}`	9: $wh13(i,z) = \{i<z, i'=i+1\}$, 1, $wh13(i',z)$

Figure 6.1.: Program 13: Running example for bound computation

At each step, the analysis only uses local information of the corresponding CE, phase or chain that is being processed.

Example 6.2. Figure 6.2 presents an evaluation of Program 13 in which the costs of some CEs, phases and chains are demarcated (in purple, blue and green respectively). The cost relation symbol has been included in each evaluation node to ease readability. The cost of an evaluation of CE 9 corresponds to the cost of its components without taking the recursive calls into account. The cost of an evaluation of the phase $(9)^+$ corresponds to the sum of the cost of the CE evaluations within the phase. Finally, the cost of an evaluation of the chain $(9)^+(8)$ is the sum of the costs of its phases $(9)^+$ and (8).

Similarly, the cost of an evaluation of CE 3.2 is the cost of its elements without taking its recursive call into account. This corresponds to the call to chain $wh6[(5)^+(4)]$. The cost of the phase $(3.1 \vee 3.2)^+$ is the sum of the cost of all the evaluations of CEs 3.1 and 3.2 in the phase.

The objective is not to compute bounds for specific evaluations but to obtain a symbolic representation of the cost of all the evaluations of a chain, phase or CE. The second defining characteristic of the analysis is the representation of these symbolic bounds. Instead of representing a bound as a simple symbolic expression, the cost of the evaluations is represented with *cost structures*. Cost structures are a data structure that can represent the costs of all the evaluations of a chain, phase or CE. Thus, a bound of a cost structure is also a bound of the corresponding chain, phase or CE. The bounds of all the chains of a cost relation can be later combined into a bound of the cost relation.

Note that thanks to the refinement, terminating and non-terminating evaluations can be considered independently. In fact, non-terminating evaluations might have been completely discarded. In that case, the algorithm can attempt to obtain net cost bounds. If non-terminating evaluations have not been discarded but the cost relation system is cumulative (with only positive costs), the algorithm can still attempt to obtain peak upper bounds for the non-terminating evaluations.

The next section contains a result that enables the analysis to obtain peak upper bounds of infinite evaluations by considering only a subset of all the possible partial evaluations. Section 6.2 provides the formal definitions of the cost of CEs and phases and how they can be composed. Section 6.3 contains

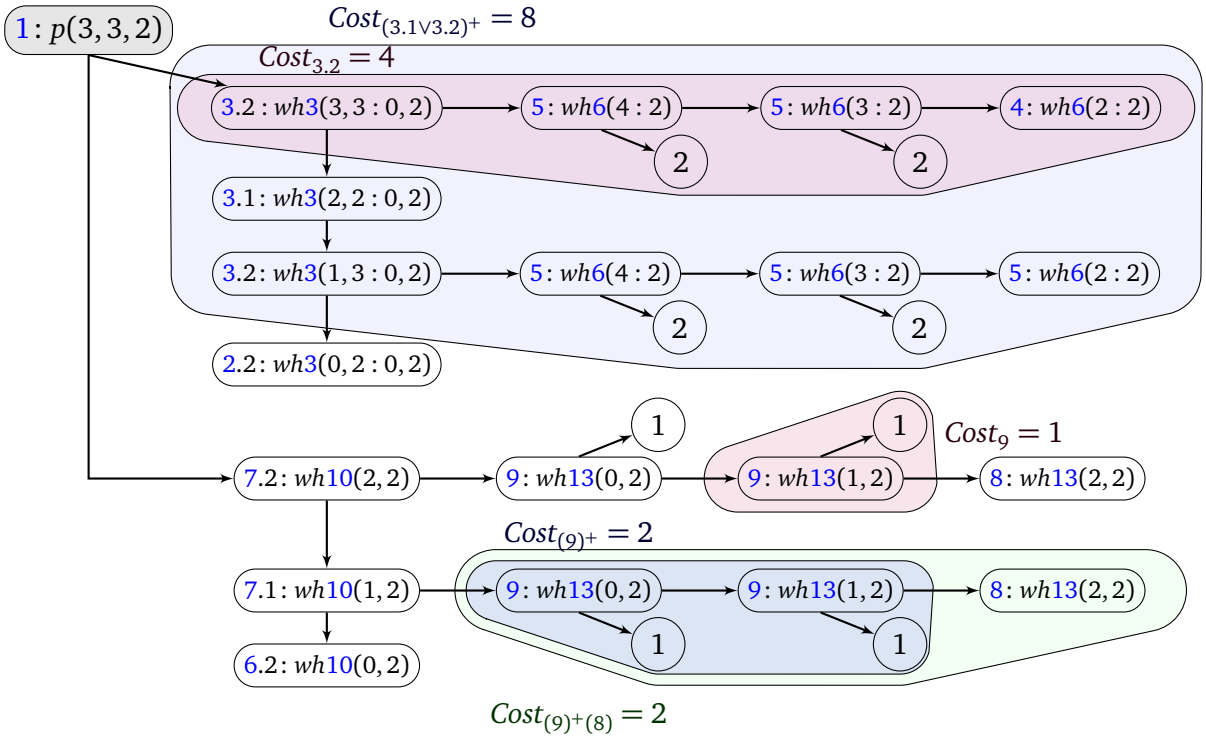


Figure 6.2.: Evaluation of Program 13 and some of its costs

the definition of cost structures and a description of their main characteristics. Sections 6.4, 6.5 and 6.6 specify how to infer cost structures of CE, phases, and chains respectively with only linear recursion. Section 6.7 extends the cost structure inference techniques to cost relations with multiple recursion. Sections 6.8 and 6.9 specify how to obtain closed-form bounds from cost structures and how to combine chain bounds to obtain piece-wise defined bounds for cost relations. Finally, Section 6.10 contains the soundness proofs for the strategies used in the inference process.

6.1 Infinite Evaluations of Cumulative *CRS*

As mentioned in Chapter 3, only peak costs are well-defined for infinite evaluations. In general, it can be challenging to obtain a peak cost upper bound because such an upper bound has to be bigger than the cost of every partial evaluation. This means that a great amount of possibilities have to be considered. In practice though, it is enough to consider a reduced set of partial evaluations whose cost is maximal.

This reduced set of partial evaluations that have maximal cost is easy to obtain if the cost relation system has cumulative cost. This is because in such a *CRS*, the cost can only increase. In a complete evaluation, the net cost is also the peak cost. In an infinite evaluation, it is enough to consider partial evaluations that are truncated along the infinite branch of the evaluation tree.

At this point, we can use the information obtained during the refinement to choose a subset of the partial evaluations that simplifies the reasoning. We know that every infinite evaluation T eventually reaches a divergent phase (that can be in a different CR) so we can consider only partial evaluations that stop at the recursive calls within divergent phases. As a result, it is sound to assume all the other phases and chains are completely evaluated (and compute only their net cost) and only evaluations in divergent phases might be truncated.

Definition 6.3 (Selected Partial Evaluations)**.** Let C be a cost relation and ch^ω a non-terminating chain, we define:

$$[\![C[ch^\omega]]\!]_f = \left\{ \downarrow_\pi (T) \;\middle|\; \begin{array}{l} T \in [\![C[ch^\omega]]\!]_\omega \text{there is a } \pi \text{ such that } T_{|\pi} \in [\![C'[ph^d]]\!]_\omega \text{ or} \\ T_{|\pi} \in [\![C'[ph^d \cdot CH]]\!]_\omega \end{array} \right\}$$

We denote $[\![C]\!]_{fc} := [\![C]\!]_c \cup [\![C]\!]_f$ the union of all complete evaluations and selected partial evaluations of CR C.

Now we need two results to be able to rely on these evaluation sets to compute bounds. First, we need to prove that the evaluations in these sets are closed under tree composition. This way, if we have valid cost for each of the subtrees of an evaluation T, these costs can be composed into the cost of T.

Proposition 6.4. *If $T \in [\![C[ch^\omega]]\!]_f$, then any subtree of T is also in $[\![C'[ch^{\omega'}]]\!]_f$ for some CR C' and some chain $ch^{\omega'}$ or in $[\![C'[ch^{c'}]]\!]_c$ for some CR C' and some chain $ch^{c'}$.*

Proof. Let $T \in [\![C[ch^\omega]]\!]_f$ with $T = t(r, [T_1, \cdots, T_{n-1}, T_n])$. By definition T is a partial evaluation of a $T' \in [\![C[ch^\omega]]\!]_\omega$ with a π such that $\downarrow_\pi (T') = T$. The position π has been selected to truncate the infinite branch. The infinite branch has to be the last one so π has to be of the form $\pi = n \cdot \pi'$ and T' has the form $T' = t(r, [T_1, \cdots, T_{n-1}, T'_n])$. Therefore, $T_i \in [\![C_i[ch^c{}_i]]\!]_c$ for some CR C_i and chain $ch^c{}_i$ for $1 \leq i < n$.

We still have to prove that $T_n \in [\![C'[ch^{\omega'}]]\!]_f$ for some CR C' and chain $ch^{\omega'}$. We also know that the non-truncated subtree $T'_n \in [\![C'[ch^{\omega'}]]\!]_\omega$ and if we truncate it with π' we obtain T_n: $\downarrow_{\pi'} (T'_n) = T_n$. By Definition 6.3, we know that $T'_{|\pi} \in [\![C''[ph^d]]\!]_\omega$ or $T'_{|\pi} \in [\![C''[ph^d \cdot CH]]\!]_\omega$ (for some CR C'') and $T'_{|\pi} = (T'_n)_{|\pi'}$. That implies that $\downarrow_{\pi'} (T'_n)$, which is T_n, is also in the set of selected partial evaluations $[\![C'[ch^{\omega'}]]\!]_f$. □

Second, any upper bound inferred for the selected partial evaluations of a chain in a cumulative *CRS* is indeed a peak upper bound of that chain.

Proposition 6.5. *Let CRS be a cumulative cost relation system and let $f(\boldsymbol{x}) : \mathbb{Q}^n \to \mathbb{Q} \cup \{\omega, -\omega\}$ be a function such that $f(\boldsymbol{a}) \geq Cost(T)$ for every $T \in [\![CRS|C[ch^\omega](\boldsymbol{a} : \boldsymbol{b})]\!]_f$ then f is a peak upper bound of $C[ch^\omega]$.*

Proof. It is enough to show that for any partial evaluation $\downarrow_\pi (T)$ there is a selected partial evaluation $\downarrow_{\pi'} (T) \in [\![CRS|C[ch^\omega]]\!]_f$ such that $Cost(\downarrow_{\pi'} (T)) \geq Cost(\downarrow_\pi (T))$. Because the *CRS* is cumulative, the cost of a bigger portion of the evaluation tree is always greater or equal than a smaller portion. In terms of positions we have that $Cost(\downarrow_{\pi'} (T)) \geq Cost(\downarrow_\pi (T))$ if $\pi' \geq_{lex} \pi$ where $\geq_{lex}$ is the lexicographical order on positions.

Given that the positions considered for $[\![CRS|C[ch^\omega]]\!]_f$ are rightmost positions (they truncate the tree on the infinite branch which is always the last branch) and they can be arbitrarily long (once the divergent phase is reached), for every $\downarrow_\pi (T)$ there is always a $\downarrow_{\pi'} (T)$ such that $\pi' \geq_{lex} \pi$ and $\downarrow_{\pi'} (T) \in [\![CRS|C[ch^\omega]]\!]_f$. □

From this point on, the bound computation procedure focuses on obtaining bounds that are valid for any evaluation in $[\![C]\!]_{fc}$ for the different chains.

6.2 Cost Definitions and Cost Composition

In order to achieve additional incrementality, the costs of CEs and phases are defined and how they can be composed.

Definition 6.6 (CE Cost)**.** Let $c\colon C(\boldsymbol{x}:\boldsymbol{y}) = \varphi, b_1, \cdots, b_n$ and let $T \in [\![C[ch]]\!]_{fc}$ be an evaluation of CE c with the form $T := t(c(\boldsymbol{a}:\boldsymbol{b}), [T_1, \cdots, T_n])$ (in any chain ch). The cost of CE c is the cost of the evaluation without taking the recursive calls into account

$$Cost_c(T) = \sum_{\{1 \le i \le n \mid b_i \neq C(\boldsymbol{x_i}:\boldsymbol{y_i})\}} Cost(T_i)$$

Note that we are considering evaluations in $T \in [\![C]\!]_{fc}$ which are finite and $Cost(T)$ is always well-defined.

Definition 6.7 (Phase Cost)**.** Let ch be a chain, let ph be its initial phase and $T \in [\![C[ch]]\!]_{fc}$ be an evaluation of the chain ch. Let $CEinst(T,c) = \{\pi \mid label(T_{|\pi}) = c\}$ be the instances of evaluations of CE c in T. The cost of the phase ph is the sum of the CE costs of the instances of CE evaluations for each $c \in ph$.

$$Cost_{ph}(T) = \sum_{c \in ph} \Big(\sum_{\pi \in CEinst(T,c)} Cost_c(T_{|\pi}) \Big)$$

Based on these definitions, we can determine how to compose the cost of a chain evaluation. For that purpose, we make use of the concept of maximal evaluation (Definition 5.31) adapted to chain evaluations. Let T be an evaluation, a sub-evaluation $T' \preceq T$ is maximal evaluation of ch if T' is an evaluation of ch and no ancestor of T' in T is an evaluation of ch.

Theorem 6.8 (Cost Composition)**.** *Let $T \in [\![C[ch]]\!]_{fc}$ be an evaluation of $ch = ph \cdot CH$. Let $maxCH(T, ch') = \{\pi \mid T_{|\pi}$ is a maximal evaluation of $ch'\}$ for each $ch' \in CH$. The cost of T can be expressed as*

$$Cost(T) = Cost_{ph}(T) + \sum_{ch' \in CH} \Big(\sum_{\pi \in maxCH(T,ch')} Cost(T_{|\pi}) \Big)$$

The cases of iterative phases with linear recursion and divergent phases can be seen as a special case of this theorem.

Corollary 6.9. *Let $T \in [\![C[ch]]\!]_{fc}$ such that $ch = ph \cdot ch'$. Then $maxCH(T, ch') = \{T'\}$ contains a single element and the cost of T is*

$$Cost(T) = Cost_{ph}(T) + Cost(T')$$

If $ch := ph$, maxCH is empty and the cost of T is

$$Cost(T) = Cost_{ph}(T)$$

This theorem and its corresponding corollary are the base to achieve incrementallity within a cost relation. The proof of Theorem 6.8 can be found in Section 6.10.

6.3 Cost Structures

In order to obtain upper and lower bounds, the analysis uses a symbolic cost representation that can represent the costs of chains, phases or CEs. This representation is called *cost structure*.

Cost structures represent combinations of linear expressions in such a way that they can be inferred and composed by merely solving problems over sets of linear constraints. Instead of a single complex expression, cost structures contain a simple linear cost expression E over *intermediate variables* (*iv*) and constraints that bind the intermediate variables to the variables of the CRs. They contain two kinds of constraints. *Non-final* constraints *IC* that relate intermediate variables among each other and

Chain/Phase/CE(Variables): Cost Structure

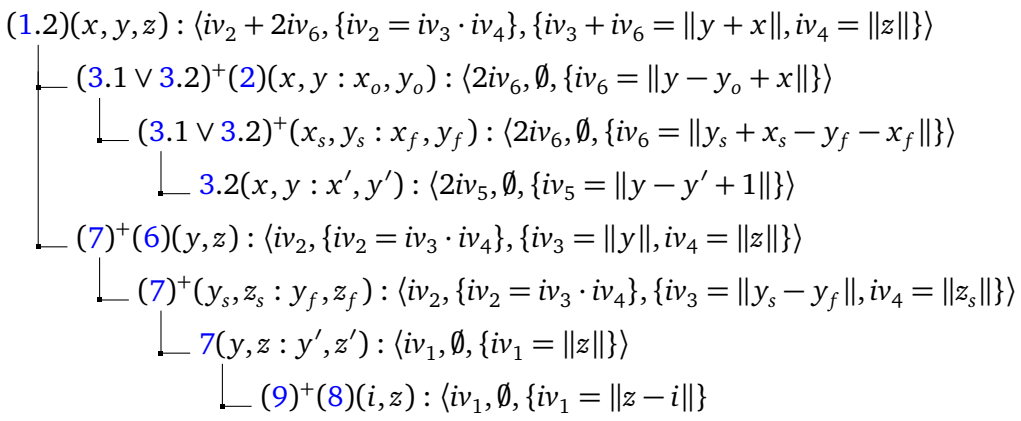


Figure 6.3.: Some cost structures of Program 13

final constraints $FC(\boldsymbol{x})$ that relate intermediate variables with the variables of the CRs ($\boldsymbol{x}$). The formal definition of cost structures is as follows:

Definition 6.10 (Cost Structure)**.** A *cost structure* is a tuple $\langle E, IC, FC(\boldsymbol{x})\rangle$.

- E is the main cost expression and is a linear expression $l(\boldsymbol{iv})$ over intermediate variables. Intermediate variables always represent non-negative numbers.
- Let $\bowtie$ be $\leq$ or $\geq$, IC is a set of *non-final* constraints of the form $ic := \sum \boldsymbol{iv} \bowtie SE$ where SE can be $SE := \quad l(\boldsymbol{iv}) \mid iv_i \cdot iv_j \mid \max(\boldsymbol{iv}) \mid \min(\boldsymbol{iv})$.
- $FC(\boldsymbol{x})$ is a set of *final* constraints of the form $fc := \sum \boldsymbol{iv} \bowtie \|l(\boldsymbol{x})\|$ where $l(\boldsymbol{x})$ is a linear expression over the CR variables and $\|l(\boldsymbol{x})\| := \max(l(\boldsymbol{x}), 0)$.

Even though the constraints in IC and $FC(\boldsymbol{x})$ are relatively simple, they can be combined to express complex polynomial expressions. Figure 6.3 contains some of the cost structures of Program 13 (in Page 76) that are obtained in the following sections ($a = b$ stands for $a \leq b$ and $a \geq b$). Thanks to the constraints a single cost structure can represent the range of all possible costs of a chain. This range of possible costs can be bound by multiple bound candidates by having several constraints that bind the same intermediate variables. In addition to that, cost structures can represent disjunctions by having multiple iv on the left side of a constraint. This is the case for $iv_3 + iv_6 = \|y + x\|$ of chain (1.2). The bigger iv_3 is, the smaller iv_6 becomes. This capability is key to obtain a non-trivial lower bound for Program 13.

Given a valuation of the variables on the right-hand side of the final constraints, an evaluation of a cost structure assigns non-negative values to all the intermediate variables in such a way that the constraints of the cost structure are satisfied. The resulting value of the main cost expression corresponds to its cost.

Definition 6.11 (Cost Structure Evaluation)**.** The set of evaluations of a cost structure $\langle E, IC, FC(\boldsymbol{x})\rangle$ with respect to some parameters $\boldsymbol{a}$ is defined as follows:

$$[\![\langle E, IC, FC(\boldsymbol{a})\rangle]\!] := \left\{ \; E\sigma \;\middle|\; \begin{array}{l} \mathrm{Dm}(\sigma) = vars(\langle E, IC, FC(\boldsymbol{x})\rangle),\ \boldsymbol{x}\sigma = \boldsymbol{a},\ \models (IC \wedge FC(\boldsymbol{x}))\sigma, \\ \sigma(iv) \geq 0 \text{ for every } iv \in \mathrm{Dm}(\sigma) \end{array} \right\}$$

The idea is to infer cost structures that can be evaluated to the cost of any of the evaluations of a chain, phase or CE.

Definition 6.12 (Valid Cost Structure of a Chain)**.** Let C be a cost relation with chain ch. The cost structure $\langle E, IC, FC(\boldsymbol{xy})\rangle$ is *valid* for ch if for every evaluation $T \in [\![C[ch](\boldsymbol{a} : \boldsymbol{b})]\!]_{fc}$, there is $q \in [\![\langle E, IC, FC(\boldsymbol{ab})\rangle]\!]$ such that $Cost(T) = q$.

Similarly, a cost structure is valid for a CE c if it can be evaluated to the $Cost_c(T)$ for any $T \in [\![C]\!]_{fc}$ with $label(T) = c$ and a cost structure is valid for a phase ph if it can be evaluated to the $Cost_{ph}(T)$ of any $T \in [\![C[ch]]\!]_{fc}$ such that ch starts with ph.

In order to maintain a precise representation of the cost and obtain amortized bounds, cost structures of chains are defined in terms of both the input and output variables ($\boldsymbol{x}$ and $\boldsymbol{y}$). This is the case of the cost of chain $(3.1 \vee 3.2)^+(2)$ in Figure 6.3. Cost structures of recursive CEs can be defined in terms of the input variables $\boldsymbol{x}$ but also in terms of the variables of the recursive calls $\boldsymbol{x'}$. This is the case of the cost of CE 3.2 in Figure 6.3. And finally, cost structures of iterative non-multiple phases i.e. $(c_1 \vee \cdots \vee c_n)^+$ can depend on the input variables of the initial and last recursive calls in the phase. These variables are denoted $\boldsymbol{x_s}$ and $\boldsymbol{x_f}$ respectively. This is the case of the phase $(3.1 \vee 3.2)^+$ in Figure 6.3.

Example 6.13. We can evaluate some of the cost structures of Figure 6.3 to match the specific evaluations of Figure 6.2. For instance, the highlighted evaluation of CE 3.2 contains the root 3.2: $wh3(3, 3 : 0, 2)$ and the recursive call 3.2: $wh3(2, 2 : 0, 2)$. If we evaluate the cost structure $\langle 2iv_5, \emptyset, \{iv_5 = \|y - y' + 1\|\}\rangle$ with the corresponding values ($y = 3$ and $y' = 2$), we obtain the precise cost $2\|3 - 2 + 1\| = 4$.

Consider also the highlighted evaluation of phase $(3.1 \vee 3.2)^+$ where the root is 3.2: $wh3(3, 3 : 0, 2)$ and its last recursive call is 2.2: $wh3(0, 2 : 0, 2)$. If we evaluate the cost structure $\langle 2iv_6, \emptyset, \{iv_6 = \|y_s + x_s - y_f - x_f\|\}\rangle$ with the corresponding values ($y_s = 3$, $x_s = 3$, $x_f = 0$, and $y_f = 2$), we obtain the precise cost $2\|3 + 3 - 2 - 0\| = 8$.

The overall analysis is as follows. In a sequence of CRs $\langle C_1, C_2, \cdots, C_n\rangle$, it starts with C_n and proceeds backwards until C_1. For each C_i it computes the cost structures of the CEs first (Section 6.4), then of the phases (Section 6.5) and finally of the chains (Section 6.6). This way, at each step, the cost structures of all the components of a CE, phase or chain have already been computed and it suffices to compose them.

Example 6.14. The sequence of CRs in Program 13 is $\langle p, wh3, wh6, wh10, wh13\rangle$. The bound computation procedure starts computing cost structures for $wh13$ and finishes by computing cost structures for p. For each CR, it computes cost structures for the CEs, the phases and the chains. Consider CR $wh10$ for instance. It computes the cost of CEs 7 and 6 first. These are $\langle iv_1, \emptyset, \{iv_1 = \|z\|\}\rangle$ which originates from its reference to $wh13[9^+8]$ and $\langle 0, \emptyset, \emptyset\rangle$ (see Figure 6.1). Then, it computes the cost of phase $(7)^+$. In phase $(7)^+$ CE 7 is evaluated a number of times and each time it has a cost $\langle iv_1, \emptyset, \{iv_1 = \|z\|\}\rangle$. The cost of $(7)^+$ is the sum of all these costs. In particular iv_2 corresponds to the sum of all the instances of iv_1 of all the evaluations of CE 7. The variables iv_3 and iv_4 have an auxiliary role. They maintain the two parts of the cost expression separated $\|y_s - y_f\|$ and $\|z_s\|$ and, together with the non-final constraint, represent a non-linear bound. Finally, the cost of $(7)^+(6)$ is the sum of the costs of $(7)^+$ and 6 but expressed only in terms of the initial variable values y and z. The process is similar for other CRs. In CR $wh3$, the costs for CEs 3.1 and 3.2 and 2 are computed first, then the costs of 3.1 and 3.2 are combined to obtain the cost of $(3.1 \vee 3.2)^+$ which in turn is combined with the cost of 2 to obtain the cost of $(3.1 \vee 3.2)^+(2)]$. Here, iv_6 represents the sum of all iv_5 of all the evaluations of CE 3.2 in phase $(3.1 \vee 3.2)^+$.

Now we define cost structure bounds and their relation to the chain bounds.

Definition 6.15 (Cost Structure Bound)**.** A function $f(\boldsymbol{x}) : \mathbb{Q}^n \to \mathbb{Q} \cup \{\omega, -\omega\}$ is an upper bound of a cost structure $\langle E, IC, FC(\boldsymbol{xy})\rangle$ if $IC \wedge FC(\boldsymbol{xy}) \Rightarrow f(\boldsymbol{x}) \geq E$. Respectively, $f(\boldsymbol{x})$ is a lower bound of $\langle E, IC, FC(\boldsymbol{xy})\rangle$ if $IC \wedge FC(\boldsymbol{xy}) \Rightarrow f(\boldsymbol{x}) \leq E$.

Theorem 6.16. *Let $\langle E, IC, FC(\boldsymbol{xy})\rangle$ be a valid cost structure of a chain ch and let f be an upper bound of $\langle E, IC, FC(\boldsymbol{xy})\rangle$, f is an upper bound of $C[ch]$. Conversely, let f be an lower bound of $\langle E, IC, FC(\boldsymbol{xy})\rangle$, f is an lower bound of $C[ch]$.*

Proof. Let $T \in [\![C[ch](\boldsymbol{a} : \boldsymbol{b})]\!]_{fc}$ with cost $Cost(T)$. Applying Definition 6.12, we know that there is a σ such that $\models (IC \wedge FC(\boldsymbol{xy}))\sigma$ and $\boldsymbol{xy}\sigma = \boldsymbol{ab}$ and $E\sigma = Cost(T)$. By Definition 6.15, if f is an upper

bound of $\langle E, IC, FC(\boldsymbol{x}\boldsymbol{y})\rangle$, then $IC \wedge FC(\boldsymbol{x}\boldsymbol{y}) \Rightarrow f(\boldsymbol{x}) \geq E$. Therefore, the assignment σ satisfies $f(\boldsymbol{x}) \geq E$. We have $f(\boldsymbol{a}) \geq E\sigma = Cost(T)$. Consequently, f is an upper bound of ch. The case for lower bounds is symmetric. □

Given a valid cost structure $\langle E, IC, FC(\boldsymbol{x})\rangle$, it is easy to obtain closed-form upper/lower bounds such as $\max(2,z)\cdot(x+y)$ and $\min(2,z)\cdot(x+y)$ for Program 13. These bounds are obtained by maximizing/minimizing the main cost expression E according to the constraints IC and $FC(\boldsymbol{x})$. This is done by incrementally substituting intermediate variables in E for their upper/lower bounds defined in the constraints until E does not contain any intermediate variable. The details on how this process is implemented can be found in Section 6.8.

Example 6.17. The lower bound of chain (1.2) is computed as follows. Starting from the main cost expression $iv_2 + 2iv_6$, each iv is minimized using the constraints: (1) $iv_2 \geq iv_3 \cdot iv_4$ (2) $iv_4 \geq \|z\|$ and (3) $iv_3 + iv_6 \geq y + x$:

$$iv_2 + 2iv_6 \geq^{(1)} iv_3 \cdot iv_4 + 2iv_6 \quad \geq^{(2)} \quad iv_3 \cdot \|z\| + 2iv_6 \quad \geq^{(3)} \quad \min(\|z\|, 2) \cdot \|y + x\|$$

Finally, let us define some additional concepts for cost structures and their constraints. A constraint $ic = \sum \boldsymbol{iv} \bowtie SE$ *defines* an intermediate variable iv if it appears in its left side $defines(ic) := vars(\boldsymbol{iv})$. A constraint *uses* an intermediate variable iv if the intermediate variable appears on the right side of the constraint $uses(ic) := vars(SE)$. Final constraints also define intermediate variables but do not use any. Conversely, the main cost expression E of a cost structure uses the intermediate variables that appear in it. We lift *uses* and *defines* to sets of constraints and to cost structures. A constraint ic depends on another ic', denoted $ic \preceq ic'$, if ic uses a variable that is defined in ic', i.e. $uses(ic) \cap defines(ic) \neq \emptyset$.

6.4 Cost Equations

This section describes the procedure to obtain a valid cost structure for a recursive cost equation of the form $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi, b_1, \cdots, b_i, C(\boldsymbol{x'} : \boldsymbol{y'}), b_{i+1}, \cdots, b_n$ where b_i for $1 \leq i \leq n$ are calls to chains $C_i[ch](\boldsymbol{x_i} : \boldsymbol{y_i})$ ($C_i \neq C$) in other CRs or linear expressions $l_i(\boldsymbol{x_i})$. The non-recursive case is analogous. According to Definition 6.6, the CE cost of an evaluation is the sum of the costs of its non-recursive elements $b_1, b_2, \cdots b_n$. Similarly, a valid cost structure for c can be obtained by composing the cost structures of each b_i.

Remark 6.18. Let $\langle E_{b_i}, IC_{b_i}, FC_{b_i}(\boldsymbol{x_i}\boldsymbol{y_i})\rangle$ be a valid cost structure of b_i[1], the following cost structure is valid for the CE cost of any evaluation $T \in [\![C]\!]_{fc}$ such that $label(T) = c$.

$$\Big\langle \sum_{i=1}^{n} E_{b_i}, \bigcup_{i=1}^{n} (IC_{b_i}), \bigcup_{i=1}^{n} \big(FC_{b_i}(\boldsymbol{x_i}\boldsymbol{y_i})\big) \Big\rangle$$

The main cost expressions E_{b_i} are summed together and the constraint sets IC_{b_i} and $FC_{b_i}(\boldsymbol{x_i}\boldsymbol{y_i})$ are joined. Note that if b_i is a call to a chain $C'[ch](\boldsymbol{x}_i : \boldsymbol{y}_i)$, the cost relation C' must appear later in the CRS sequence and thus its cost structure has already been computed. If b_i is a linear expression l, the equivalent cost structure is $\langle iv_p - iv_n, \emptyset, \{iv_p = \|l\|, iv_n = \|-l\|\}\rangle$ where iv_p and iv_n are fresh intermediate variables that represent the positive and negative part of l.[2] If b_i is a constant k, we can simply consider the cost structure $\langle k, \emptyset, \emptyset\rangle$.

[1] We assume cost structures do not share intermediate variables.

[2] By case distinction, if l is positive $iv_p = \|l\| = l$, and $iv_n = \|-l\| = 0$ so $iv_p - iv_n = l$. If l is negative, we have $iv_p = \|l\| = 0$, $iv_n = \|-l\| = -l$ and $iv_p - iv_n = 0 - (-l) = l$.

In order for this joint cost structure to be useful, its final constraints $\bigcup_{i=1}^{n}(FC_{b_i}(\boldsymbol{x_i y_i}))$ have to be transformed so they are expressed in terms of the initial variables of the CE $\boldsymbol{x}$ and the variables of the recursive call $\boldsymbol{x'}$.

Transformation of final constraints

This transformation is performed with the help of the CE's constraint set φ which relates the different variables in the CE. Recall that final constraints are of an almost linear form ($\sum \boldsymbol{iv} \bowtie \|l\|$). If l is guaranteed to be non-negative ($\varphi \Rightarrow l \geq 0$), the linear constraint $\sum \boldsymbol{iv} \bowtie l$ can be used. Let FC^+ be the set of all constraints obtained thus from $\bigcup_{i=1}^{n} FC_{b_i}(\boldsymbol{x_i y_i})$. The transformation procedure performs (Fourier-Motzkin) quantifier elimination on $\exists \boldsymbol{x_1 y_1} \cdots \boldsymbol{x_n y_n}.(FC^+ \wedge \varphi)$ and obtains a constraint set that relates directly the intermediate variables of FC^+ with $\boldsymbol{x x'}$.[3] Then new final constraints in terms of $\boldsymbol{x x'}$ can be syntactically extracted from the resulting constraint set.

The number of extracted final constraints can grow large. In practice, the implementation limits the number of constraints that it keeps for each intermediate variable. In order to maximize precision, the implementation sorts the generated constraints heuristically first and keeps only the "best" ones. For instance, the implementation prioritizes upper bound constraints i.e. $\sum \boldsymbol{iv} \leq \|l\|$ where l is a constant over constraints where l is non-constant.

Example 6.19. The cost structures of chains $(3.1 \vee 3.2)^+(2)$ and $7.1^+(6)$ from Figure 6.3 are combined to form the cost structure of CE 1.2. Such cost structures are instantiated according to the variables in CE 1.2: $(x, y : x_o, y_o)$ and (y_o, z). The resulting expression is:

$$\left\langle iv_2 + 2iv_6, \qquad \{iv_2 = iv_3 \cdot iv_4\}, \qquad \{iv_6 = \|y - y_o + x\|, iv_3 = \|y_o\|, iv_4 = \|z\|\} \right\rangle$$

This is the cost structure of 1.2 in Figure 6.3 except for the final constraints which need to be transformed. The constraint set of CE 1.2 from Figure 6.1 ($\varphi_{1.2}$) guarantees that $y - y_o + x$, y_o and z are non-negative. Therefore, the transformation procedure generates the constraint set $FC^+ = \{iv_6 = y - y_o + x, iv_3 = y_o, iv_4 = z\}$ and performs quantifier elimination over $\exists x_o, y_o.(FC^+ \wedge \varphi_{1.2})$. This results in $\{iv_6 + iv_3 = y + x, iv_4 = z, x > 0, y > 0, z > 0\}$ from which the constraints $iv_3 + iv_6 = \|y + x\|$ and $iv_4 = \|z\|$ are extracted. This procedure transforms multiple constraints together and is able to find dependencies among constraints ($iv_6 = y - y_o + x$ and $iv_3 = y_0$) and merge them precisely (into $iv_3 + iv_6 = \|y + x\|$).

The rest of the final constraints, that is, the ones that cannot be guaranteed to be positive, are transformed one by one. Let $\sum \boldsymbol{iv} \bowtie \|l\|$ be a constraint, if we find another linear expression in terms of the variables of interest $l'(\boldsymbol{x x'})$ such that $\varphi \Rightarrow l \bowtie l'(\boldsymbol{x x'})$, then $\sum \boldsymbol{iv} \bowtie \|l'(\boldsymbol{x x'})\|$ holds as well.[4] The procedure finds $l'(\boldsymbol{x x'})$ by creating a linear template of $l'(\boldsymbol{x x'})$ and finding coefficients that satisfy $\varphi \Rightarrow l \bowtie l'(\boldsymbol{x x'})$ using Farkas' Lemma.

Soundness

Remark 6.18 is a direct consequence of Definition 6.6. The transformed final constraints are a logical consequence of the original final constraints, and the constraint set of the cost equation. Therefore, any assignment σ that satisfies both $\bigcup_{i=1}^{n}(FC_{b_i}(\boldsymbol{x_i y_i}))$ and φ, also satisfies the transformed constraints. In general, we say a constraint is *valid* if we can add that constraint to a valid cost structure and the cost structure remains valid.

[3] This is equivalent to projecting FC^+ onto the intermediate variables in FC^+ and $\boldsymbol{x x'}$.

[4] This can be easily seen by distinguishing cases ($l \geq 0$ and $l \leq 0$).

6.5 Phases

This section describes how to obtain cost structures for phases. For non-iterative phases $ph = (c)$ the cost structure of CE c corresponds directly to the cost structure of the phase. Hence, the section focuses on iterative and divergent phases. Let $ph = (c_1 \vee \cdots \vee c_n)^+$ be an iterative phase, the objective is to compute a valid cost structure $\langle E_{ph}, IC_{ph}, FC_{ph}(\boldsymbol{x_s x_f})\rangle$ for ph. Such a cost structure must be expressed in terms of initial values of the variables ($\boldsymbol{x_s}$) and the values of the variables in the last recursive call of the phase ($\boldsymbol{x_f}$) and must represent the sum of all the evaluations of $c_i \in ph$ (according to Definition 6.7). The case of divergent phases $ph = (c_1 \vee \cdots \vee c_n)^\omega$ is analogous with the exception that the cost structure depends only on the initial values of the variables $\boldsymbol{x_s}$.

Remark 6.20. Let $T \in [\![C[ch]]\!]_{fc}$ be an arbitrary evaluation such that ch starts with ph. Let $\langle E_{c_i}, IC_{c_i}, FC_{c_i}(\boldsymbol{x x'})\rangle$ be a valid cost structure of c_i. Consider that c_i is evaluated $\#c_i := |CEinst(T, c_i)|$ times in T (see Definition 6.7) and $\langle E_{c_ij}, IC_{c_ij}, FC_{c_ij}(\boldsymbol{a_{c_ij} a'_{c_ij}})\rangle$ represents the cost structure instance of the j-th CE evaluation of c_i for $1 \leq j \leq \#c_i$. That is, the cost structure of c_i instantiated with the parameters corresponding to the j-th CE evaluation of c_i: $\boldsymbol{a_{c_ij} a'_{c_ij}}$. The following cost structure, which represents the sum of all the cost structure instances for $1 \leq j \leq \#c_i$ and for all $c_i \in ph$, can be evaluated to the cost of the phase $Cost_{ph}(T)$.

$$\Big\langle \sum_{i=1}^{n}\sum_{j=1}^{\#c_i} E_{c_ij},\quad \bigcup_{i=1}^{n}\bigcup_{j=1}^{\#c_i}(IC_{c_ij}),\quad \bigcup_{i=1}^{n}\bigcup_{j=1}^{\#c_i}\big(FC_{c_ij}(\boldsymbol{a_{c_ij} a'_{c_ij}})\big)\Big\rangle$$

This remark is a direct consequence of Definition 6.7. Based on this remark, the procedure to generate a cost structure $\langle E_{ph}, IC_{ph}, FC_{ph}(\boldsymbol{x_s x_f})\rangle$ has three steps:

1. The expression $\sum_{i=1}^{n}\sum_{j=1}^{\#c_i} E_{c_ij}$ is transformed into a valid main cost expression E_{ph};
2. A set of non-final constraints IC_{ph} is generated using the CEs' non-final constraints IC_{c_i} (in Section 6.5.1);
3. A set of final constraints $FC_{ph}(\boldsymbol{x_s x_f})$ is generated using the CEs' final constraints $FC_{c_i}(\boldsymbol{x_{c_i} x'_{c_i}})$ and the CE definitions (in Section 6.5.2).

In order to transform $\sum_{i=1}^{n}\sum_{j=1}^{\#c_i} E_{c_ij}$ into a valid cost expression E_{ph}, the sums over the unknowns $\#c_i$ have to be removed. For this purpose, we define the following new intermediate variables:

Definition 6.21 (Sum Intermediate Variables)**.** Let iv be an intermediate variable in the cost structure $\langle E_{c_i}, IC_{c_i}, FC_{c_i}(\boldsymbol{x x'})\rangle$. The intermediate variable $smiv := \sum_{j=1}^{\#c_i} iv_j$ is the sum of all instances of iv in the different evaluations of c_i in the phase.

Now, each $\sum_{j=1}^{\#c_i} E_{c_ij}$ can be reformulated into a linear expression in terms of $smiv$. Let $E_{c_i} = q_0 + q_1 iv_1 + \cdots + q_m iv_m$, we have that $\sum_{j=1}^{\#c_i} E_{c_ij} = q_0 \#c_i + q_1 smiv_1 + \cdots + q_m smiv_m$ (where $\#c_i$ is also an intermediate variable). Note that $q_0 \#c_i + q_1 smiv_1 + \cdots + q_m smiv_m$ is a linear expression over (new) intermediate variables. The procedure does this transformation for each i in $\sum_{i=1}^{n}\sum_{j=1}^{\#c_i} E_{c_ij}$ thus obtaining a valid cost expression for the phase E_{ph}. In practice, this amounts to substituting each intermediate variables iv by their corresponding sum variables $smiv$ and multiplying the constant factor in each E_{c_i} by $\#c_i$.

Example 6.22. Let $E_9 = 1$ be the main cost expression of CE 9, the main cost expression of the phase $(9)^+$ is $E_{(9)^+} = 1\#c_9$ (where $\#c_9$ corresponds to iv_1 in Figure 6.3).

Example 6.23. Consider phase $(3.1 \vee 3.2)^+$. Let $E_{3.1} = 0$ and $E_{3.2} = 2iv_5$ be the main cost expressions of CEs 3.1 and 3.2. The main cost expression of the phase is $E_{(3.1\vee3.2)^+} = \sum_{j=1}^{\#c_{3.1}} 0 + \sum_{j=1}^{\#c_{3.2}} 2iv_{5j} = 2smiv_5$ (where $smiv_5$ corresponds to iv_6 in Figure 6.3).

Algorithm 2 Non-final constraints transformation

```
1:  function TRANSFORM_NON_FINAL(E_ph, ⟨IC_c1, IC_c2, ··· , IC_cn⟩)
2:      IC_ph := ∅
3:      for each IC_ci ∈ ⟨IC_c1, IC_c2, ..., IC_cn⟩ do
4:          for each Σiv ⋈ SE ∈ IC_ci do
5:              if (smiv ∩ uses(⟨E_ph, IC_ph⟩)) ≠ ∅ then
6:                  IC_ph ∪= transformSum(Σiv ⋈ SE)
7:              if ⋈=≤ and (⌈iv⌉ ∩ uses(⟨E_ph, IC_ph⟩)) ≠ ∅ then
8:                  IC_ph ∪= transformMax(Σiv ⋈ SE)
9:              if ⋈=≥ and (⌊iv⌋ ∩ uses(⟨E_ph, IC_ph⟩)) ≠ ∅ then
10:                 IC_ph ∪= transformMin(Σiv ⋈ SE)
11:     return IC_ph
```

Note that the Sum intermediate variables are defined for an arbitrary evaluation $T \in [\![C[ch]]\!]_{fc}$ and thus the resulting main cost expression is valid for all evaluations.

6.5.1 Transforming Non-final Constraints

This section describes how to generate a new set of non-final constraints IC_{ph} that bind the new intermediate variables (*smiv*) from the main cost expression E_{ph}. These constraints are generated from the non-final constraints of each $c_i \in ph$.

Algorithm 2 generates IC_{ph}. It receives the main cost expression of the phase E_{ph} and the non-final constraints IC_{c_i} of each $c_i \in ph$. The algorithm iterates over the non-final constraints of each IC_{c_i} for $c_i \in ph$ (Lines 3 and 4). The constraints of each c_i are visited (Line 4) in the topological order according to the dependency relation $\preceq$ (see end of Section 6.3). For each constraint that defines $\boldsymbol{iv}$, the algorithm checks whether any of the corresponding Sum variables $\boldsymbol{smiv}$ are used in the (partial) phase cost structure $\langle E_{ph}, IC_{ph}\rangle$ (Line 5). If that is the case, the algorithm calls function transformSum.

Function transformSum (Line 6) receives a constraint $\sum \boldsymbol{iv} \bowtie SE$ and generates a new constraint that binds the corresponding sum variables $\sum \boldsymbol{smiv}$. The main idea is to sum up all the instances of the constraint into a new constraint $\sum_{j=1}^{\#c_i} \sum \boldsymbol{iv}_j \bowtie \sum_{j=1}^{\#c_i} SE_j$ and reformulate it using *smiv* variables. The left-hand side can be directly reformulated: $\sum_{j=1}^{\#c_i} \sum \boldsymbol{iv}_j = \sum \boldsymbol{smiv}$. However, the right-hand side might contain a sum over non-linear expressions that cannot be reformulated only in terms of sum variables. Therefore, a new kind of intermediate variable is defined:

Definition 6.24 (Max/Min Intermediate Variables)**.** The variables $\lceil iv \rceil := \max_{1 \le j \le \#c_i}(iv_j)$ and $\lfloor iv \rfloor := \min_{1 \le j \le \#c_i}(iv_j)$ are the maximum and minimum value that an instance iv_j of iv can take in an evaluation of c_i in ph.

With the help of this new kind of variables the right-hand side of the expression: $\sum_{j=1}^{\#c_i} SE_j$ can be reformulated:

- Let $SE = q_0 + q_1 iv_1 + \cdots + q_m iv_m$, the function transformSum generates $\sum_{j=1}^{\#c_i} SE_j = q_0 \#c_i + q_1 smiv_1 + \cdots + q_m smiv_m$. This is similar to the transformation of the main cost expression.
- Let $SE = iv_k \cdot iv_p$, the expression $\sum_{j=1}^{\#c_i} SE_j$ can be approximated with the help of $\lfloor iv \rfloor_p$ or $\lceil iv \rceil_p$ depending on whether $\bowtie$ is $\le$ or $\ge$, we have that $\sum_{j=1}^{\#c_i} SE_j \le smiv_k \cdot \lceil iv \rceil_p$ and $\sum_{j=1}^{\#c_i} SE_j \ge smiv_k \cdot \lfloor iv \rfloor_p$. [5]

[5] It could also be approximated to $\lfloor iv \rfloor_k \cdot smiv_p$ and $\lceil iv \rceil_k \cdot smiv_p$ but in general the chosen approximation works better. The variable iv_k usually represents an outer loop and iv_p and inner loop (see Basic Product Strategy in Section 6.5.2).

- Let $SE = \max(\boldsymbol{iv})$ or $\min(\boldsymbol{iv})$, this case is reduced to the previous one. The expression SE is reformulated as $1 \cdot SE$ and each factor is substituted by a fresh intermediate variable: $iv_k \cdot iv_p$. Then, the constraints $iv_k \bowtie 1$ and $iv_p \bowtie SE$ are added to IC_{c_i} so they are later transformed. This way, $smiv_p$ is not generated ($\lceil iv \rceil_p$ or $\lfloor iv \rfloor_p$ will be generated instead) and $\sum_{j=1}^{\#c_i} SE_j$ will not have to be computed. Remeber that transformSum is only executed if some of the sum variables corresponding to the left side of the constraint are used in cost structure computed so far (Line 5).

The constraints generated by function transformSum might contain new Sum variables and new Max/Min variables.

Functions transformMax (Lines 8) and transformMin (Lines 10) generate constraints that define Max $\lfloor iv \rfloor$ and Min $\lceil iv \rceil$ variables. These functions are also called only if some of the Max or Min variables corresponding to the left side of the constraint are used in cost structure computed so far (Lines 7 and 9). Function transformMax is implemented as follows (transformMin is symmetric). Given a constraint $iv \leq SE \in IC_{c_i}$, it distinguishes cases for SE:

- Let $SE = q_0 + q_1 iv_1 + \cdots + q_m iv_m$ and let $V_k := \lceil iv \rceil_k$ if $q_k \geq 0$ or $V_k := \lfloor iv \rfloor_k$ if $q_k < 0$, it generates $\lceil iv \rceil \leq q_0 + q_1 V_1 + \cdots + q_m V_m$.
- Let $SE = iv_k \cdot iv_p$, it generates $\lceil iv \rceil \leq \lceil iv \rceil_k \cdot \lceil iv \rceil_p$.
- Let $SE := \max(iv_1 \cdots iv_n)$, it generates $\lceil iv \rceil \leq \max(\lceil iv \rceil_1, \cdots, \lceil iv \rceil_n)$.
- Let $SE := \min(iv_1, \cdots, iv_n)$, it generates $\lceil iv \rceil \leq \lceil iv \rceil_k$ (for $1 \leq k \leq n$).

This transformation is not valid for constraints with multiple variables on the left side. Constraints with the operator $\leq$ can be split ($\sum_{k=1}^{m} iv_k \leq SE$ implies $iv_k \leq SE$ for $1 \leq k \leq m$). But this is not the case for the constraints with the operator $\geq$. In such a case, no constraint is generated.

Example 6.25. The table below represents the transformation of the following non-final constraint set $IC_{c_i} := \{iv_1 \leq 2iv_2 + 3iv_3,\ iv_2 \leq iv_4 \cdot iv_5,\ iv_4 \leq \max(iv_6, iv_7),\ iv_5 \leq 2iv_6 - 3iv_7\}$ with the main cost expression $E := smiv_1$. Each row represents an iteration of the inner loop in Algorithm 2. The column *newIC* represents the generated constraints and the columns *SumVars* and *MVars* keep track of the used Sum and Max/Min variables. The intermediate variables iv_8 and iv_9 are added fresh in the third iteration. Finally, the marker $*$ indicates that the constraint considered at that point did not belong to the original set IC_{c_i}, but instead, it has been added in the process. This is the case of the fourth and fifth constraints. They are generated during the transformation of the constraint $iv_9 \leq \max(iv_6, iv_7)$.

	Constraint	*newIC*	*SumVars*	*MVars*
	$iv_1 \leq 2iv_2 + 3iv_3$	$\{smiv_1 \leq 2smiv_2 + 3smiv_3\}$	$\{smiv_1, smiv_2, smiv_3\}$	$\{\}$
	$iv_2 \leq iv_4 \cdot iv_5$	$\{smiv_2 \leq smiv_4 \cdot \lceil iv \rceil_5\}$	$\{smiv_{1-4}\}$	$\{\lceil iv \rceil_5\}$
	$iv_4 \leq \max(iv_6, iv_7)$	$\{smiv_4 \leq smiv_8 \cdot \lceil iv \rceil_9\}$	$\{smiv_{1-4}, smiv_8\}$	$\{\lceil iv \rceil_5, \lceil iv \rceil_9\}$
$*$	$iv_8 \leq 1$	$\{smiv_8 \leq \#c_i\}$	$\{smiv_{1-4}, smiv_8, \#c_i\}$	$\{\lceil iv \rceil_5, \lceil iv \rceil_9\}$
$*$	$iv_9 \leq \max(iv_6, iv_7)$	$\{\lceil iv \rceil_9 \leq \max(\lceil iv \rceil_6, \lceil iv \rceil_7)\}$	$\{smiv_{1-4}, smiv_8, \#c_i\}$	$\{\lceil iv \rceil_{5-7}, \lceil iv \rceil_9\}$
	$iv_5 \leq 2iv_6 - 3iv_7$	$\{\lceil iv \rceil_5 \leq 2\lceil iv \rceil_6 - 3\lfloor iv \rfloor_7\}$	$\{smiv_{1-4}, smiv_8, \#c_i\}$	$\{\lceil iv \rceil_{5-7,9}, \lfloor iv \rfloor_7\}$

All these newly generated constraints form the non-final constraint set IC_{ph}.

Soundness

All the constraints generated in this section derive directly from the original constraints in each IC_{c_i}. Therefore, the resulting (partial) cost structure is valid.

Algorithm 3 Final constraints transformation

```
1: function TRANSFORM_FINAL(⟨E_ph, IC_ph⟩, ⟨FC_c1, FC_c2, ··· , FC_cn⟩, ph)
2:     FC_ph := ∅
3:     for each FC_ci ∈ ⟨FC_c1, FC_c2, ··· , FC_cn⟩ do
4:         ⟨Psums^ci, Pms^ci⟩ := initPending(FC_ci, ⟨E_ph, IC_ph⟩)
5:     Psums := ⟨Psums^c1, Psums^c2, ··· , Psums^cn⟩
6:     Pms := ⟨Pms^c1, Pms^c2, ··· , Pms^cn⟩
7:     while (Psums ∪ Pms) ≠ ∅ do
8:         ⟨fc, origin⟩ := takeElem(Psums, Pms)
9:         ⟨IC_ph, FC_ph, Psums, Pms⟩ ∪= applyStrategies(fc, origin, ph, Psums)
10:    return ⟨E_ph, IC_ph, FC_ph⟩
```

6.5.2 Transforming Final Constraints

So far, a main cost expression E_{ph} and a set of non-final constraints IC_{ph} for a phase $ph = (c_1 \vee \cdots \vee c_n)^+$ have been computed. Algorithm 3 completes the phase's cost structure with a set of final constraints $FC_{ph}(x_s x_f)$ (and possibly additional non-final constraints) that bind the intermediate variables of E_{ph} and IC_{ph}. The algorithm receives the partial cost structure computed so far $\langle E_{ph}, IC_{ph}\rangle$, the final constraints of the CE's cost structures $\langle FC_{c_1}, FC_{c_2}, \cdots, FC_{c_n}\rangle$, and the cost equations of the phase ph.

Algorithm 3 Initialization

For each c_i with cost structure $\langle E_{c_i}, IC_{c_i}, FC_{c_i}(\boldsymbol{xx'})\rangle$, the algorithm maintains two sets of *pending* constraints $Psums^{c_i}$ and Pms^{c_i}. These sets contain the constraints that have to be transformed to bind their corresponding Sum variables or Max/Min variables. Function initPending (Line 4) initializes the pending sets using the final constraints of each $c_i \in ph$ and considering the set of used variables in E_{ph} and IC_{ph}. Let $Used := uses(\langle E_{ph}, IC_{ph}\rangle)$, it returns the sets:

$$Psums^{c_i} := \left\{ \sum \boldsymbol{iv} \bowtie \|l\| \in FC_{c_i} \;\middle|\; (\boldsymbol{smiv} \cap Used) \neq \emptyset \right\} \cup \left\{ iv_{it_i} \leq 1, iv_{it_i} \geq 1 \;\middle|\; \#c_i \in Used \right\}$$

$$Pms^{c_i} := \left\{ iv_k \leq \|l\| \in FC_{c_i} \;\middle|\; \lceil iv \rceil_k \in Used \right\} \cup \left\{ iv_k \geq \|l\| \in FC_{c_i} \;\middle|\; \lfloor iv \rfloor_k \in Used \right\}$$

(1) $Psums^{c_i}$ is initialized with the constraints of FC_{c_i} such that some $smiv_k$ in $\boldsymbol{smiv}$ has to be bound (because it is used in the cost structure). Additionally, the constraints $iv_{it_i} \leq 1$ and $iv_{it_i} \geq 1$ are added if $\#c_i$ has to be bound. The variable iv_{it_i} represents the number of times c_i is evaluated such that $smiv_{it_i} = \#c_i$.
(2) Pms^{c_i} is initialized with the constraints of FC_{c_i} with a single variable on the left side such that the corresponding $\lceil iv \rceil$ or $\lfloor iv \rfloor$ has to be bound (it is used in the cost structure). As mentioned before, the constraints of the form $\sum_{k=1}^{m} iv_k \leq \|l\|$ can be split into $iv_k \leq \|l\|$ for $1 \leq k \leq m$ on demand during the initialization of Pms^{c_i}.

Algorithm 3 Main loop

The main loop of the algorithm iterates over the pending sets (Lines 7-9). In each iteration, the algorithm removes one constraint *fc* from one of the pending sets (function takeElem in Line 8) and applies one or several strategies to the removed constraint (function applyStrategies in Line 9). The variable *origin* indicates the origin of the constraint, that is, whether the constraint was in a set $Psums^{c_i}$ or Pms^{c_i} and from which CE c_i. A strategy generates new constraints (final or non-final) for the phase's cost structure using the cost equations in ph. *Psums* is also passed to applyStrategies because some strategies can take additional pending constraints into account. The generated constraints are added to

the sets IC_{ph} or FC_{ph}. A strategy can also add additional constraints to the pending sets *Psums* or *Pms* to be processed later. The algorithm repeats the process until all $Psums^{c_i}$ and Pms^{c_i} are empty (Line 7).

In principle, the algorithm can finish without generating constraints for all intermediate variables. For instance, if the cost of the phase is actually infinite or some of the strategies fail to generate any constraint. It is also possible that the algorithm does not terminate if new constraints keep being added to the pending sets indefinitely. However, this does not happen often in practice. The loop condition can be strengthened to ensure termination. For instance, the loop can exit if all the intermediate variables are already directly or indirectly bound by final constraints. In addition, a limit can be established on the total number of iterations or on the number of times that additional pending constraints can be added to *Psums* and *Pms*.

The remaining of this section contains a series of strategies to deal with different kind of constraints and phase behaviors and, at the end of the section, there is an example of how these strategies work together to obtain a complex cost structure.

Inductive Sum Strategy

Let $\sum \boldsymbol{iv} \bowtie \|l(\boldsymbol{x}\boldsymbol{x}')\| \in Psums^{c_i}$ be a pending constraint, the strategy tries to find a linear expression that approximates the sum $\sum_{j=1}^{\#c_i} \|l(\boldsymbol{a}_{c_i j}\boldsymbol{a}'_{c_i j})\|$ for any evaluation in terms of the initial and final values of the phase $(\boldsymbol{a}_s\boldsymbol{a}_f)$.

Remember that the CEs in the phase have the form $c_i : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_i, \boldsymbol{b}, C(\boldsymbol{x}' : \boldsymbol{y}'), \boldsymbol{b}$ where $C(\boldsymbol{x}' : \boldsymbol{y}')$ is the only recursive call; φ_i is the CE's constraint set; and $\boldsymbol{x}\boldsymbol{y}$ and $\boldsymbol{x}'\boldsymbol{y}'$ are the input and output variables of the head and the recursive call.

Let us consider first the simple case where c_i is the only CE in the phase. The strategy uses the CE's constraint set φ_i and Farkas' Lemma to generate a candidate linear expression $cd(\boldsymbol{x})$ such that $\varphi_i \Rightarrow (\|l(\boldsymbol{x}\boldsymbol{x}')\| \bowtie cd(\boldsymbol{x}) - cd(\boldsymbol{x}') \geq 0)$. If a candidate $cd(\boldsymbol{x})$ is found, for any evaluation we have:

$$\sum \boldsymbol{smiv} \bowtie \sum_{j=1}^{\#c_i} \|l(\boldsymbol{a}_{c_i j}\boldsymbol{a}'_{c_i j})\| \bowtie \sum_{j=1}^{\#c_i} (cd(\boldsymbol{a}_{c_i j}) - cd(\boldsymbol{a}'_{c_i j})) = cd(\boldsymbol{a}_s) - cd(\boldsymbol{a}_f)$$

This is because each intermediate $-cd(\boldsymbol{a}'_{c_i j})$ and $cd(\boldsymbol{a}_{c_i j+1})$ cancel each other $(cd(\boldsymbol{a}'_{c_i j}) = cd(\boldsymbol{a}_{c_i j+1}))$. Therefore, the constraint $\sum \boldsymbol{smiv} \bowtie \|cd(\boldsymbol{x}_s) - cd(\boldsymbol{x}_f)\|$ is valid and can be added to FC_{ph}.

Example 6.26. This is the case of phase $(9)^+$ of Program 13 with the variables i_s, z_s, i_f, z_f and $Psums^9 = \{iv_{it_9} \leq 1, iv_{it_9} \geq 1\}$. The strategy generates the candidate $-i$ for both $iv_{it_9} \leq 1$ and $iv_{it_9} \geq 1$. We have $\{i < z, i' = i + 1\} \Rightarrow \|1\| \leq -i - (-i')$ and $\{i < z, i' = i + 1\} \Rightarrow \|1\| \geq -i - (-i')$. The generated final constraints are $smiv_{it_9} \leq \|i_f - i_s\|$ and $smiv_{it_9} \geq \|i_f - i_s\|$. Later $\|i_f - i_s\|$ will become $\|z - i\|$ in chain $(9)^+(8)$ and $\|z\|$ in CE 7. The variable $smiv_{it_9}$ corresponds to iv_1 in Figure 6.3.

If the phase contains other CEs c_e $(e \neq i)$, their effect on the sum has to be taken into account. For example, suppose that we have another c_e $(e \neq i)$ that increments our candidate by two $(\varphi_e \Rightarrow cd(\boldsymbol{x}') = cd(\boldsymbol{x})+2)$. Between two consecutive evaluations of c_i (the j-th and the $(j+1)$-th evaluations), there might be a number of evaluations n_e of c_e that increment the candidate by $2n_e$ $(cd(\boldsymbol{x}_{c_i j+1}) = cd(\boldsymbol{x}'_{c_i j}) + 2n_e)$. If we consider the complete phase evaluation where c_e is evaluated $\#c_e$ times, we obtain the following sum:

$$\sum \boldsymbol{smiv} \bowtie \sum_{j=1}^{\#c_i} \|l(\boldsymbol{a}_{c_i j}\boldsymbol{a}'_{c_i j})\| \bowtie \sum_{j=1}^{\#c_i} cd(\boldsymbol{a}_{c_i j}) - cd(\boldsymbol{a}'_{c_i j}) = cd(\boldsymbol{a}_s) - cd(\boldsymbol{a}_f) + 2\#c_e$$

Table 6.1.: CE Classification conditions for strategies: A CE c_e can be classified into a class with respect to a candidate $cd(\boldsymbol{x})$ if its condition is satisfied. Each class defines an expression/intermediate variable.

Class	Condition when $\bowtie$ is $\leq$	Condition when $\bowtie$ is $\geq$	Defines
Cnt	$(\sum \boldsymbol{iv'} \bowtie \|\|l'\|\|) \in Psums^{c_e} \wedge \|\|l'\|\| \bowtie cd(\boldsymbol{x}) - cd(\boldsymbol{x'}) \geq 0$		$cnt_e := \sum \boldsymbol{smiv'}$
Dc	$0 \leq dc_e(\boldsymbol{x x'}) \leq cd(\boldsymbol{x}) - cd(\boldsymbol{x'})$	$dc_e(\boldsymbol{x x'}) \geq cd(\boldsymbol{x}) - cd(\boldsymbol{x'})$	$iv_{dc_e} := \|\|dc_e(\boldsymbol{x x'})\|\|$
Ic	$ic_e(\boldsymbol{x x'}) \geq cd(\boldsymbol{x'}) - cd(\boldsymbol{x})$	$0 \leq ic_e(\boldsymbol{x x'}) \leq cd(\boldsymbol{x'}) - cd(\boldsymbol{x})$	$iv_{ic_e} := \|\|ic_e(\boldsymbol{x x'})\|\|$
CntR	$(\sum \boldsymbol{iv'} \leq \|\|l'\|\|) \in Psums^{c_e} \wedge \|\|l'\|\| \leq \|\|cd(\boldsymbol{x})\|\| - \|\|cd(\boldsymbol{x'})\|\|$		$cntr_e := \sum \boldsymbol{smiv'}$
Rst	$cd(\boldsymbol{x'}) \bowtie \|\|rst_e(\boldsymbol{x})\|\|$		$iv_{rst_e} := \|\|rst_e(\boldsymbol{x})\|\|$
CntT	$\left(\sum \boldsymbol{iv'} \bowtie \|\|l'\|\|\right) \in Psums^{c_e} \wedge l' \bowtie cd(\boldsymbol{x}) \geq 0 \wedge cd(\boldsymbol{x'}) - cd(\boldsymbol{x}) \bowtie q_e$		$cntt_e := \sum \boldsymbol{smiv'}$
Nop	$cd(\boldsymbol{x'}) - cd(\boldsymbol{x}) = 0$		

That is, the sum computed for the simple case $cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f})$ plus the sum of all the increments to the candidate $2\#c_e$ effected by CE c_e. This constraint can be expressed with the following intermediate and final constraints:

$$\sum \boldsymbol{smiv} \bowtie iv_{cd+} - iv_{cd-} + 2\#c_e \qquad iv_{cd+} \bowtie \|cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})\| \qquad iv_{cd-}! \bowtie \|-cd(\boldsymbol{x_s}) + cd(\boldsymbol{x_f})\|$$

where iv_{cd+} and iv_{cd-} represent the positive and negative part of $cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})$ and $! \bowtie$ is the opposite operator of $\bowtie$ (if $\bowtie$ is $\leq$, the operator $! \bowtie$ is $\geq$ and vice-versa).

In the general case, the strategy is divided into three steps:

1. First, the strategy generates a candidate cd using c_i's constraint set φ_i, the condition $\|l(\boldsymbol{x x'})\| \bowtie cd(\boldsymbol{x}) - cd(\boldsymbol{x'}) \geq 0$, and Farkas' Lemma (as in the simple case before).

2. Next, it classifies the CEs of the phase $c_e \in ph$ (including c_i) according to their effect on the candidate.

3. Finally, it uses this classification to generate constraints that take these effects into account.

Let us consider now the second and third steps:

Cost Equation Classification

Each $c_e \in ph$ has to be classified with respect to a candidate cd into a class. Each class has a condition and defines a linear expression (see Table 6.1). In order to classify a CE c_e into a class, its condition has to be implied by the corresponding CE's constraint set φ_e. Some of the conditions contain unknown linear expressions: $dc_e(\boldsymbol{x x'})$, $ic_e(\boldsymbol{x x'})$ or $rst_e(\boldsymbol{x x'})$ (For the classes *Dc*, *Ic* and *Rst* respectively). The implication can be verified and the unknown linear expressions can be inferred using Farkas' Lemma and linear templates. The considered classes in this strategy are[6]:

- *Cnt*: $c_e \in Cnt$ if there is a constraint $\sum \boldsymbol{iv'} \bowtie \|l'\| \in Psums^{c_e}$ that can also be bound by the candidate: $\varphi_e \Rightarrow \|l'\| \bowtie cd(\boldsymbol{x}) - cd(\boldsymbol{x'})$. The expression $\sum \boldsymbol{smiv'}$ can be incorporated to the left-hand side of the constraint. We define $cnt_e := \sum \boldsymbol{smiv'}$ as a shorthand. Note that c_i, whose constraint was used to generate the candidate in the first place, trivially satisfies the condition and thus $c_i \in Cnt$.

[6] The rest of the classes are used by other strategies.

- *Dc*: $c_e \in Dc$ if in each evaluation of c_e the candidate is decremented by at least $dc_e(\boldsymbol{x}\boldsymbol{x}')$ (or at most $dc_e(\boldsymbol{x}\boldsymbol{x}')$ if $\bowtie$ is $\geq$). A fresh intermediate variable is assigned to this amount $iv_{dc_e} := \|dc_e(\boldsymbol{x}\boldsymbol{x}')\|$. To generate a valid constraint, the sum of all those decrements, that is $smiv_{dc_e}$, is subtracted from the right-hand side of the constraint.

- *Ic*: $c_e \in Ic$ if in each evaluation of c_e the candidate is incremented by at most $ic_e(\boldsymbol{x}\boldsymbol{x}')$ (or at least $ic_e(\boldsymbol{x}\boldsymbol{x}')$ if $\bowtie$ is $\geq$). As before, a fresh intermediate variable is assigned to that amount $iv_{ic_e} := \|ic_e(\boldsymbol{x}\boldsymbol{x}')\|$. To generate a valid constraint, the sum of all those increments, that is $smiv_{ic_e}$, is added to the right-hand side of the constraint.

Constraint Generation

Once all the CEs in a phase have been classified with respect to a candidate, the strategy generates the following constraints to be added to the phase's constraints:

Theorem 6.27. *Let $!\bowtie$ be the reverse of $\bowtie$ (e.g. $\geq$ if $\bowtie$ is $\leq$). If every $c_e \in ph$ has been successfully classified into Cnt, Ic or Dc with respect to a candidate $cd(\boldsymbol{x})$, the following constraints are valid:*

$$\sum_{c_e \in Cnt} cnt_e \bowtie iv_{cd+} - iv_{cd-} + \sum_{c_e \in Ic} smiv_{ic_e} - \sum_{c_e \in Dc} smiv_{dc_e} \qquad \begin{array}{l} iv_{cd+} \bowtie \|cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})\| \\ iv_{cd-}! \bowtie \|-cd(\boldsymbol{x_s}) + cd(\boldsymbol{x_f})\| \end{array}$$

The proof of this theorem can be found in Section 6.10. Note that iv_{cd+} and $-iv_{cd-}$ represent the positive and negative part of $cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})$. The constraints bind the sum of all *smiv* in cnt_e (for each $c_e \in Cnt$) to $cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})$ plus all the increments $\sum_{c_e \in Ic} smiv_{ic_e}$ minus all the decrements $\sum_{c_e \in Dc} smiv_{dc_e}$. If the class *Ic* is empty, $cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})$ is guaranteed to be positive (the candidate is never incremented) and the summand $-iv_{cd-}$ and its corresponding constraint $iv_{cd-}! \bowtie \|-cd(\boldsymbol{x_s}) + cd(\boldsymbol{x_f})\|$ can be eliminated.

Finally, the strategy adds constraints for the new intermediate variables iv_{ic_e} and iv_{dc_e} to the corresponding pending set $Psums^{c_e}$ so their sums $smiv_{ic_e}$ and $smiv_{dc_e}$ are bound afterwards:

- For each $c_e \in Ic$, the constraint $iv_{ic_e} \bowtie \|ic_e(\boldsymbol{x}\boldsymbol{x}')\|$ is added to $Psums^{c_e}$
- For each $c_e \in Dc$, the constraint $iv_{dc_e}! \bowtie \|dc_e(\boldsymbol{x}\boldsymbol{x}')\|$ is added to $Psums^{c_e}$

Furthermore, if $\bowtie$ is $\leq$ the first constraint in Theorem 6.27 can be simplified by removing the intermediate variables that are subtracted. This maintains the validity of the constraint (remember that intermediate variables are always non-negative) which has the following form:

$$\sum_{c_e \in Cnt} cnt_e \leq iv_{cd+} + \sum_{c_e \in Ic} smiv_{ic_e}$$

This simplification also implies that the pending constraints for each $c_e \in Dc$ can be discarded.

Example 6.28. In phase $(3.1 \vee 3.2)^+$ we have $iv_5 \leq \|y - y' + 1\| \in Psums^{3.2}$. A valid candidate is $y + x$. The CEs are classified as follows: CE $3.2 \in Cnt$ because it has generated the candidate and $cnt_{3.2} := smiv_5$; and CE $3.1 \in Dc$ because $y + x$ decreases in CE 3.1 by $dc_{3.1} = 0$. The generated constraints are: $smiv_5 \leq iv_{cd+} - iv_{cd-} - smiv_{dc}$, $iv_{cd+} \leq \|(y_s + x_s) - (y_f + x_f)\|$ and $iv_{cd+} \leq \|-(y_s + x_s) + (y_f + x_f)\|$. However, given that *Ic* is empty and $dc_{3.1} = 0$, they can be simplified to a single constraint: $smiv_5 \leq \|(y_s + x_s) - (y_f + x_f)\|$ (where $smiv_5$ is iv_6 in Figure 6.3).

Example 6.29. The class *Cnt* makes possible to bind Sum variables of different c_i under a single constraint. For instance, if we had[7] $iv_{it_{3.1}} \geq 1 \in Psums^{3.1}$ and $iv_{it_{3.2}} \geq 1 \in Psums^{3.2}$, the expression x would be a valid candidate with the classification $Cnt = \{3.1, 3.2\}$ with $cnt_{3.1} := smiv_{it_{3.1}}$ and $cnt_{3.2} := smiv_{it_{3.2}}$.

[7] $smiv_{it_{3.1}}$ and $smiv_{it_{3.2}}$ are actually not needed for computing the cost of the program in this case. Therefore, these constraints are never added to the pending sets.

Program 14	Cost relations

`void cds(int x,int y){`
`int z=*;`
`while(x>0 && y>0 && z>0)`
`{`
`if(*) y--;`
`else y=*;`
`x--;`
`z--;`
`tick(1);`
`}`
	1.1: $cds(x,y) = \{x>0, y>0, z>0\}, wh[(3 \vee 4)^+(2)](x,y,z)$
	1.2: $cds(x,y) = wh[(2)](x,y,z)$
	2: $wh(x,y,z) = \{\}$
	3: $wh(x,y,z) = \{x>0, y>0, z>0, x' = x-1, y' = y-1, z' = z-1\}, 1, wh(x',y',z')$
	4: $wh(x,y,z) = \{x>0, y>0, z>0, x' = x-1, z' = z-1\}, 1, wh(x',y',z')$

Figure 6.4.: Program 14: Example where multiple candidates are important

The strategy would generate the (simplified) constraint $smiv_{it_{3.1}} + smiv_{it_{3.2}} \geq \|x_s - x_f\|$ which is equivalent to $\#c_{3.1} + \#c_{3.2} \geq \|x_s - x_f\|$ and represents that $wh3$ iterates at least $\|x_s - x_f\|$ times. Without *Cnt*, the strategy would fail to obtain a non-trivial lower bound for $\#c_{3.1}$ or $\#c_{3.2}$ as they can both be 0 (if considered individually).

Discussion

It is worth noting that while the initial motivation for this strategy was to find a linear expression that could bind the sum $\sum \boldsymbol{smiv}$, the resulting constraints do not necessarily represent a linear expression due to the effects of other CEs in the phase. For instance, the added variables capturing increments $smiv_{ic_e}$ can represent non-linear expressions.

The strategy can fail at two points. (1) It can fail to generate a candidate (if there is no linear constraint that satisfies the conditions) or (2) it can fail to classify the CEs of a phase with respect to a given candidate. In the latter case, it can attempt to generate a different candidate (Possibly taking the CEs where the classification failed into account). Moreover, even if the strategy does not fail, we can adjust it to consider several candidates. In general, the more candidates it considers, the more constraints it will be able to generate and the better precision it can achieve. Evidently, this comes at a cost in terms of performance.

Example 6.30. Let us compute a cost structure for the phase $(3 \vee 4)^+$ of Program 14 (Figure 6.4). The initial pending set of CE 3 is $Psums^3 = \{iv_{it_3} \leq 1, iv_{it_3} \geq 1\}$. The expression y is a valid candidate for $iv_{it_3} \leq 1$ in CE 3. However, CE 4 cannot be classified successfully with respect to the candidate y and consequently the strategy fails at this point. The strategy can backtrack and consider a different candidate.

Both x and z are valid candidates and can successfully classify CE 4 in *Cnt* with $cnt_4 = iv_{it_4}$. The generated constraints are: $\#c_3 + \#c_4 \leq \|x_s - x_f\|$ and $\#c_3 + \#c_4 \leq \|z_s - z_f\|$ which can be transformed to $\#c_3 + \#c_4 \leq \|x\|$ and $\#c_3 + \#c_4 \leq \|z\|$ for the chain $(3 \vee 4)^+(2)$. Unfortunately, even though the candidate z was successfully classified, later in the analysis its corresponding constraint $\#c_3 + \#c_4 \leq \|z\|$ will be lost. This is because z is initialized to an unknown value and thus it cannot be expressed in terms of the input parameters of function x and y. Therefore, if the analysis considers only the candidate z, it will fail to obtain a bound for CE 1.1.

Finally, note that this strategy is completely symmetric. It can be used to obtain upper and lower bounds. However, it is not useful for divergent phases in which the variables $\boldsymbol{x_f}$ cannot be related to any specific value or output variable.

Inductive Sum Strategy with Resets

This variant of the *Inductive Sum* strategy allows us to obtain upper bounds of sums in terms of the initial variables of the phase only ($\boldsymbol{x_s}$). Such bounds are also useful for divergent phases. The strategy follows the same scheme as before.

Candidate Generation

Let $\sum \boldsymbol{iv} \leq \|l\| \in Psums^{c_i}$, it generates a candidate $cd(\boldsymbol{x})$ using the constraint set φ_i of CE c_i and Farkas' Lemma. However, this time the strategy uses the condition $\|l'\| \leq \|cd(\boldsymbol{x})\| - \|cd(\boldsymbol{x'})\|$. This condition considers only the positive part of the candidate which allows us to ignore the final value of the candidate $cd(\boldsymbol{x}_f)$ and guarantee that the generated constraint is valid for (selected partial evaluations of) divergent phases (see Definition 6.3).

Cost Equation Classification

As in the previous strategy, the CEs in the phase are classified. This strategy considers the class *CntR*, instead of *Cnt*, whose condition considers the positive part of the candidates as well (see Table 6.1). It considers the classes *Dc* and *Ic* to capture decrements and increments of the candidate and, in addition, it considers the *Rst* class to support phases where the candidate is reset to a completely different value. If $c_e \in Rst$ the candidate is reset to a value of at most $\|rst_e(\boldsymbol{x})\|$. A fresh intermediate variable is assigned to such reset value $iv_{rst_e} := \|rst_e(\boldsymbol{x})\|$.

Constraint Generation

Theorem 6.31. *If every $c_e \in ph$ is classified into CntR, Ic, Dc and Rst with respect to a candidate $cd(\boldsymbol{x})$, the following constraints are valid:*

$$\sum_{c_e \in CntR} cntr_e \leq iv_{cd} + \sum_{c_e \in Ic} smiv_{ic_e} + \sum_{c_e \in Rst} smiv_{rst_e} \qquad iv_{cd} \leq \|cd(\boldsymbol{x_s})\|$$

The strategy generates the following pending constraints:

- For each $c_e \in Ic$, the constraint $iv_{ic_e} \leq \|ic(\boldsymbol{x x'})\|$ is added to $Psums^{c_e}$.
- For each $c_e \in Rst$, the constraint $iv_{rst_e} \leq \|rst(\boldsymbol{x})\|$ is added to $Psums^{c_e}$.

Note that this strategy ignores the decrements of $c_e \in Dc$. For divergent phases the lower bounds on variables $smiv_{dc_e}$ always be 0, because the evaluation can be truncated at any point.

Basic Product Strategy

In many cases, the previous strategies fail to even infer a candidate. Given a constraint $\sum \boldsymbol{iv} \bowtie \|l(\boldsymbol{x x'})\| \in Psums^{c_i}$, it might be impossible to infer a linear expression representing $\sum_{j=1}^{\#c_i} \|l(\boldsymbol{x}_j \boldsymbol{x}'_j)\|$. This is the case for most nested loops such as *wh*10 in Program 13.

Example 6.32. Consider the cost computation of phase $(7)^+$ of Program 13 in which the pending set $Psums^7$ contains the constraint $iv_1 \leq \|z\|$. In the phase, the variable z does not change in CE 7 and $\#c_7$ is at most y so $\sum_{j=1}^{\#c_7} \|z\| = \|y\| \cdot \|z\|$ which is non-linear. This result can be obtained by rewriting the constraint $iv_1 \leq \|z\|$ as $iv_1 \leq 1 \cdot \|z\|$ and generating the constraint $smiv_1 \leq smiv_{it_7} \cdot \lceil iv \rceil_{mz}$ (that corresponds to $iv_2 \leq iv_3 \cdot iv_4$ using the intermediate variables of Figure 6.3). Then, $iv_{it_7} \leq 1$ is added to $Psums^7$ and $iv_{mz} \leq \|z\|$ is added to Pms^7. These constraints will be later processed by the strategies *Inductive Sum* and *Max-Min* respectively.

In general, given a constraint $\sum \boldsymbol{iv} \leq \|l\| \in Psums^{c_i}$ where l is not a constant, the *Basic Product* strategy generates the non-final constraint:

$$\sum \boldsymbol{smiv} \leq smiv_{it_i} \cdot \lceil iv \rceil_p$$

And it adds the pending constraints $iv_{it_i} \leq 1$ to $Psums^{c_i}$ and $iv_p \leq \|l\|$ to Pms^{c_i}. This way, it reduces a complex sum into a simpler sum and a max/minimization. This strategy is the main source of non-final constraints in the form of a product of two intermediate variables. It proceeds analogously for constraints with the operator $\geq$.

Max-Min Strategy

This strategy deals with constraints $iv \bowtie \|l\| \in Pms^{c_i}$ and its role is to generate constraints for Max $\lceil iv \rceil$ and Min $\lfloor iv \rfloor$ variables. It follows the same three steps scheme or the *Inductive Sum* strategies.

First, it generates a candidate $cd(\boldsymbol{x})$ using the CE's constraint set φ_i. However, the condition used to generate the candidate is simply $l \bowtie cd(\boldsymbol{x})$ since it has to bind a single instance of l instead of the sum of all its instances. Second, the strategy classifies the $c_e \in ph$ with respect to the candidate. In this case, it does not consider the class *Cnt* but it considers *Ic*, *Dc* and *Rst* (see Table 6.1). Third, the strategy generates constraints:

Theorem 6.33. *Let $iv \leq \|l\| \in Pms^{c_i}$ and let $cd(\boldsymbol{x})$ be a candidate such that $\varphi_i \Rightarrow l \leq cd(\boldsymbol{x})$. If every $c_e \in ph$ is classified into Dc, Ic and Rst with respect to $cd(\boldsymbol{x})$, the following constraints are valid:*

$$\lceil iv \rceil \leq iv_{max} + \sum_{c_e \in Ic} smiv_{ic_e} \qquad iv_{max} \leq \max_{c_e \in Rst}(\lceil iv \rceil_{rst_e}, iv_{cd}) \qquad iv_{cd} \leq \|cd(\boldsymbol{x_s})\|$$

These constraints bind $\lceil iv \rceil$ to the sum of all the increments $smiv_{ic_e}$ for $c_e \in Ic$ plus the maximum of all the maximum values that the resets can take $\lceil iv \rceil_{rst_e}$. This maximum also includes the initial value of the candidate $cd(\boldsymbol{x_s})$ in case it is never reset.

The strategy adds the following pending constraints:

- For each $c_e \in Ic$, the constraint $iv_{ic_e} \leq \|ic_e(\boldsymbol{x x'})\|$ is added to $Psums^{c_e}$.
- For each $c_e \in Rst$, the constraint $iv_{rst_e} \leq \|rst_e(\boldsymbol{x})\|$ is added to Pms^{c_e}.

The strategy proceeds analogously for constraints with the operator $\geq$ but it subtracts the decrements instead of adding the increments and takes the minimum of the resets $\lfloor iv \rfloor_{rst_e}$:

Theorem 6.34. *Let $iv \geq \|l\| \in Pms^{c_i}$ and let $cd(\boldsymbol{x})$ be a candidate such that $\varphi_i \Rightarrow l \geq cd(\boldsymbol{x})$. If every $c_e \in ph$ is classified into Dc, Ic and Rst with respect to $cd(\boldsymbol{x})$, the following constraints are valid:*

$$\lfloor iv \rfloor \geq iv_{min} - \sum_{c_e \in Dc} smiv_{dc_e} \qquad iv_{min} \geq \min_{c_e \in Rst}(\lfloor iv \rfloor_{rst_e}, iv_{cd}) \qquad iv_{cd} \geq \|cd(\boldsymbol{x_s})\|$$

Example 6.35. In Example 6.32 the constraint $iv_{mz} \leq \|z\|$ was added to Pms^7 during the computation of the cost of $(7)^+$. The *Max-Min* strategy generates a candidate z and classifies CE 7 in *Dc* with $dc_7 := 0$ (z is not modified in CE 7). The resulting (simplified) constraint is $\lceil iv \rceil_{mz} \leq \|z_s\|$ (which corresponds to $iv_4 \leq \|z_s\|$ using the intermediate variables of Figure 6.3).

Triangular Sum Strategy

This strategy represents an alternative to the *Basic Product* strategy for dealing with constraints $\sum \boldsymbol{iv} \bowtie \|l(\boldsymbol{x x'})\| \in Psums^{c_i}$ where $\|l(\boldsymbol{x x'})\|$ varies in each iteration by a constant amount.

Refined cost relations of Program 2

2 : $for_2(x,n) = \{y = x, x < n, x' = x+1\}, for_3[(3)^+(4)](y,n), for_2(x',n)$
3 : $for_2(x,n) = \{x \geq n\}$
4 : $for_3(y,n) = \{y < n, y' = y+1\}, 1, for_3(y',n)$
5 : $for_3(y,n) = \{y \geq n\}$

Lower bound: $\|n\|^2/2 + \|n\|/2$

Figure 6.5.: Refined cost relations of Program 2 and its lower bound

Example 6.36. A typical example is Program 2 (in Page 5). Figure 6.5 contains its refined cost relations. In this example, cost equations 2 and 3 represent the outer loop and 4 and 5 the inner loop. The chain that represents the total cost of the program is $(2)^+(3)$. The final values of the variables x_o, y_o and n_o are not included in the CRs to simplify the presentation. Let us consider obtaining the lower bound of such an example.

Assume the cost of the inner loop (chain $(4)^+(5)$) is $\langle iv_1, \emptyset, \{iv_1 = \|n-y\|\}\rangle$ which yields a cost of $\langle iv_1, \emptyset, \{iv_1 = \|n-x\|\}\rangle$ for CE 2. The main cost expression of the phase $(2)^+$ is $E_{(2)^+} := smiv_1$, there are no non-final constraints and the pending sets are: $Psums^2 = \{iv_1 \leq \|n-x\|, iv_1 \geq \|n-x\|\}$ and $Pms^2 = \emptyset$. If we apply the Basic Product strategy to $iv_1 \geq \|n-x\|$, we would obtain $smiv_1 \geq smiv_{it_2} \cdot \lfloor iv \rfloor_2$ and later $smiv_{it_2} \geq \|n_s - x_s\|$ and $\lfloor iv \rfloor_2 \geq 1$ (the minimum value of $\|n-x\|$ is 1 in the last iteration) which represents the imprecise lower bound $\|n-x\| \cdot 1$.

Instead, we consider that $\|n-x\|$ decreases by at most 1 in each iteration so we can reformulate:

$$\sum_{j=1}^{\#c_2} \|n_j - x_j\| \geq \sum_{j=1}^{\#c_2} (\|n_s - x_s\| - (j-1)) \quad = \|n_s - x_s\| \cdot \#c_2 - \sum_{j=0}^{\#c_2 - 1} j = \|n_s - x_s\| \cdot \#c_2 - \tfrac{1}{2}(\#c_2^2 - \#c_2)$$

This expression can be represented with constraints as follows:

$$smiv_1 \geq iv_{p1} - \tfrac{1}{2} iv_{p2} + \tfrac{1}{2} smiv_{it_2} \qquad iv_{p1} \geq iv_{ini} \cdot smiv_{it_2}$$
$$iv_{p2} \leq smiv_{it_2} \cdot smiv_{it_2} \qquad iv_{ini} \geq \|n_s - x_s\|$$

Note that the constraint over iv_{p2} has $\leq$ instead of $\geq$. This is because iv_{p2} appears negated in the first constraint and it has to be maximized. Later, applying the *Inductive Sum* strategy to $iv_{it_2} \leq 1$ and $iv_{it_2} \geq 1$ (in $Psums^2$), we generate $smiv_{it_2} = \|(n_s - x_s) - (n_f - x_f)\|$. When we compute the cost of the complete chain $(2)^+(3)$, we transform $\|(n_s - x_s) - (n_f - x_f)\|$ into $\|n_s - x_s\|$ (because $n_f - x_f$ must be 0 in chain $(2)^+(3)$). If we minimize the cost of the resulting cost structure, we obtain:

$$\|n_s - x_s\|^2 - \tfrac{1}{2}\|n_s - x_s\|^2 + \tfrac{1}{2}\|n_s - x_s\| \quad = \tfrac{1}{2}\|n_s - x_s\|^2 + \tfrac{1}{2}\|n_s - x_s\| = \tfrac{1}{2}\|n_s\|^2 + \tfrac{1}{2}\|n_s\|$$

In the general case, given a constraint $\sum \boldsymbol{iv} \bowtie \|l\| \in Psums^{c_i}$, the strategy generates a candidate $cd(\boldsymbol{x})$ under the condition that it approximates the cost of one instance of $\|l(\boldsymbol{x}\boldsymbol{x'})\|$, it is positive, and it varies by a constant amount $q_i \in \mathbb{Q}$. The condition is:

$$l \bowtie cd(\boldsymbol{x}) \geq 0 \quad \wedge \quad cd(\boldsymbol{x'}) - cd(\boldsymbol{x}) \bowtie q_i$$

This strategy considers the classes *CntT* and *Nop* (see Table 6.1 in Page 89). If $c_e \in CntT$, there exists a constraint $(\sum \boldsymbol{iv'} \bowtie \|l'\|) \in Psums^{c_e}$ that is bound by the candidate and the candidate varies by an amount q_e. As in previous strategies, this condition coincides with the one used to generate the candidate and $c_i \in CntT$. The CEs $c_e \in Nop$ do not modify the candidate.

Program 15	Refined cost relations
1 while(x>0){	
2 if(*){	1: $wh1(x,y,z) = \{x \leq 0\}$
3 while(y>0 && *){	2: $wh1(x,y,z) = \{x > 0, y > y' \geq 0, x' = x-1\}, wh3[(7)^+(6)](y:y'),$ $wh1(x,y',z)$
4 y--;	
5 tick(2);	3: $wh1(x,y,z) = \{x > 0, y = y' \geq 0, x' = x-1\}, wh3[(6)](y:y'),$ $wh1(x,y',z)$
6 }	
7 }else{	4: $wh1(x,y,z) = \{x' = x-1 \geq 0, y' = y+1\}, wh1(x',y',z)$
8 if(*) y++;	5: $wh1(x,y,z) = \{x > 0, x' = x-1, y = z\}, wh1(x',y',z)$
9 else y=z;	
10 }	6: $wh3(y:y_o) = \{y = y_o\}$
11 x--;	7: $wh3(y:y_o) = \{y \geq 1, y' = y-1\}, 2, wh3(y':y_o)$
12 }	

Figure 6.6.: Program 15: Example with complex phase

Theorem 6.37. *If every $c_e \in ph$ is classified into CntT and Nop with respect to a candidate $cd(\boldsymbol{x})$. Let q be $q := \max_{c_e \in CntT}(q_e)$ when $\bowtie$ is $\leq$ or $q := \min_{c_e \in CntT}(q_e)$ when $\bowtie$ is $\geq$. The following constraints are valid:*

$$\sum_{c_e \in CntT} cntt_e \bowtie iv_{p1} + \tfrac{q}{2} iv_{p2} - \tfrac{q}{2} iv_{its} \ , \qquad iv_{p1} \bowtie iv_{ini} \cdot iv_{its}$$
$$iv_{p2} = iv_{its} \cdot iv_{its}, \qquad iv_{its} = \sum_{c_e \in CntT} smiv_{it_e}, \qquad iv_{ini} \bowtie \|cd(\boldsymbol{x_s})\|$$

The intermediate variables iv_{p1}, iv_{p2}, iv_{its}, and iv_{ini} are fresh: iv_{p1} and iv_{p2} simply represent different parts of the expression, the variable iv_{its} represents the sum of all the iterations $\#c_e$ of $c_e \in CntT$, and iv_{ini} represents the initial value of the candidate. The constraints of the form $iv = x$ stand for $iv \leq x$ and $iv \geq x$. Finally, for each $c_e \in CntT$, the constraints $iv_{it_e} \leq 1$ and $iv_{it_e} \geq 1$ are added to each $Psums^{c_e}$.

This strategy allows us to obtain both upper and lower bounds that are more precise than the ones obtained with the Basic Product strategy. However, it also generates many more constraints so its adequacy highly depends on the goals of the analysis. For instance, if we are only interested in the asymptotic complexity, this strategy can be advantageous for lower bounds but not for upper bounds. Consider Program 2, the expression $\frac{\|n\|^2}{2} + \frac{\|n\|}{2}$ is the upper and lower bound obtained with the Triangular Sum strategy and it is quadratic. Without the Triangular Sum strategy, the obtained upper and lower bounds are n^2 and n respectively. While both are imprecise at the concrete level, the upper bound is asymptotically precise. In the future work chapter (Chapter 10) there is a discussion on how this strategy could be improved and extended.

6.5.3 Example of Phase Cost Structure Inference

This section contains a complete example of the inference of a cost structure for a phase to illustrate how the different strategies work together. Figure 6.6 contains Program 15 and its refined cost relations. The inner loop of the program coincides with the loop *wh6* of Program 13. The outer loop has 5 cost equations. CEs 2 and 3 represent the loop paths that reach the inner loop. In CE 2 the body of the inner loop is executed at least once and in CE 3 the body of inner loop is not executed. CE 4 corresponds to the loop path that visits Line 8 in which y is incremented. CE 5 corresponds the loop path that visits Line 9. There, variable y is reset to z. Finally, CE 1 is the exit path of the loop.

The cost structure of the phase $(2 \vee 3 \vee 4 \vee 5)^+$ is computed based on the cost structures of CEs $2-5$. The example assumes that the cost structure for CE 2 is $\langle 2iv_1, \emptyset, iv_1 \leq \|y - y'\| \rangle$ and the cost structures

Table 6.2.: Cost structure computation of phase $(2 \vee 3 \vee 4 \vee 5)^+$ in Program 15.

Pending	$Psums^2 = \{iv_1 \leq \|y - y'\|\}$
Selected	$iv_1 \leq \|y - y'\|$
Strategy	*ISR* $cd := y$ Classification: $CntR = \{2\}$, $Dc = \{3\}$, $Ic = \{4\}$, $Rst = \{5\}$ $cntr_2 = smiv_5$, $dc_3 = 0$, $ic_4 = 1$, $rst_5 = \|z\|$
NewCs	$smiv_1 \leq iv_2 + smiv_{ic_4} + smiv_{rst_5}$, $iv_2 \leq \|y\|$
Pending	$Psums^4 = \{iv_{ic_4} \leq 1\}$, $Psums^5 = \{iv_{rst_5} \leq \|z\|\}$
Selected	$iv_{rst_5} \leq \|z\|$
Strategy	*BP*
NewCs	$smiv_{rst_5} \leq smiv_{it_5} \cdot \lceil iv \rceil_3$
Pending	$Psums^4 = \{iv_{ic_4} \leq 1\}$, $Psums^5 = \{iv_{it_5} \leq 1\}$, $Pms^5 = \{iv_3 \leq \|z\|\}$
Selected	$iv_3 \leq \|z\|$
Strategy	*MM* $cd := z$ Classification: $Dc = \{2, 3, 4, 5\}$, $dc_{2,3,4,5} = 0$
NewCs	$\lceil iv \rceil_3 \leq \|z\|$
Pending	$Psums^4 = \{iv_{ic_4} \leq 1\}$, $Psums^5 = \{iv_{it_5} \leq 1\}$
Selected	$iv_{ic_4} \leq 1$
Strategy	*ISR* $cd := x$ Classification: $CntR = \{4, 5\}$, $Dc = \{2, 3\}$ $cntr_4 = smiv_{ic_4}$, $cntr_5 = smiv_{it_5}$, $dc_{2,3} = 1$
NewCs	$smiv_{ic_4} + smiv_{it_5} \leq \|x\|$
Done	

of CEs $3 - 5$ are empty. For simplicity, it only considers constraints for upper bounds (with $\leq$). The main cost expression of the phase is $2smiv_1$. Table 6.2 contains all the iterations of the main loop in Algorithm 3 where each iteration has four parts:

Pending The pending sets $Psums^{c_i}$ and Pms^{c_i}.

Selected The constraint selected by function takeElem from one of the pending sets.

Strategy The strategy applied to the selected constraint: *ISR* (Inductive Sum with Resets), *BP* (Basic Product), or *MM* (Max-Min). In this example the Inductive Sum and Triangular Sum strategies are not used. This part also contains the classification of the CEs $c_e \in ph$ and the related defined expressions $cntr_e$, ic_e, etc.

NewCs The constraints generated. The constraints added to the pending sets are not included here but they can be seen in the pending sets of the next iteration.

The algorithm iterates four times until all the intermediate variables are bound. The resulting cost structure $\langle E, IC, FC(\boldsymbol{x}) \rangle$ contains all the generated constraints (NewCs):

$$\left\langle 2smiv_1, \left\{ \begin{array}{l} smiv_1 \leq iv_2 + smiv_{ic_4} + smiv_{rst_5} \\ smiv_{rst_5} \leq smiv_{it_5} \cdot \lceil iv \rceil_3 \end{array} \right\}, \left\{ \begin{array}{l} iv_2 \leq \|y\|,\ \lceil iv \rceil_3 \leq \|z\| \\ smiv_{ic_4} + smiv_{it_5} \leq \|x\| \} \end{array} \right\} \right\rangle$$

This cost structure represents the upper bound $2(\|y\| + \max(\|x\|, \|x\| \cdot \|z\|))$.

This example illustrates how the complex problem of obtaining a cost structure for a phase is reduced into a set of simpler problems: computation of sums, maximization, minimization of simple constraints. These smaller problems are solved incrementally through strategies that collaborate with each other by adding new constraints to the pending sets. The inference problems in the strategies can be solved efficiently using Farkas' Lemma as they only use the constraint set of one CE at a time.

6.6 Chains

Chains can be formed by one or several phases. If a chain consists solely of one phase, its cost structure is the cost structure of the phase (see Corollary 6.9). This section focuses on chains composed of several phases $ch = ph \cdot ch'$.

The cost structure of a chain $ch = ph \cdot ch'$ can be computed following a similar approach to the one for cost equations (Section 6.4). Based on the first part of Corollary 6.9, we have:

Remark 6.38. Let $\langle E_{ph}, IC_{ph}, FC_{ph}(\boldsymbol{x_s x_f})\rangle$ and $\langle E_{ch'}, IC_{ch'}, FC_{ch'}(\boldsymbol{x_f y_f})\rangle$ be valid cost structures for ph and ch', the following cost structure is valid for any evaluation $T \in [\![C[ch]]\!]_{fc}$ of the chain $ch = ph \cdot ch'$:

$$\left\langle E_{ph} + E_{ch'}, \quad IC_{ph} \cup IC_{ch'}, \quad FC_{ph}(\boldsymbol{x_s x_f}) \cup FC_{ch'}(\boldsymbol{x_f y_f}) \right\rangle$$

As in the CE case, the final constraints have to be transformed to be expressed in terms of the input and output values of the initial call $\boldsymbol{x_s y_s}$. This transformation can also be done as in the CE case but instead of using the constraint set of the CE, the used constraint set is a transitive invariant $\varphi_{ph}(\boldsymbol{x_s y_s x_f y_f})$ that relates the initial and final values of the phase joined with the summary of ch' $summary(ch')(\boldsymbol{x_f y_f})$. Let $\varphi := \varphi_{ph}(\boldsymbol{x_s y_s x_f y_f}) \cup summary(ch')(\boldsymbol{x_f y_f})$, the constraint set FC^+ contains the constraints $\sum \boldsymbol{iv} \bowtie \|l\|$ from FC_{ph} and $FC_{ch'}$ such that their linear expression is guaranteed to be positive $\varphi \Rightarrow l \geq 0$. The transformation performs (Fourier-Motzkin) quantifier elimination on $\exists \boldsymbol{x_f y_f}.(FC^+ \wedge \varphi)$ and obtains a constraint set that relates directly the intermediate variables of FC^+ with $\boldsymbol{x_s y_s}$. Then, new final constraints can be syntactically extracted from the resulting constraint set.

Similarly to the case of CEs, the rest of the constraints $\sum \boldsymbol{iv} \bowtie \|l\|$ (the ones that cannot be guaranteed to be positive) are transformed one by one. They are transformed by inferring the coefficients of a linear template $l'(\boldsymbol{x_s y_s})$ such that $\varphi \Rightarrow l \bowtie l'(\boldsymbol{x_s y_s})$ using Farkas' Lemma. The resulting constraint (if it exists) is $\sum \boldsymbol{iv} \bowtie \|l'\|$.

Example 6.39. Let us compute the cost structure of chain $(3.1 \vee 3.2)^+(2)$ of Program 13 (in Page 76). The cost structures of the phase $(3.1 \vee 3.2)^+$ and chain (2) only contain the constraint $iv_6 = \|y_s + x_s - y_f - x_f\|$ (see Figure 6.3). We have to transform this constraint and express it only in terms of the initial variables of the chain. For this purpose, we consider the transitive relational invariant $\varphi_{(3.1\vee3.2)^+} = \{x_s > x_f, x_s + y_s > y_f, x_{os} = x_{of}, y_{os} = y_{of}\}$ and the chain summary $summary((2)) = \{x_f = x_{of} = 0, y_{of} = y_f\}$, which joined give us φ. Under this constraint set, the expression $y_s + x_s - y_f - x_f$ is guaranteed to be positive and we can perform quantifier elimination over $\exists y_f, x_f, y_{of}, x_{of}.(\varphi \cup \{iv_6 = y_s + x_s - y_f - x_f\})$ obtaining the constraint set $\{iv_6 = x_s + y_s - y_{os}, x_s + y_s \geq y_{os}, x_s > x_{os} = 0\}$. From this constraint set, we can extract the final constraint $iv_6 = \|x_s + y_s - y_{os}\|$ (which corresponds to the constraint in Figure 6.3 modulo variable renaming).

Example 6.40. Let us compute the cost structure of a chain where several phases have non-zero cost. We consider the chain $(4)^+(3)(2)^+(1)$ from Program 11 (this program appears in Page 53 and it has been refined throughout the previous chapter).

In this example, we assume we have already computed the cost structures of chain $(2)^+(1)$ $\langle iv_1, \emptyset, \{iv_1 = \|x - i\|\}\rangle$ and the phases (3) $\langle iv_2 - iv_3, \emptyset, \{iv_2 = \|j_s\|, iv_3 = \|-j_s\|\}\rangle$ and $(4)^+$ $\langle iv_4, \emptyset, \{iv_4 = \|i_f - i_s\|\}\rangle$. Therefore, the next step is to compute the cost of chain $(3)(2)^+(1)$. We also have

Program 16	Refined cost relations
1 void koat(int a){	
2 int b=0;	1.1 : $koat(a) = \{b = 0, a > 0\}, wh3[(3)^+(2.1)](a, b)$
3 while(a>0){	1.2 : $koat(a) = \{b = 0, a \leq 0\}, wh3[(2.2)](a, b)$
4 a--;	2.1 : $wh3(a, b) = \{a \leq 0, b > 0\}, wh7[(5)^+(4)](b)$
5 b=b+a;	2.2 : $wh3(a, b) = \{a \leq 0, b \leq 0\}, wh7[(4)](b)$
6 }	3 : $wh3(a, b) = \{a > 0, a' = a - 1, b' = b + a\}, 1, wh3(a', b')$
7 while(b>0)	4 : $wh7(b) = \{b \leq 0\}$
8 b--;	5 : $wh7(b) = \{b > 0, b' = b - 1\}, 1, wh7(b')$
9 }	

Figure 6.7.: Program 16: Example with non-linear size relation

$summary((2)^+(1)) = \{j_f \geq x_f, x_f \geq i_f + 1\}$ and the transitive invariant $\varphi_{(3)} = \{i_s < x_s, i_s = j_s, i_f = i_s + 1, j_f = j_s, x_f = x_s\}$. Note that because the phase (3) is not iterative, its transitive invariant is simply the constraint set of the cost equation (renamed with the subscripts s and f).

We use $\varphi = \varphi_{(3)} \cup summary((2)^+(1))$ to check which expressions in the final constraints of the components are guaranteed to be non-negative and generate $FC^+ = \{iv_1 = x_f - i_f\}$. Then, we apply quantifier elimination to $\varphi \cup FC^+$ and from the result extract $iv_1 = \|x_s - i_s - 1\|$. The constraints $iv_2 = \|j_s\|$ and $iv_3 = \|-j_s\|$ are not guaranteed to be positive so they are transformed independently. However, such constraints are already expressed in terms of the initial variables and thus remain unchanged. The cost structure for chain $(3)(2)^+(1)$ is

$$\langle iv_1 + iv_2 - iv_3, \quad \emptyset, \quad \{iv_1 = \|x - i - 1\|, iv_2 = \|j\|, iv_3 = \|-j\|\}\rangle$$

Let us now compute the cost structure of chain $(4)^+(3)(2)^+(1)$. We use the cost structure of chain $(3)(2)^+(1)$ (that we have just computed) and of phase $(4)^+$. In this case, we have $\varphi = \varphi_{(4)^+} \cup summary((3)(2)^+1)$ where $\varphi_{(4)^+} = \{i_s < x_s, i_s < j_s, i_s < i_j, j_s = j_f, x_s = x_f\}$ and $summary((3)(2)^+(1)) = \{j_f = i_f, x_f \geq j_f + 2\}$. We generate $FC^+ = \{iv_1 = x_f - i_f - 1, iv_4 = i_f - i_s\}$ and apply quantifier elimination to $\varphi \cup FC^+$. We can extract the following constraint from the resulting constraint set: $iv_1 + iv_4 = \|x_s - i_s - 1\|$. Here again, the constraints $iv_2 = \|j_f\|$ and $iv_3 = \|-j_f\|$ are transformed one by one into $iv_2 = \|j_s\|$ and $iv_3 = \|-j_s\|$ (j is not modified in the phase $(4)^+$). The final cost structure of the chain $(4)^+(3)(2)^+(1)$ is

$$\langle iv_1 + iv_2 - iv_3 + iv_4, \quad \emptyset, \quad \{iv_1 + iv_4 = \|x - i - 1\|, iv_2 = \|j\|, iv_3 = \|-j\|\}\rangle$$

Discussion

This approach is compositional. The cost structures of ph and ch' are computed independently and they are computed only once even though ph or ch' might appear in several chains. This approach is also adequate to compute amortized costs thanks to the quantifier elimination of FC^+ which can generate constraints that relate multiple intermediate variables. In particular, it works well for imperative programs where loops are represented as tail recursion and the relation between $\boldsymbol{y_s}$ and $\boldsymbol{y_f}$ is simple enough to be captured by a linear transitive invariant.

On the other hand, this approach does not work so well when the relation between the initial $\boldsymbol{x_s y_s}$ and the final $\boldsymbol{x_f y_f}$ values of the variables in the phase is not linear. In this case, the relation cannot be captured by a linear transitive invariant and the approach fails to generate the necessary constraints.

Example 6.41. Consider Program 16 (taken from [BEF+16]). The program has been encoded into cost relations such that the second loop is called at the base case of the first loop. This continuation-based encoding is unusual but correct, and it helps to illustrate the point. We focus on the chain $(3)^+(2.1)$. The problem is that the cost structure of the chain (2.1) is $\langle iv, \emptyset, \{iv = \|b_f\|\}\rangle$ but b_f is quadratic with respect

to the original value of a, i.e. $b_f \leq a_s^2$ in phase $(3)^+$. This cannot be captured by a linear transitive invariant over $(3)^+$, the procedure fails to infer any constraint for iv, and consequently, the cost structure of $(3)^+(2.1)$ is unbounded.

Finally, there is no obvious way to generalize this approach for chains that contain multiple recursion. The evaluation of a multiple phase has the shape of a tree in which each of the leaves is the evaluation of a chain. This has two implications. First, it is not possible to compute bounds in terms of the "final" variables of the phase because there is not a single final call but many. Second, the chains that follow a multiple phase are not evaluated only once, but multiple times (once for each leaf of the phase evaluation tree). The number of times one of these chains is evaluated depends on the evaluation of the multiple phase. Therefore, we cannot simply add the cost structures of the multiple phase and the ones of the following chains.

The next section introduces an alternative approach to obtain cost structures of chains that overcomes this limitation, it is applicable to chains with multiple recursion, and can obtain a bound of Program 16.

6.7 Chains with Multiple Recursion

This section contains an alternative algorithm to compute cost structures of chains that can also be applied to chains with multiple recursion. The main idea is to extend the algorithm for phases (Section 6.5) to include the cost of the chain or chains that appear after the given phase.

We consider a general algorithm for chains that start with multiple phases first. Non-multiple iterative phases represent a particular case of this general approach. Let $ch = ph \cdot CH$ and $ph = M(c_1 \vee \cdots \vee c_n)^{+/\omega}$ or $ph = M(c_1 \vee \cdots \vee c_n)^+$, we start from valid cost structures $\langle E_{c_i}, IC_{c_i}, FC_{c_i}(\boldsymbol{x}\boldsymbol{x'})\rangle$ for each c_i and valid cost structures $\langle E_{ch'}, IC_{ch'}, FC_{ch'}(\boldsymbol{x}\boldsymbol{y})\rangle$ for each $ch' \in CH$.

Remark 6.42. Let $T \in [\![C[ch]]\!]_{fc}$ be an arbitrary evaluation. Each c_i is evaluated $\#c_i := |CEinst(T, c_i)|$ times (see Definition 6.7) and $\langle E_{c_ij}, IC_{c_ij}, FC_{c_ij}(\boldsymbol{a_{c_ij}}\boldsymbol{a'_{c_ij}})\rangle$ represents the cost structure instance of the j-th CE evaluation of c_i for $1 \leq j \leq \#c_i$. Similarly, $\#ch' = |maxCH(T, ch')|$ is the number of maximal evaluation trees of chain ch' in T (see Theorem 6.8) and $\langle E_{ch'j}, IC_{ch'j}, FC_{ch'j}(\boldsymbol{a_{ch'j}}\boldsymbol{b_{ch'j}})\rangle$ is the cost structure instance of the j-th maximal evaluation tree of ch' for $1 \leq j \leq \#ch'$. The following cost structure can be evaluated to $Cost(T)$.

$$\left\langle \sum_{i=1}^{n}\sum_{j=1}^{\#c_i} E_{c_ij} + \sum_{ch'\in CH}\sum_{j=1}^{\#ch'} E_{ch'j},\ \left(\bigcup_{i=1}^{n}\bigcup_{j=1}^{\#c_i}(IC_{c_ij}) \cup \bigcup_{ch'\in CH}\bigcup_{j=1}^{\#ch'}(IC_{ch'j})\right),\ \left(\bigcup_{i=1}^{n}\bigcup_{j=1}^{\#c_i}\left(FC_{c_ij}(\boldsymbol{a_{c_ij}}\boldsymbol{a'_{c_ij}})\right) \cup \bigcup_{ch'\in CH}\bigcup_{j=1}^{\#ch'}\left(FC_{ch'j}(\boldsymbol{a_{ch'j}}\boldsymbol{b_{ch'j}})\right)\right)\right\rangle$$

This remark is a direct result of Theorem 6.8 and Definitions 6.7 and 6.6. As in the case for phases, this cost structure has to be transformed to remove the sums over unknowns. For the main cost expressions and non-final constraints, this can be done using the same approach as with phases (see Sections 6.5 and 6.5.1). Sum variables can also be defined for the intermediate variables of the chains $ch' \in CH$ and the transformations are also valid for the constraints originated in ch'. Algorithm 2 now receives a tuple that also contains $IC_{ch'}$ for each $ch' \in CH$ and the returned constraint set is IC_{ch} instead of IC_{ph}. The rest is not modified.

The shape of Algorithm 3 also remains largely the same. In addition to the final constraint sets of the CEs of the phase, it receives the final constraint sets of each $ch' \in CH$ and generates the corresponding pending sets $Psums^{ch'}$ and $Pms^{ch'}$. The algorithm has access to the chains inside CH and its summaries so they can be used by the strategies. Finally, the returned constraint sets are the constraint sets of the complete chain ch, not only of the phase ph.

Next, the strategies for phases are adapted to deal with constraints in these new pending sets and to generate constraints that depend only on the initial input and output initial values $(\boldsymbol{x_s}\boldsymbol{y_s})$ of the chain.

Table 6.3.: Classification conditions for Inductive Sum Strategy in multiple chains: A $c_e \in ph$ or a $ch' \in CH$ named B can be classified into a class with respect to a candidate $cd(\boldsymbol{x}\boldsymbol{y})$ if its condition is satisfied.

Class	Condition when $\bowtie$ is $\leq$	Condition when $\bowtie$ is $\geq$
Cnt	$(\sum \boldsymbol{iv'} \bowtie \|\|l'\|\|) \in Psums^B \wedge \|\|l'\|\| \bowtie cd(\boldsymbol{x}\boldsymbol{y}) - \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k}) \geq 0$	
Dc	$0 \leq dc_B(\boldsymbol{x}) \leq cd(\boldsymbol{x}\boldsymbol{y}) - \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k})$	$dc_B(\boldsymbol{x}) \geq cd(\boldsymbol{x}\boldsymbol{y}) - \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k})$
Ic	$ic_B(\boldsymbol{x}) \geq \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k}) - cd(\boldsymbol{x}\boldsymbol{y})$	$0 \leq ic_B(\boldsymbol{x}) \leq \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k}) - cd(\boldsymbol{x}\boldsymbol{y})$

Inductive Sum Strategy for Multiple Chains

This strategy follows the same scheme of the original *Inductive Sum strategy* and is valid for chains of the form $ch = ph \cdot CH$ where $ph = M(c_1 \vee \cdots c_n)^+$, that is, it is not applicable for phases that might diverge. In this case, the CEs in the phase can have multiple recursive calls $c_i : C(\boldsymbol{x} : \boldsymbol{y}) = \varphi_{c_i}, \boldsymbol{b_0}, C(\boldsymbol{x_1} : \boldsymbol{y_1}), \boldsymbol{b_1}, C(\boldsymbol{x_2} : \boldsymbol{y_2}), \cdots, C(\boldsymbol{x_n} : \boldsymbol{y_n}), \boldsymbol{b_n}$. The function $\#calls(c_i)$ denotes the number of recursive calls of c_i and $C(\boldsymbol{x_k} : \boldsymbol{y_k})$ is the k-th recursive call for $1 \leq k \leq \#calls(i)$. For chains $ch' \in CH$, we define $\#calls(ch') := 0$ and $\varphi_{ch'} := summary(ch')$ so cost equations $c_i \in ph$ and chains $ch' \in CH$ can be treated uniformly.

Given a constraint $\sum \boldsymbol{iv} \bowtie \|l\| \in Psums^A$ where A is a $c_i \in ph$ or a chain $ch' \in CH$, the strategy generates a candidate $cd(\boldsymbol{x}\boldsymbol{y})$ such that:

$$\varphi_A \Rightarrow \left(\|l\| \bowtie cd(\boldsymbol{x}\boldsymbol{y}) - \sum_{k=1}^{\#calls(A)} cd(\boldsymbol{x_k}\boldsymbol{y_k}) \geq 0 \right)$$

This is a direct generalization of the condition used for linear phases. Note that if A is a chain ch', $\#calls(A)$ is zero and the condition is $\varphi_A \Rightarrow (\|l\| \bowtie cd(\boldsymbol{x}\boldsymbol{y}) \geq 0)$.

Once a candidate has been generated, the CEs $c_e \in ph$ and the chains $ch' \in CH$ have to be classified according to their behavior with respect to the candidate. This strategy has the same classes as before but the conditions have been generalized for CEs with multiple recursive calls and for chains.

Let B be a $c_e \in ph$ or a $ch' \in CH$, Table 6.3 contains the classes and their conditions. The classes *Cnt*, *Dc* and *Ic* also define the expressions $cnt_B := \sum \boldsymbol{smiv'}$, $iv_{dc_B} := \|dc_B(\boldsymbol{x})\|$ and $iv_{ic_B} := \|ic_B(\boldsymbol{x})\|$ respectively. Note that when B is a chain, the expression $cd(\boldsymbol{x}\boldsymbol{y}) - \sum_{k=1}^{\#calls(B)} cd(\boldsymbol{x_k}\boldsymbol{y_k})$ simply corresponds to $cd(\boldsymbol{x}\boldsymbol{y})$.

Theorem 6.43. *If every $c_e \in ph$ and every $ch' \in CH$ is classified into Cnt, Ic or Dc with respect to a candidate $cd(\boldsymbol{x}\boldsymbol{y})$, the following constraints are valid:*

$$\sum_{B \in Cnt} cnt_B \bowtie iv_{cd+} - iv_{cd-} + \sum_{B \in Ic} smiv_{ic_B} - \sum_{B \in Dc} smiv_{dc_B} \qquad \begin{array}{l} iv_{cd+} \bowtie \|cd(\boldsymbol{x_s}\boldsymbol{y_s})\| \\ iv_{cd-}! \bowtie \|-cd(\boldsymbol{x_s}\boldsymbol{y_s})\| \end{array}$$

Finally, the strategy adds the constraints $iv_{ic_B} \bowtie \|ic_B(\boldsymbol{x})\|$ and $iv_{dc_B} \bowtie \|dc_B(\boldsymbol{x})\|$ to the corresponding pending sets $Psums^B$ for each $B \in Ic$ and $B \in Dc$.

Table 6.4.: Classification conditions for Inductive Sum Strategy wit Resets in multiple chains: A $c_e \in ph$ or a $ch' \in CH$ named B can be classified into a class with respect to a candidate $cd(\boldsymbol{x}\,\boldsymbol{y})$ if its condition is satisfied.

Class	Condition
CntR	$(\sum \boldsymbol{iv'} \leq \|\|l'\|\|) \in Psums^B \wedge \|\|l'\|\| \leq \|\|cd(\boldsymbol{x})\|\| - \sum_{k=1}^{\#calls(B)} \|\|cd(\boldsymbol{x_k})\|\|$
DcR	$0 \leq \|\|cd(\boldsymbol{x})\|\| - \sum_{k=1}^{\#calls(B)} \|\|cd(\boldsymbol{x_k})\|\|$
IcR	$icr_B(\boldsymbol{x}) \geq \sum_{k=1}^{\#calls(B)} \|\|cd(\boldsymbol{x_k})\|\| - \|\|cd(\boldsymbol{x})\|\|$

Inductive Sum Strategy with Resets for Multiple Chains

As in the linear case, Inductive Sum Strategy with Resets can be applied to chains with multiple phases that might diverge, that is, chains of the form $ch := ph \cdot CH$ where $ph := M(c_1 \vee \cdots c_n)^{+/\omega}$ and it generates only upper bound constraints.

The candidates are generated using the condition of class *CntR* (see Table 6.4) and only depend on the input variables $cd(\boldsymbol{x})$. The strategy considers the classes *CntR*, *DcR*, and *IcR* from Table 6.4. In the case of multiple recursion, the conditions from the Inductive Sum Strategy cannot be reused. The conditions for this strategy have to always consider the positive part of the candidates ($\|\|cd(\boldsymbol{x})\|\|$) not only for *CntR* but also for *DcR* and *IcR*. This is necessary for the soundness of the strategy. Given these conditions, resets become a special case of *IcR* and thus no distinct class *Rst* is considered. It is also worth noting that the condition for the *DcR* is trivial for chains $ch' \in CH$ because the right-hand side $\|\|cd(\boldsymbol{x})\|\| - \sum_{k=1}^{\#calls(ch')} \|\|cd(\boldsymbol{x_k})\|\| = \|\|cd(\boldsymbol{x})\|\|$ is always positive. Consequently, the strategy can always classify chains into *DcR* where they are ignored.

Theorem 6.44. *If every $c_e \in ph$ and every $ch' \in CH$ is classified into CntR, DcR, and IcR with respect to a candidate $cd(\boldsymbol{x})$, the following constraints are valid:*

$$\sum_{B \in CntR} cntr_B \leq iv_{cd} + \sum_{B \in IcR} smiv_{icr_B} \qquad iv_{cd} \leq \|\|cd(\boldsymbol{x_s})\|\|$$

For each $B \in IcR$, the strategy adds the constraint $iv_{icr_B} \leq \|\|icr(\boldsymbol{x})\|\|$ to $Psums^B$.

Basic Product Strategy

The basic product strategy does not require any changes, it can be directly applied to constraints from pending sets of chains.

Max-Min Strategy for Multiple Chains

This strategy requires no changes in the candidate generation and in the constraint generation. Theorems 6.33 and 6.34 are also valid for constraints $iv \geq \|\|l\|\| \in Pms^{ch'}$ and $iv \leq \|\|l\|\| \in Pms^{ch'}$ and for phases with multiple recursion. However, the classification differs. In this strategy, the chains $ch' \in CH$ do not need to be classified and the classification conditions are given in Table 6.5. Instead of considering the sum of the values of the candidate in all the recursive calls, these conditions consider these values independently.

Example 6.45. Let us consider Program 4 (in Page 11) and its chain $M(4^c \vee 5^c)^+\{(3^c)\}$. Its cost relations are in Figure 5.4 and they have been strengthened in Example 5.66 (in Page 74). For simplicity, we omit the superscript of the CEs and we assume the cost structures of its components have been already computed. The cost structure of (3) is empty, and the cost structures for 4 and 5 are $\langle iv_1+1, \emptyset, \{iv_1 = t1\}\rangle$ and $\langle 1, \emptyset, \emptyset\rangle$ respectively ($t1 = l1$ is the cost of the call to $append[(2)^+(1)]$). The simplified constraint sets are:

CE/Chain	Constraint set
4	$\{t = 1+t1+t2, t2 \geq 0, t1 \geq 1, t = l\}$
5	$\{t = 1+t1+t2, t1 \geq 0, t2 \geq 0, l = t2+1, t1 = 0\}$
(3)	$\{t = l = 0\}$

The initial pending sets are $Psums^4 = \{iv_{it_4} \leq 1, iv_{it_4} \geq 1, iv_1 \leq \|t1\|, iv_1 \geq \|t1\|\}$ and $Psums^5 = \{iv_{it_5} \leq 1, iv_{it_5} \geq 1\}$. The cost structure computation is as follows:

1. Consider $iv_{it_4} \leq 1 \in Psums^4$. The algorithm applies the Inductive Sum Strategy and generates a candidate t. The classification is $Cnt := \{4,5\}$ and $Dc := \{(3)\}$ with $dc_{(3)} := 0$ and the generated constraints are $smiv_{it_4} + smiv_{it_5} \leq iv_{cd^+} - iv_{cd^-}$, $iv_{cd^+} \leq \|t\|$, and $iv_{cd^-} \geq \|-t\|$ which can be simplified to $smiv_{it_4} + smiv_{it_5} \leq \|t\|$.

2. The processing of $iv_{it_4} \geq 1$ is symmetric and its generated (and simplified) constraint is $smiv_{it_4} + smiv_{it_5} \geq \|t\|$.

3. The Inductive Sum Strategy fails to generate a candidate for $iv_1 \leq \|t1\|$. Instead, the Basic Product Strategy can be applied and it generates $smiv_1 \leq smiv_{it_4} \cdot \lceil iv \rceil_1$. The variable $smiv_{it_4}$ has already been bounded so $iv_{it_4} \leq 1$ does not need to be added to the pending sets. The constraint $iv_1 \leq \|t1\|$ is added to Pms^4.

4. Consider $iv_1 \leq \|t1\| \in Pms^4$. The algorithm applies the Max-Min strategy and it generates the candidate t ($t \geq t1$). Both CEs 4 and 5 can be classified in Dc and the generated constraint is $\lceil iv \rceil_1 \leq t$.

5. Finally, for $iv_1 \geq \|t1\| \in Psums^4$ no non-trivial constraint can be obtained.

The resulting cost structure is:

$$\left\langle smiv_1 + smiv_{it_4} + smiv_{it_5}, \quad \{smiv_1 \leq smiv_{it_4} \cdot \lceil iv \rceil_1\}, \quad \{\lceil iv \rceil_1 \leq t, smiv_{it_4} + smiv_{it_5} = \|t\|\}\right\rangle$$

which represents an upper bound $(1+t)\cdot t$ and a lower bound of t. Note that the lower bound is precise for the given cost model. It corresponds to the case where the input argument t is a degenerate tree and *append* is always called with $l1 =$ Nil.

Linear Phases as a Special Case

The approach for chains with multiple recursion can also be applied for chains $ch := ph \cdot ch'$. It corresponds to having a set CH with only one element ch' (see Corollary 6.9).

The only particularity is that in such a chain, we know that $\#ch' = 1$ and consequently for intermediate variables from ch, we have $smiv = \lceil iv \rceil = \lfloor iv \rfloor$. Therefore, the *Basic Product strategy* can be simplified for those cases. Let $\sum \boldsymbol{iv} \leq \|l\| \in Psums^{ch'}$, the modified strategy generates the constraint $\sum \boldsymbol{smiv} \leq \lceil iv \rceil'$ and adds $iv' \leq \|l\|$ to $Pms^{ch'}$ where iv' is a fresh intermediate variable. Similarly, let $\sum \boldsymbol{iv} \geq \|l\| \in Psums^{ch'}$, it generates the constraint $\sum \boldsymbol{smiv} \geq \lfloor iv \rfloor'$ and adds $iv' \geq \|l\|$ to $Pms^{ch'}$.

Table 6.5.: Classification conditions for Max-Min strategy in multiple chains: A c_e can be classified into the classes *Dc*, *Ic* or *Rst* with respect to a candidate $cd(\boldsymbol{x})$ if its conditions are satisfied.

Class	Condition when $\bowtie$ is $\leq$	Condition when $\bowtie$ is $\geq$
Dc	$\bigwedge_{k=1}^{\#calls(c_e)} \left(0 \leq dc_B(\boldsymbol{x}) \leq cd(\boldsymbol{x}) - cd(\boldsymbol{x_k})\right)$	$\bigwedge_{k=1}^{\#calls(c_e)} \left(dc_B(\boldsymbol{x}) \geq cd(\boldsymbol{x}) - cd(\boldsymbol{x_k})\right)$
Ic	$\bigwedge_{k=1}^{\#calls(c_e)} \left(ic_B(\boldsymbol{x}) \geq cd(\boldsymbol{x_k}) - cd(\boldsymbol{x})\right)$	$\bigwedge_{k=1}^{\#calls(c_e)} \left(0 \leq ic_B(\boldsymbol{x}) \leq cd(\boldsymbol{x_k}) - cd(\boldsymbol{x})\right)$
Rst	$\bigwedge_{k=1}^{\#calls(c_e)} \left(cd(\boldsymbol{x_k}) \bowtie \|rst_B(\boldsymbol{x})\|\right)$	

Example 6.46. This approach obtains bounds for Program 16. Consider the chain $(3)^+(2.1)$. The cost structures for CE 3 and chain (2.1) are $\langle 1, \emptyset, \emptyset \rangle$ and $\langle iv_1, \emptyset, \{iv_1 = \|b\|\} \rangle$ respectively. The main cost expression is $smiv_{it_3} + smiv_1$ and the initial pending sets are $Psums^{(2.1)} = \{iv_1 \leq \|b\|, iv_1 \geq \|b\|\}$ and $Psums^3 = \{iv_{it_3} \leq 1, iv_{it_3} \geq 1\}$. The cost structure computation is as follows:

1. First, the constraints $iv_{it_3} \leq 1$ and $iv_{it_3} \geq 1$ can be processed by the Inductive Sum Strategy. The generated (simplified) constraints are $smiv_{it_3} \leq \|a\|$ and $smiv_{it_3} \geq \|a\|$.

2. Next, the constraint $iv_1 \leq \|b\| \in Psums^{(2.1)}$ is selected. The Inductive Sum strategy generates the candidate b. The classification is $Cnt := \{(2.1)\}$ and $Ic := \{3\}$ with $ic_3 := \|a\|$. The generated (simplified) constraints are $smiv_1 \leq iv_2 + smiv_{ic_3}$ and $iv_2 \leq \|b\|$. The constraint $iv_{ic_3} \leq \|a\|$ is added to the pending set $Psums^3$. The treatment of $iv_1 \geq \|b\| \in Psums^{(2.1)}$ is symmetric.

3. The constraint $iv_{ic_3} \leq \|a\| \in Psums^3$ can be processed using the Basic Product strategy, which generates $smiv_{ic_3} \leq smiv_{it_3} \cdot \lceil iv \rceil_{ic_3}$, followed by the Max-Min strategy which generates $\lceil iv \rceil_{ic_3} \leq \|a\|$. This is enough to obtain an upper bound. The cost structure at this point is:

$$\left\langle smiv_{it_3} + smiv_1, \quad \left\{ \begin{array}{l} smiv_1 = iv_2 + smiv_{ic_3} \\ smiv_{ic_3} \leq smiv_{it_3} \cdot \lceil iv \rceil_{ic_3} \end{array} \right\}, \quad \left\{ \begin{array}{l} smiv_{it_3} = \|a\|, \\ iv_2 = \|b\|, \\ \lceil iv \rceil_{ic_3} \leq \|a\| \end{array} \right\} \right\rangle$$

   and the corresponding upper bound is $\|a\| + \|b\| + (\|a\|^2)$.

4. Alternatively, the constraints $iv_{ic_3} \leq \|a\|$ and $iv_{ic_3} \geq \|a\|$ can also be processed using the Triangular Sum strategy with $q = 1$. The resulting (simplified) cost structure is:

$$\left\langle smiv_{it_3} + smiv_1, \quad \left\{ \begin{array}{l} smiv_1 = iv_2 + smiv_{ic_3} \\ smiv_{ic_3} = iv_{p1} + \frac{1}{2} iv_{p2} - \frac{1}{2} iv_{its} \\ iv_{p1} = iv_{ini} \cdot iv_{its} \\ iv_{p2} = smiv_{it_3} \cdot iv_{its} \\ iv_{its} = smiv_{it_3} \end{array} \right\}, \quad \left\{ \begin{array}{l} smiv_{it_3} = \|a\|, \\ iv_2 = \|b\|, \\ iv_{ini} = \|a\| \end{array} \right\} \right\rangle$$

   which represents the precise cost (upper and lower bound) $\|a\| + \|b\| + (\|a\|^2/2 + \|a\|/2)$

This approach can obtain bounds for programs where the approach based on composing the cost structures of the individual phases fails. This is because using this approach, the strategies for Sum and Max variables are used to infer non-linear size relations whereas the compositional approach relies on linear summaries that cannot capture such size relations. That is precisely the case with Program 16.

However, this alternative does not solve all the limitations of the analysis with respect to non-linear size relations and it presents other disadvantages. Namely, this approach is less modular. Instead of

computing a cost structure for the phase once, a computation has to be done for each chain that starts with such phase. In addition, it is harder to generate good candidates for the Inductive Sum strategy and obtain expressions that depend on the output as well. That is, it is harder to obtain amortized bounds using this approach.

A more detailed account of the limitations of the current analysis with respect to non-linear size relations is given in Chapter 10.

6.8 Solving Cost Structures

Up to this point, we have seen how to obtain cost structures for the chains of a cost relation. Cost structures give us very precise information about the cost of cost relations, but they can be complex and hard to interpret. Therefore, we want to obtain upper or lower bounds instead. Theorem 6.16 guarantees that a cost structure bound also represents a bound of the chain. Thus, this section focuses on obtaining cost structure bounds.

Recall the definition of cost structure upper bound (Definition 6.15 in Page 81). A function $f(\boldsymbol{x})$ is a cost structure upper bound of $\langle E, IC, FC(\boldsymbol{xy})\rangle$ if $IC \wedge FC(\boldsymbol{xy}) \Rightarrow f(\boldsymbol{x}) \geq E$. A cost structure upper bound can be computed incrementally. First, final constraints are transformed so they only contain input variables $FC(\boldsymbol{x})$ using the corresponding chain summary[8]. Then, starting from the main cost expression $f_0 := E$, the bound computation procedure rewrites $f_i \rightarrow_{ub} f_{i+1}$ repeatedly such that $IC \wedge FC(\boldsymbol{x}) \Rightarrow f_i \leq f_{i+1}$. This process substitutes intermediate variables by their bounds until no intermediate variables are left. At that point f_i is a valid cost structure upper bound.

Let $(\sum_{i=1}^{n} iv \leq e) \in IC \cup FC(\boldsymbol{x})$, we have that $e_1 iv_1 + e_2 iv_2 + \cdots + e_n iv_n \leq \max(e_1, e_2, \cdots, e_n)\|e\|$ holds for arbitrary expressions $e_1, e_2, \cdots, e_n$. In the particular case where $n = 1$, we simply have $iv_1 \leq \|e\|$. Similarly, let $(\sum_{i=1}^{n} iv \geq e) \in IC \cup FC(\boldsymbol{x})$, we have that $e_1 iv_1 + e_2 iv_2 + \cdots + e_n iv_n \geq \min(e_1, e_2, \cdots, e_n)\|e\|$ and if $n = 1$, then $iv_1 \geq \|e\|$. Based on this, the following set of rewrite rules are defined:

$$\begin{aligned}\mathscr{R} := &\left\{ (e_1 iv_1 + e_2 iv_2 + \cdots + e_n iv_n) \rightarrow_{\leq} (\max(e_1, e_2, \cdots, e_n)\|e\|) \;\middle|\; \textstyle\sum_{i=1}^{n} iv \leq e \in (IC \cup FC(\boldsymbol{x})) \right\} \\ &\cup \left\{ (e_1 iv_1 + e_2 iv_2 + \cdots + e_n iv_n) \rightarrow_{\geq} (\min(e_1, e_2, \cdots, e_n)\|e\|) \;\middle|\; \textstyle\sum_{i=1}^{n} iv \geq e \in (IC \cup FC(\boldsymbol{x})) \right\}\end{aligned}$$

Now, it is left to define how to apply these rules. For upper bounds, the rewrite steps maximize sub-expressions that appear positively and minimize expressions that appear negatively. Given that intermediate variables are always positive and other expressions always appear inside the operator $\|\|$, it is always possible to determine syntactically if a sub-expression appears positively or negatively. For instance, in an expression $iv_1 - (iv_2 - iv_3)$, the sub-expression iv_1 appears positively, $(iv_2 - iv_3)$ and iv_2 appear negatively, and iv_3 appears positively.

The rewrite rules can be applied as follows. Let an expression f such that $f_{|\pi}$ (the sub-expression at position π) matches l for a rule $l \rightarrow_{\leq} r \in \mathscr{R}$ and $f_{|\pi}$ appears *positively*, the sub-expression l can be substituted by r at that position $f \rightarrow_{ub} f[l/r]_{\pi}$. Conversely, if f_{π} matches l such that $l \rightarrow_{\geq} r \in \mathscr{R}$ and $f_{|\pi}$ appears *negatively*, the sub-expression l can be substituted by r at that position $f \rightarrow_{ub} f[l/r]_{\pi}$.

For lower bounds, f has to be rewritten $f_i \rightarrow_{lb} f_{i+1}$ such that $IC \wedge FC(\boldsymbol{x}) \Rightarrow f_i \geq f_{i+1}$. Therefore, the rewrite rules can be applied inversely. Let an expression f such that $f_{|\pi}$ matches l for a rule $l \rightarrow_{\geq} r \in \mathscr{R}$ and $f_{|\pi}$ appears positively, then $f \rightarrow_{lb} f[l/r]_{\pi}$. Conversely, if f_{π} matches l such that $l \rightarrow_{\leq} r \in \mathscr{R}$ and $f_{|\pi}$ appears negatively, then $f \rightarrow_{lb} f[l/r]_{\pi}$.

It is certainly possible that no rule can be matched directly. However, constraints of the form $\sum_{i=1}^{n} iv \leq e$ can always be split into $iv_i \leq e$ for $1 \leq i \leq n$ which in turn generate the corresponding rules $iv_i \rightarrow_{\leq} e$ and individual intermediate variables can always be matched. Furthermore, we know that intermediate

[8] For each $\sum \boldsymbol{iv} \bowtie \|l(\boldsymbol{xy})\|$, we infer $l'(\boldsymbol{x})$ such that $summary(ch) \Rightarrow l(\boldsymbol{xy}) \bowtie l'(\boldsymbol{x})$ and generate $\sum \boldsymbol{iv} \bowtie \|l'(\boldsymbol{x})\|$. If no l' is found, no constraint is generated.

variables are always positive so for every iv we have the trivial rules $iv \to_{\geq} 0$ and $iv \to_{\leq} \omega$. Consequently, the procedure can always obtain a cost structure bound.

Example 6.47. Consider the cost structure of chain [1.2] of Program 13:

$$\langle 1iv_2 + 2iv_6, \quad \{iv_2 = iv_3 \cdot iv_4\}, \quad \{iv_3 + iv_6 = \|y + x\|, iv_4 = \|z\|\}\rangle$$

We have the following (numbered) rules[9]:

$$\begin{array}{lll} 1\colon iv_2 \to_{\leq} iv_3 \cdot iv_4 & 3\colon iv_4 \to_{\leq} \|z\| & 5\colon e_1 iv_3 + e_2 iv_6 \to_{\leq} \max(e_1, e_2) \cdot \|y + x\| \\ 2\colon iv_2 \to_{\geq} iv_3 \cdot iv_4 & 4\colon iv_4 \to_{\geq} \|z\| & 6\colon e_1 iv_3 + e_2 iv_6 \to_{\geq} \min(e_1, e_2) \cdot \|y + x\| \end{array}$$

We can obtain an upper bound with the following rewrite sequence ($\to^n_{ub}$ indicates that rule n is applied):

$$1iv_2 + 2iv_6 \quad \to^1_{ub} \quad iv_3 \cdot iv_4 + 2iv_6 \quad \to^3_{ub} \quad iv_3 \cdot \|z\| + 2iv_6 \quad \to^5_{ub} \quad \max(\|z\|, 2) \cdot \|y + x\|$$

Note that we assume that arithmetic simplifications can be applied in between rule applications. In particular, the product $iv_3 \cdot \|z\|$ can be rewritten as $\|z\| \cdot iv_3$ so rule 5 can be matched.

Similarly, a lower bound can be obtained as follows:

$$1iv_2 + 2iv_6 \quad \to^2_{lb} \quad iv_3 \cdot iv_4 + 2iv_6 \quad \to^4_{lb} \quad iv_3 \cdot \|z\| + 2iv_6 \quad \to^6_{lb} \quad \min(\|z\|, 2) \cdot \|y + x\|$$

Matching rules with several intermediate variables on their left side (such as 5 and 6) is harder than matching rules with a single intermediate variable. We can simplify the process by generating the rules $iv_3 \to_{\leq} \|y+x\|$ and $iv_3 \to_{\leq} \|y+x\|$ instead of rule 5 and rewrite $iv_3 \cdot \|z\| + 2iv_6$ to $\|y+x\| \cdot \|z\| + 2\|y+x\|$. The resulting expression is also a valid upper bound although it is less precise. Unfortunately, this simplification cannot be applied to rule 6.

Note that given a cost structure, there might be multiple bound expressions that can be extracted depending on which rules are considered. Moreover, the different possible bound expressions are often not comparable among each other. For instance, given a cost structure $\langle iv, \emptyset, \{iv \leq \|x\|, iv \leq \|y\|\}\rangle$, both $\|x\|$ and $\|y\|$ are valid upper bounds. These upper bounds are not comparable (we do not know whether x is bigger that y or not) thus the best upper bound is $\min(\|x\|, \|y\|)$. However, the current implementation prioritizes efficiency over precision. Instead of trying to obtain the best bound considering all the constraints, it performs a heuristic preselection of the constraints to be considered. This preselection tries to minimize the asymptotic complexity of the upper bounds and maximize the asymptotic complexity of the lower bounds.

6.9 Piece-Wise Symbolic Bounds

The approach presented so far computes upper and lower bounds of all the chains of one or several cost relations. However, we are usually interested in a bound for any evaluation of a CR, not for a specific chain. In order to obtain such a bound, the bounds from each of the chains have to be combined.

The simplest approach is to take the maximum of the upper bounds and the minimum of the lower bounds.

Corollary 6.48 (CR bound)**.** *Let CH be the set of chains in CR C and let f_{ch} be an upper bound of $C[ch]$ for each $ch \in CH$. The function $f := \max_{ch \in CH} f_{ch}$ is an upper bound of C. Conversely, let g_{ch} be an lower bound of $C[ch]$ for each $ch \in CH$, the function $g := \min_{ch \in CH} g_{ch}$ is a lower bound of C.*

[9] In principle rules 1 and 2 should be $iv_2 \to_{\leq} \|iv_3 \cdot iv_4\|$ and $iv_2 \to_{\geq} \|iv_3 \cdot iv_4\|$ but they can be simplified because iv_3 and iv_4 (and any other intermediate variable) are always non-negative.

This result follows directly from the completeness of the refinement. This approach can suffice if we are only interested in the worst case asymptotic complexity of a cost relation but it can be insufficient if we want a precise upper bound. For lower bounds, this naive approach will almost always generate a trivial constant bound.

Example 6.49. Program 8 (in Figure 1.14 in Page 15) has the following chains, chain summaries and bounds:

Chain	Summary	Ub	Lb	Chain	Summary	Ub	Lb
$(6)^+(4)$	$\{n \geq 1, l \geq n, n + ret = l\}$	$\|\|n\|\|$	$\|\|n\|\|$	(4)	$\{0 \geq n, ret = l\}$	0	0
$(6)^+(5)$	$\{l \geq 1, n > l, 0 = ret\}$	$\|\|l\|\|$	$\|\|l\|\|$	(5)	$\{l = 0, n \geq 1, ret = 0\}$	0	0

If we take the maximum of the upper bounds, we obtain $\max(n, l)$ instead of the precise bound $\min(n, l)$ (the bound $\|\|n\|\|$ has the condition that $l \geq n$ and the bound $\|\|l\|\|$ has the opposite condition $n > l$). In the case of lower bounds, the imprecision is even more significant. We obtain the trivial lower bound $\min(n, 0) = 0$.

This imprecision comes from the fact that we have completely ignored the information contained in the chain summaries which tells us when each chain is applicable. Chain summaries can be seen as necessary preconditions (see Definition 5.26). Note though that they are not sufficient preconditions so summaries of different chains can be compatible (not mutually exclusive).

Example 6.50. The cost relation p in Program 13 contains two chains whose summaries are not all mutually exclusive:

Chain	Summary	Ub	Lb
(1.1)	$\{x > 0, y > 0, z > 0\}$	$2\|\|x + y\|\|$	$2\|\|x + y\|\|$
(1.2)	$\{x > 0, y > 0, z > 0\}$	$\max(\|\|z\|\|, 2) \cdot \|\|x + y\|\|$	$\min(\|\|z\|\|, 2) \cdot \|\|x + y\|\|$

Despite not being mutually exclusive, we can use the constraints in the chain summaries to partition the input space and obtain more precise bounds for each of the partitions. In this manner, we obtain a piece-wise defined bound function.

Example 6.51. In the case of Program 8, the summaries are already mutually exclusive, that is, they represent a partition of the input space. Therefore, the piece-wise upper and lower bound of Program 8 is:

$$f(l,n) = \begin{cases} \|n\| & \text{if } l \geq n \geq 1 \\ \|l\| & \text{if } n \geq l \geq 1 \\ 0 & \text{if } (n \leq 0) \vee (n \geq 1 \wedge l = 0) \end{cases}$$

In general, given a CR C with input variables $\boldsymbol{x}$, a partition of its input space is a set of constraint sets $P_C := \{\varphi_1(\boldsymbol{x}), \cdots \varphi_n(\boldsymbol{x})\}$ such that they cover the complete input space ($\varphi_1(\boldsymbol{x}) \vee \cdots \vee \varphi_n(\boldsymbol{x}) = \top$) and they are all incompatible, i.e. for all $i, j \in [1, n]$ such that $i \neq j$ $\varphi_i(\boldsymbol{x}) \wedge \varphi_j(\boldsymbol{x}) = \bot$. Given a partition P_{ch} and a chain set CH, a valid allocation is a function $\text{alloc} : P_C \to \mathscr{P}(CH)$ (where $\mathscr{P}(CH)$ is the power set of CH) such that $ch \in \text{alloc}(\varphi)$ if and only if $summary(ch) \wedge \varphi$ is satisfiable.

Theorem 6.52 (CR Piece-Wise Bound)**.** *Let C be a cost relation with chains CH, upper bounds f_{ch} and lower bounds g_{ch} for each chain $ch \in CH$. Let $P_C := \{\varphi_1, \cdots, \varphi_n\}$ be a partition of the input space and let*

alloc *be a valid allocation of CH to* P_C*. The following functions* f *and* g *are respectively upper and lower bound of C.*

$$f := \begin{cases} \max\limits_{ch \in \text{alloc}(\varphi_1)} (f_{ch}) & \text{if } \varphi_1 \\ \vdots & \vdots \\ \max\limits_{ch \in \text{alloc}(\varphi_n)} (f_{ch}) & \text{if } \varphi_n \end{cases} \qquad g := \begin{cases} \min\limits_{ch \in \text{alloc}(\varphi_1)} (g_{ch}) & \text{if } \varphi_1 \\ \vdots & \vdots \\ \min\limits_{ch \in \text{alloc}(\varphi_n)} (g_{ch}) & \text{if } \varphi_n \end{cases}$$

Proof. Let $T \in [\![C]\!]_{fc}$ with parameters $(\boldsymbol{a} : \boldsymbol{b})$. Because P_C is a partition, only one $\varphi_i \in P_C$ is satisfied by the input parameters $\boldsymbol{a}$ ($\models \varphi_i(\boldsymbol{a})$). By refinement completeness, we know there is a chain $ch_j \in CH$ such that $T \in [\![C[ch_j]]\!]_{fc}$. Given the definition of chain summaries $summary(ch_j)$ is satisfiable with the parameters $\boldsymbol{ab}$ so we have that $\varphi_i \wedge summary(ch_j)$ must be satisfiable. Therefore, $ch_j \in \text{alloc}(\varphi_i)$. Finally, We have that f_{ch_j} is a valid upper bound for every evaluation $T \in [\![C[ch_j]]\!]_{fc}$, therefore:

$$Cost(T) \leq f_{ch_j}(\boldsymbol{a}) \leq \max_{ch \in \text{alloc}(\varphi_i)} f_{ch}(\boldsymbol{a}) = f(\boldsymbol{a})$$

The proof for lower bounds is analogous. □

The partitioning of the input space can be done by creating a *Binary Space Partitioning Tree* (BSP) that allocates the chains and their bounds according to their summary. A linear constraint $lc := l \geq 0$ splits the input space in two semi-spaces. The subspace that satisfies $l \geq 0$ and the one that satisfies its complement $l < 0$. Constructing a BSP can be done by recursively dividing the input space using linear constraints. We use the linear constraints that appear in the chain summaries. Given a linear constraint lc, each chain ch can be allocated into the first sub-space if $lc \Rightarrow summary(ch)$, into the second sub-space if $\neg lc \Rightarrow summary(ch)$, or into both if none of the implications hold. The total number of partitions can grow very large (at most 2^n partitions where n is the number of constraints in all the chain summaries). Fortunately, the process can be stopped at any moment, for instance, by limiting the maximum depth of the partitioning tree. Thus, we can trade precision and efficiency.

6.10 Proofs

This section contains the proofs for the cost composition theorem (Theorem 6.8) and the proofs for the theorems corresponding to the strategies for the phase (and multiple recursive chain) cost structure computation.

6.10.1 Theorem 6.8: Cost Composition

We have to prove that for any evaluation $T \in [\![C[ch]]\!]_{fc}$ where $ch = ph \cdot CH$:

$$Cost(T) = Cost_{ph}(T) + \sum_{ch' \in CH} \Big(\sum_{\pi \in maxCH(T,ch')} Cost(T_{|\pi}) \Big)$$

where $maxCH(T, ch') = \{\pi \mid T_{|\pi} \text{ is a maximal tree of } ch'\}$ for each $ch' \in CH$. We apply the definition of phase cost (Definition 6.7) and we have the expression

$$Cost(T) = \sum_{c \in ph} \Big(\sum_{\pi \in CEinst(T,c)} Cost_c(T_{|\pi}) \Big) + \sum_{ch' \in CH} \Big(\sum_{\pi \in maxCH(T,ch')} Cost(T_{|\pi}) \Big)$$

where $CEinst(T, c) = \{\pi \mid label(T_{|\pi}) = c\}$ for each $c \in ph$.

Given that all the sets $maxCH(T, ch')$ and $CEinst(T, c)$ are disjoint, we consider only two sets $maxCH(T)$ and $CEinst(T)$ that are the union of all $maxCH(T, ch')$ for $ch' \in CH$ and $CEinst(T, c)$ for $c \in ph$ respectively.

$$CEinst(T) = \bigcup_{c \in ph} CEinst(T, c) \qquad maxCH(T) = \bigcup_{ch' \in CH} maxCH(T, ch')$$

The resulting expression is

$$Cost(T) = \sum_{\pi \in CEinst(T)} Cost_c(T_{|\pi}) + \sum_{\pi \in maxCH(T)} Cost(T_{|\pi})$$

We prove this expression by induction on the size of CE evaluations $|CEinst(T)|$.

Base case

In the base case ($n = 1$) we have $T = t(c_i(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![C[ch]]\!]_{fc}$, such that $c_i \in ph$ and $CEinst(T) = \{\epsilon\}$.

$$\begin{aligned}
Cost(T) &=^{(1)} \sum_{k=1}^{n} Cost(T_k) \\
&=^{(2)} \sum_{i \in NRec} Cost(T_i) + \sum_{i \in Rec} Cost(T_i) + \sum_{i \in Trunc} Cost(T_i) \\
&=^{(3)} \sum_{i \in NRec} Cost(T_i) + \sum_{i \in Rec} Cost(T_i) \\
&=^{(4)} Cost_c(T) + \sum_{i \in Rec} Cost(T_i) \\
&=^{(5)} \sum_{\pi \in CEinst(T)} Cost_c(T_{|\pi}) + \sum_{i \in Rec} Cost(T_i) \\
&=^{(6)} \sum_{\pi \in CEinst(T)} Cost_c(T_{|\pi}) + \sum_{\pi \in maxCH(T)} Cost(T_{|\pi})
\end{aligned}$$

1. Definition of *Cost* (Definition 3.8)
2. Let $c\colon C(\boldsymbol{x} : \boldsymbol{y}) = \varphi, b_1, \ldots, b_n$, we divide $T_1, \cdots, T_n$ in three sets:
   a) The evaluations with recursive calls that have been truncated because of a selected partial evaluation $Trunc = \{i \mid b_i = C(\boldsymbol{x}_i : \boldsymbol{y}_i) \wedge label(T_i) = \bot\}$
   b) The evaluations with recursive calls that have not been truncated $Rec = \{i \mid b_i = C(\boldsymbol{x}_i : \boldsymbol{y}_i) \wedge label(T_i) \neq \bot\}$
   c) The evaluations with non-recursive calls $Nrec = \{i \mid T_i \notin [\![C]\!]_{fc}\}$
3. For each $i \in Trunc$ we have $Cost(T_i) = Cost(0) = 0$
4. Definition of cost of CE (Definition 6.6)
5. Because of the value of $CEinst(T) = \{\epsilon\}$ and $T_{|\epsilon} = T$
6. For $i \in Rec$ we have that $T_i \in [\![C[ch']]\!]_{fc}$ for $ch' \in CH$. Therefore, $Rec = maxCH(T)$ and $\sum_{i \in Rec} Cost(T_i) = \sum_{\pi \in maxCH(T)} Cost(T_{|\pi})$

Inductive step

We have $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![C[ch]]\!]_{fc}$ with $c \in ph$.

$$
\begin{aligned}
Cost(T) &=^{(1)} \sum_{k=1}^{n} Cost(T_k) \\
&=^{(2)} \sum_{i \in NRec} Cost(T_i) + \sum_{i \in Rec_{ph}} Cost(T_i) + \sum_{i \in Rec_{CH}} Cost(T_i) + \sum_{i \in Trunc} Cost(T_i) \\
&=^{(3)} \sum_{i \in NRec} Cost(T_i) + \sum_{i \in Rec_{ph}} Cost(T_i) + \sum_{i \in Rec_{CH}} Cost(T_i) \\
&=^{(4)} Cost_c(T) + \sum_{i \in Rec_{ph}} Cost(T_i) + \sum_{i \in Rec_{CH}} Cost(T_i) \\
&=^{(5)} Cost_c(T) + \sum_{i \in Rec_{ph}} \Big(\sum_{\pi \in CEinst(T_i)} Cost_c((T_i)_{|\pi}) + \sum_{\pi \in maxCH(T_i)} Cost((T_i)_{|\pi}) \Big) + \\
&\quad + \sum_{i \in Rec_{CH}} Cost(T_i) \\
&=^{(6)} Cost_c(T_{|\epsilon}) + \sum_{i \in Rec_{ph}} \Big(\sum_{\pi \in CEinst(T_i)} Cost_c(T_{|i \cdot \pi}) \Big) \\
&\quad + \sum_{i \in Rec_{ph}} \Big(\sum_{\pi \in maxCH(T_i)} Cost(T_{|i \cdot \pi}) \Big) + \sum_{i \in Rec_{CH}} Cost(T_{|i}) \\
&=^{(7)} \sum_{\pi \in CEinst(T)} Cost_c(T_{|\pi}) + \sum_{\pi \in maxCH(T)} Cost(T_{|\pi})
\end{aligned}
$$

1. Definition of *Cost* (Definition 3.8)
2. We can divide $T_1, \cdots, T_n$ in four sets:
   a) The recursive evaluations that have been truncated because of a finite approximation $Trunc = \{i \mid b_i = C(\boldsymbol{x}_i : \boldsymbol{y}_i) \wedge label(T_i) = \bot\}$
   b) The recursive evaluations in the phase $Rec_{ph} = \{i \mid T_i \in [\![C[ch]]\!]_{fc}\}$
   c) The recursive evaluations outside the phase $Rec_{CH} = \{i \mid T_i \in [\![C[ch']]\!]_{fc}$ and $ch' \in CH\}$
   d) The evaluations of other CRs $Nrec = \{i \mid T_i \notin [\![C]\!]_{fc}\}$
3. As in the base case, for each $i \in Trunc$ we have $Cost(T_i) = Cost(0) = 0$
4. Definition of CE cost (Definition 6.6)
5. We apply the induction hypothesis to each T_i such that $i \in Rec_{ph}$
6. We re-arrange terms and express the sub-trees of T_i in terms of positions from T
7. We have that

$$maxCH(T) = Rec_{CH} \cup \bigcup_{i \in Rec_{ph}} \{i \cdot \pi \mid \pi \in maxCH(T_i)\}$$

and

$$CEinst(T) = \{\epsilon\} \cup \bigcup_{i \in Rec_{ph}} \{i \cdot \pi \mid \pi \in CEinst(T_i)\}$$

in which all sets are disjoint

6.10.2 Strategy Proofs Preliminaries

In Section 6.5, intermediate variable instances were numbered for a given evaluation T. In order to complete the proofs, intermediate variable instances are explicitly parameterized by an evaluation.

Let *iv* be an intermediate variable defined in a CE c, and let $T \in [\![C]\!]_{fc}$ be an evaluation with $label(T) = c$, the instance of *iv* in T is expressed as $iv(T)$. Similarly, the definition of Sum intermediate variables can be reformulated.

Definition 6.53 (Sum Intermediate Variable - Alternative Notation)**.** Let $T \in [\![C]\!]_{fc}$ be an evaluation. A Sum intermediate variable of an *iv* defined for CE c is:

$$smiv(T) = \sum_{\pi \in CEinst(T,c)} iv(T_{|\pi})$$

A Sum intermediate variable of an *iv* defined for a Chain ch' is:

$$smiv(T) = \sum_{\pi \in maxCH(T,ch')} iv(T_{|\pi})$$

This definition is equivalent to Definition 6.21. However, in Definition 6.21 the instances $CEinst(T,c)$ and their corresponding instances of intermediate variables $iv(T_{|\pi})$ had been numbered from 1 to $\#c_i = |CEinst(T,c)|$. The same applies for the instances $maxCH(T,ch')$ in Section 6.7.

Based on these definitions, the following lemmas can be established. These are extensively used in the proofs of the strategies.

Lemma 6.54. *Let $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![C[ch]]\!]_{fc}$ be an evaluation in a phase $c \in ph$ (ch starts with ph) where $T_{i_1}, \cdots, T_{i_m}$ correspond to the recursive calls. Let iv be an intermediate variable defined in B (where B can be a CE or a chain $ch' \neq ch$).*

$$smiv(T) = \begin{cases} iv(T) + \sum_{j=1}^{m} smiv(T_{i_j}) & \text{if } B = c \\ \sum_{j=1}^{m} smiv(T_{i_j}) & \text{if } B \neq c \end{cases}$$

Lemma 6.55. *Let $T \in [\![C[ch']]\!]$ and let iv be an intermediate variable defined in B (where B can be a CE $c \notin ch'$ or a chain).*

$$smiv(T) = \begin{cases} iv(T) & \text{if } B = ch' \\ 0 & \text{if } B \neq ch' \end{cases}$$

These lemmas derive directly from the definition of Sum Intermediate Variables (Definition 6.53) and the definitions of *CEinst* and *maxCH*.

6.10.3 Theorem 6.27: Inductive Sum Strategy

Let $ph = (c_1 \vee \cdots \vee c_n)^+$ be an iterative phase. We have to prove that, given a candidate $cd(\boldsymbol{x})$ such that we could classify every $c_e \in ph$ into the classes *Cnt*, *Dc* and *Ic* according to their definitions in Table 6.1. The following constraints are valid for any evaluation of the phase.

$$\sum_{c_e \in Cnt} cnt_e \bowtie iv_{cd+} - iv_{cd-} + \sum_{c_e \in Ic} smiv_{ic_e} - \sum_{c_e \in Dc} smiv_{dc_e} \qquad \begin{array}{l} iv_{cd+} \bowtie \|cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f})\| \\ iv_{cd-}! \bowtie \|-cd(\boldsymbol{x_s}) + cd(\boldsymbol{x_f})\| \end{array}$$

In this setting, we are not restricted by the format of the cost structures and we can merge the three constraints into one:

$$\sum_{c_e \in Cnt} cnt_e \bowtie cd(\boldsymbol{x_s}) - cd(\boldsymbol{x_f}) + \sum_{c_e \in Ic} smiv_{ic_e} - \sum_{c_e \in Dc} smiv_{dc_e}$$

This constraint involves intermediate variables from the cost structures of the CEs and additional intermediate variables iv_{ic_e} and iv_{dc_e} defined during the candidate classification. We have iv_{ic_e} and iv_{dc_e} for each $c_e \in Ic$ and $c_e \in Dc$ whose value is defined as $iv_{ic_e} := \|ic_e(\boldsymbol{x x'})\|$ and $iv_{dc_e} := \|dc_e(\boldsymbol{x x'})\|$ (Table 6.1). The addition of $iv_{ic_e} \bowtie \|ic(\boldsymbol{x x'})\|$ and $iv_{dc_e}! \bowtie \|dc(\boldsymbol{x x'})\|$ to $Psums^{c_e}$ follows directly from these definitions.

The constraint has to be valid for any evaluation $T \in [\![C[ph \cdot ch]]\!]_{fc}$. Consequently, we instantiate the constraint with respect to an evaluation T. Given an evaluation $T = t(c(\boldsymbol{a_s} : \boldsymbol{b_s}), [T_1, \cdots, T_n])$ with maximal evaluation tree of ch $T' = t(c'(\boldsymbol{a_f} : \boldsymbol{b_f}), \_)$. Its instantiated constraint is:

$$\sum_{c_e \in Cnt} cnt_e(T) \bowtie \quad cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T) - \sum_{c_e \in Dc} smiv_{dc_e}(T)$$

Now we prove that this constraint holds for any evaluation $T \in [\![C[ph \cdot ch']]\!]_{fc}$ by induction on $n = |CEinst(T)|$ (as defined in Section 6.10.1). That is, the number of evaluation nodes that belong to the phase.

Base Case

The case $n = 0$ corresponds to an evaluation $T \in [\![C[ch']]\!]_{fc}$. The constraint is $0 \bowtie 0$ because all *smiv* are zero (Lemma 6.55) and $T = T'$ (T is the maximal evaluation of ch in T) so $cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f}) = 0$.

Inductive Case

For the inductive case, we assume the expression holds for every evaluation of size smaller than n and prove it for size n. Let $T = t(c_i(\boldsymbol{a_s} : \boldsymbol{b_s}), [T_1, \cdots, T_n]) \in [\![C[ph \cdot ch](\boldsymbol{a_s} : \boldsymbol{b_s})]\!]_{fc}$ such that there is a recursive evaluation T_j ($T_j \in [\![C[ph \cdot ch](\boldsymbol{a'_s} : \boldsymbol{b'_s})]\!]_{fc}$ or $T_j \in [\![C[ch](\boldsymbol{a'_s} : \boldsymbol{b'_s})]\!]_{fc}$). $CEinst(T_j)$ is smaller than $CEinst(T)$ so the induction hypothesis can be applied to T_j. We distinguish cases depending on which CE is evaluated. In particular, whether c_i belongs to *Cnt*, *Dc* or *Ic*. In each case, we reduce the constraint on T to the constraint on T_j (we apply Lemma 6.54 for the Sum variables at both sides of the constraint) plus some additional summands and we prove that the additional summands maintain the inequality.

- If $c_i \in Cnt$, the left-hand side of the constraint is:

$$\sum_{c_e \in Cnt} cnt_e(T) = \sum_{c_e \in Cnt} cnt_e(T_j) + \sum_{smiv_k \in cnt_i} iv_k(T)$$

  The right-hand side of the constraint is:

$$\begin{aligned} & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T) - \sum_{c_e \in Dc} smiv_{dc_e}(T) \\ = \; & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s}) + cd(\boldsymbol{a'_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) - \sum_{c_e \in Dc} smiv_{dc_e}(T_j) \end{aligned}$$

  If we apply the induction hypothesis, we are left to prove:

$$\sum_{smiv_k \in cnt_i} iv_k(T) \quad \bowtie cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s})$$

This constraint is directly guaranteed by the classification condition of *Cnt*

$$\sum_{smiv_k \in cnt_i} iv_k \bowtie \|l'\| \in Psums^{c_i} \wedge \|l'\| \bowtie cd(\boldsymbol{x}) - cd(\boldsymbol{x'})$$

which is valid for any evaluation of a $c_i \in Cnt$ and in particular for T (with values $\boldsymbol{a_s}$ and $\boldsymbol{a'_s}$).

- If $c_i \in Dc$, the left side of the constraint does not have additional summands:

$$\sum_{c_e \in Cnt} cnt_e(T) = \sum_{c_e \in Cnt} cnt_e(T_j)$$

And the right-hand side is:

$$\begin{aligned} & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T) - \sum_{c_e \in Dc} smiv_{dc_e}(T) \\ = \; & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s}) + cd(\boldsymbol{a'_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) - \sum_{c_e \in Dc} smiv_{dc_e}(T_j) - iv_{dc_i}(T) \end{aligned}$$

We apply the induction hypothesis and we are left to prove:

$$0 \quad \bowtie cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s}) - iv_{dc_i}(T)$$

By definition of *Dc* we have $cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s})$ is positive and $iv_{dc_i}(T) = \|dc_e(\boldsymbol{a_s}\boldsymbol{a'_s})\| \bowtie cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s})$ which guarantees the condition that we want to prove.

- If $c_i \in Ic$, the left side of the constraint does not have additional summands as in the previous case. The right-hand side of the constraint can be decomposed as follows:

$$\begin{aligned} & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T) - \sum_{c_e \in Dc} smiv_{dc_e}(T) \\ = \; & cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s}) + cd(\boldsymbol{a'_s}) - cd(\boldsymbol{a_f}) + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) + iv_{ic_i}(T) - \sum_{c_e \in Dc} smiv_{dc_e}(T_j) \end{aligned}$$

We apply the induction hypothesis and we are left to prove:

$$0 \quad \bowtie cd(\boldsymbol{a_s}) - cd(\boldsymbol{a'_s}) + iv_{ic_i}(T)$$

This is directly guaranteed by the definition of *Ic* (given that $\|ic_i(\boldsymbol{a_s}\boldsymbol{a'_s})\| = iv_{ic_i}(T)$).

6.10.4 Theorem 6.31: Inductive Strategy with Resets

Let $ph = (c_1 \vee \cdots \vee c_n)^+$ be an iterative phase or $ph = (c_1 \vee \cdots \vee c_n)^\omega$ be a divergent phase. We have to prove that, given a candidate $cd(\boldsymbol{x})$ such that we could classify every $c_e \in ph$ into the classes *CntR*, *Dc*, *Ic* and *Rst* according to their definitions in Table 6.1. The following constraints are valid for any evaluation of the phase.

$$\sum_{c_e \in CntR} cntr_e \leq iv_{cd} + \sum_{c_e \in Ic} smiv_{ic_e} + \sum_{c_e \in Rst} smiv_{rst_e} \qquad iv_{cd} \leq \|cd(\boldsymbol{x_s})\|$$

Similarly to the previous proof, we merge the constraints into a single one and instantiate it for an arbitrary evaluation $T \in [\![C[ph \cdot ch']]\!]_{fc}$ or $T \in [\![C[ph]]\!]_{fc}$.

$$\sum_{c_e \in CntR} cntr_e(T) \leq \|cd(\boldsymbol{a_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T)$$

We prove that the constraint holds for all evaluations by induction on $n = |CEinst(T)|$.

Base Case

The case of $n = 0$ corresponds to an evaluation $T \in [\![C[ch']]\!]_{fc}$ or a $T = t(\bot(\boldsymbol{a} : \boldsymbol{b}), [])$ originated in a divergent phase. In both cases, the constraint is $0 \leq \|cd(\boldsymbol{a_s})\| + 0 + 0$ which holds trivially (all *smiv* are 0 according to Lemma 6.55).

Inductive Case

For the inductive case, we assume the expression holds for every evaluation of size smaller than n and prove it for size n. Let $T = t(c_i(\boldsymbol{a}_s : \boldsymbol{b}_s), [T_1, \cdots, T_n])$ be an evaluation with $c_i \in ph$ such that there is a recursive evaluation T_j with parameters $\boldsymbol{a}'_s : \boldsymbol{b}'_s$. $CEinst(T_j)$ is smaller than $CEinst(T)$ so the induction hypothesis can be applied to T_j. We distinguish cases depending on which CE is evaluated. In particular, whether c_i belongs to *CntR*, *Dc*, *Ic*, or *Rst*. In each case, we reduce the constraint on T to the constraint on T_j plus some additional summands and prove that the additional summands maintain the inequality.

- If $c_i \in CntR$, the left-hand side of the constraint is:

$$\sum_{c_e \in CntR} cntr_e(T) = \sum_{c_e \in CntR} cntr_e(T_j) + \sum_{smiv_k \in cntr_i} iv_k(T)$$

  The right-hand side of the constraint is:

$$\begin{aligned} & \|cd(\boldsymbol{a_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T) \\ = \; & \|cd(\boldsymbol{a_s})\| - \|cd(\boldsymbol{a'_s})\| + \|cd(\boldsymbol{a'_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) + \sum_{c_e \in Rst} smiv_{rst_e}(T_j) \end{aligned}$$

  If we apply the induction hypothesis, we are left to prove:

$$\sum_{smiv_k \in cntr_i} iv_k(T) \quad \leq \|cd(\boldsymbol{a_s})\| - \|cd(\boldsymbol{a'_s})\|$$

  which is directly guaranteed by the condition of *CntR*.

- If $c_i \in Dc$, the left side of the constraint does not have additional summands and the right-hand side is:

$$\begin{aligned} & \|cd(\boldsymbol{a_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T) \\ = \; & \|cd(\boldsymbol{a_s})\| - \|cd(\boldsymbol{a'_s})\| + \|cd(\boldsymbol{a'_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) + \sum_{c_e \in Rst} smiv_{rst_e}(T_j) \end{aligned}$$

  After applying the induction hypothesis, we have to prove:

$$0 \leq \|cd(\boldsymbol{a_s})\| - \|cd(\boldsymbol{a'_s})\|$$

  which is guaranteed by the condition from *Dc*: $0 \leq dc_i(\boldsymbol{x}\boldsymbol{x'}) \leq cd(\boldsymbol{x}) - cd(\boldsymbol{x'})$

- If $c_i \in Ic$, the left side of the constraint does not have additional summands as in the previous case. The right-hand side of the constraint can be decomposed as follows:

$$\begin{aligned} & \|cd(\boldsymbol{a_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T) \\ = \; & \|cd(\boldsymbol{a_s})\| - \|cd(\boldsymbol{a'_s})\| + \|cd(\boldsymbol{a'_s})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) + iv_{ic_i}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T_j) \end{aligned}$$

After applying the induction hypothesis, we have to prove (with $iv_{ic_i}(T) = \|ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s)\|$):

$$0 \leq \|cd(\boldsymbol{a}_s)\| - \|cd(\boldsymbol{a}'_s)\| + \|ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s)\|$$

From the definition of *Ic*, we know:

$$0 \leq cd(\boldsymbol{a}_s) - cd(\boldsymbol{a}'_s) + ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s) \tag{6.1}$$

We distinguish cases:

- If $cd(\boldsymbol{a}'_s)$ is negative, $\|cd(\boldsymbol{a}'_s)\| = 0$ and we have $0 \leq \|cd(\boldsymbol{a}_s)\| + \|ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s)\|$ which is trivially true (both summands are non-negative).
- If $cd(\boldsymbol{a}'_s)$ is non-negative, $\|cd(\boldsymbol{a}'_s)\| = cd(\boldsymbol{a}'_s) \leq cd(\boldsymbol{a}_s) + ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s)$ (because of Condition 6.1) which implies the condition that we have to prove: $\|cd(\boldsymbol{a}'_s)\| \leq \|cd(\boldsymbol{a}_s)\| + \|ic_i(\boldsymbol{a}_s \boldsymbol{a}'_s)\|$.

- If $c_i \in Rst$, the left side of the constraint does not have additional summands as in the previous case. The right-hand side of the constraint can be decomposed as follows:

$$\begin{aligned} & \|cd(\boldsymbol{a}_s)\| + \sum_{c_e \in Ic} smiv_{ic_e}(T) + \sum_{c_e \in Rst} smiv_{rst_e}(T) \\ = \; & \|cd(\boldsymbol{a}_s)\| - \|cd(\boldsymbol{a}'_s)\| + \|cd(\boldsymbol{a}'_s)\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_j) + \sum_{c_e \in Rst} smiv_{rst_e}(T_j) + iv_{rst_i}(T) \end{aligned}$$

After applying the induction hypothesis, we have to prove (with $iv_{rst_i}(T) = \|rst_i(\boldsymbol{a}_s)\|$):

$$0 \leq \|cd(\boldsymbol{a}_s)\| - \|cd(\boldsymbol{a}'_s)\| + \|rst_i(\boldsymbol{a}_s)\|$$

By the definition of *Rst* we have $\|rst_i(\boldsymbol{a}_s)\| \geq cd(\boldsymbol{a}'_s)$ which is sufficient to prove that $\|cd(\boldsymbol{a}_s)\| - \|cd(\boldsymbol{a}'_s)\| + \|rst_i(\boldsymbol{a}_s)\|$ is non-negative.

6.10.5 Theorems 6.33 and 6.34: Max-Min Strategy

These theorems are used to generate constraints during the computation of a cost structure for a phase *ph* (Section 6.5.2) or a chain $ph \cdot CH$ (Section 6.7) and they are valid also for divergent phases.

Given a constraint $iv \leq \|l\| \in Pms^B$ (where $B \in ph$ or $B \in CH$), if we manage to classify all the $c_e \in ph$ into *Dc*, *Ic*, and *Rst*, we generate:

$$\lceil iv \rceil \leq iv_{max} + \sum_{c_e \in Ic} smiv_{ic_e}, \quad iv_{max} \leq \max_{c_e \in Rst}(\lceil iv \rceil_{rst_e}, iv_{cd}), \quad iv_{cd} \leq \|cd(\boldsymbol{x}_s)\|$$

Similarly to the previous proofs, we merge the constraints into a single one and instantiate it for an arbitrary evaluation $T_1 \in [\![C[ch]]\!]_{fc}$ such that *ch* starts with phase *ph* (the chain *ch* can be either $ch = ph$, $ch = ph \cdot ch'$ or $ch = ph \cdot CH$).

$$\lceil iv \rceil(T_1) \leq \max_{c_e \in Rst}(\lceil iv \rceil_{rst_e}(T_1), \|cd(\boldsymbol{a}_s)\|) + \sum_{c_e \in Ic} smiv_{ic_e}(T_1) \tag{6.2}$$

Let $T_j \preceq T_1$ such that $T_j = (T_1)_{|\pi}$ for a $\pi \in CEinst(T_1, B)$ or $\pi \in maxCH(T_1, B)$ (depending on whether B is a CE or a chain) and parameters $param(T_i) = (\boldsymbol{a}_j : \boldsymbol{b}_j)$ (in the case of T_1, $param(T_1) = (\boldsymbol{a}_1 : \boldsymbol{b}_1) = (\boldsymbol{a}_s : \boldsymbol{b}_s)$). In order to prove that the constraint 6.2 holds, it is enough to prove that the following constraint

$$iv(T_j) \leq \max_{c_e \in Rst}(\lceil iv \rceil_{rst_e}(T_1), \|cd(\boldsymbol{a_1})\|) + \sum_{c_e \in Ic} smiv_{ic_e}(T_1) \tag{6.3}$$

holds for any T_j. This is because if all instances $iv(T_j)$ for all $T_j \preceq T_1$ are smaller or equal than an amount, $\lceil iv \rceil(T_1)$ (which is the biggest instance) is also smaller or equal than that amount.

Given a T_j and its intermediate variable instance $iv(T_j)$, we prove that either:

1. There is a $c_k \in Rst$ such that: $iv(T_j) \leq \lceil iv \rceil_{rst_k}(T_1) + \sum_{c_e \in Ic} smiv_{ic_e}(T_1)$;

2. or $iv(T_j) \leq \|cd(\boldsymbol{a_1})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_1)$.

Both cases imply the constraint 6.3.

First, given the definition of the candidate, we have that $iv(T_j) \leq \|l(\boldsymbol{a_j a'_j})\| \leq \|cd(\boldsymbol{a_j})\|$. Then, we consider the evaluations between T_1 and T_j, that is, the path of evaluations $T_1, T_2, \cdots, T_{j-1}$ where each T_{i+1} is a direct successor of T_i and T_i has $param(T_i) = (\boldsymbol{a}_i : \boldsymbol{b}_i)$ and $label(T_i) \in ph$. In this path, we take the last T_r in the path such that the candidate is reset, that is $label(T_r) \in Rst$. The path from T_r to T_j contains evaluations of T_i for $r < i < j$ such that $label(T_i)$ can only belong to Dc or Ic.

- For each T_i such that $label(T_i) \in Dc$, we have $\|cd(\boldsymbol{a_{i+1}})\| \leq \|cd(\boldsymbol{a_i})\|$.

- For each T_i such that $label(T_i) = c_e \in Ic$, we have $\|cd(\boldsymbol{a_{i+1}})\| \leq \|cd(\boldsymbol{a_i})\| + iv_{ic_e}(T_i)$. This is because $cd(\boldsymbol{a_{i+1}}) \leq cd(\boldsymbol{a_i}) + ic_e(\boldsymbol{a_i})$, from the classification condition, implies $\|cd(\boldsymbol{a_{i+1}})\| \leq \|cd(\boldsymbol{a_i})\| + \|ic_e(\boldsymbol{a_i})\| = \|cd(\boldsymbol{a_i})\| + iv_{ic_e}(T_i)$ (see the case $c_i \in Ic$ in the previous proof).

Therefore, we have:

$$iv(T_j) \leq \|cd(\boldsymbol{a_{r+1}})\| + \sum_{r<i<j \ \wedge \ label(T_i)=c_e \in Ic} iv_{ic_e}(T_i) \leq \|cd(\boldsymbol{a_{r+1}})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_1)$$

Given that $label(T_r) = c_k \in Rst$, we have that the classification condition for Rst holds $\|cd(\boldsymbol{a_{r+1}})\| \leq \|rst_e(\boldsymbol{a_r})\| = iv_{rst_e}(T_r)$ which by definition is $iv_{rst_k}(T_r) \leq \lceil iv \rceil_{rst_k}(T_1)$. Therefore, we conclude with the first case:

$$iv(T_j) \leq \lceil iv \rceil_{rst_k}(T_1) + \sum_{c_e \in Ic} smiv_{ic_e}(T_1)$$

If there is no evaluation T_r in the range $1 \leq r < j$ such that $label(T_r) \in Rst$, we carry up the transformation to the root of the evaluation and obtain the second case:

$$iv(T_j) \leq \|cd(\boldsymbol{a_1})\| + \sum_{1<i<j \ \wedge \ label(T_i)=c_e \in Ic} iv_{ic_e}(T_k) \leq \|cd(\boldsymbol{a_1})\| + \sum_{c_e \in Ic} smiv_{ic_e}(T_1)$$

The proof for Theorem 6.34 for constraints $\lfloor iv \rfloor \geq \|l\| \in Pms^B$ is analogous.

6.10.6 Theorem 6.37: Triangular Sum Strategy

Let $ph := (c_1 \vee \cdots \vee c_n)^+$ be an iterative phase. We have to prove that, given a candidate $cd(\boldsymbol{x})$ such that we could classify every $c_e \in ph$ into the classes *CntT* and *Nop* according to their definitions in Table 6.1. The following constraints are valid:

$$\sum_{c_e \in CntT} cntt_e \bowtie \|cd(\boldsymbol{x_s})\| \cdot iv_{its} + \tfrac{q}{2} iv_{its}^2 - \tfrac{q}{2} iv_{its} \quad , \quad iv_{its} = \sum_{c_e \in CntT} smiv_{it_e}$$

These constraints are a merged version of the ones stated in Section 6.5.2. Here $q := \max_{c_e \in CntT}(q_e)$ if $\bowtie$ is $\leq$ or $q := \min_{c_e \in CntT}(q_e)$ otherwise. The variable iv_{its} represents the number of evaluations of CE in the phase evaluation such that $c_e \in CntT$.

Let us consider an arbitrary evaluation of the phase $T_1 \in [\![C[ch]]\!]_{fc}$ for a chain $ch = ph \cdot ch'$. Let $T_f \preceq T_1$ such that $T_f \in [\![C[ch']]\!]_{fc}$ is the maximal evaluation of ch' in T_1. We denote with T_i the sub-evaluations of T_1 in the phase for $1 \leq n < f$ (that is, $T_i = (T_1)_{|\pi}$ with $\pi \in CEinst(T_1)$) with variables $param(T_i) = (\boldsymbol{a_i} : \boldsymbol{b_i})$. We introduce the following auxiliary notion.

Definition 6.56 (Partial count)**.** $iv_{its}[1..i]$ is the partial count of *CntT* in the evaluations $[1..i]$. It represents the number of CE evaluations of $c_e \in CntT$ in the range $1 \leq j < i$. Note that we have $iv_{its} = iv_{its}[1..f]$.

Then, we state the following lemma:

Lemma 6.57. *For all* $1 \leq i < f$ *we have*

$$cd(\boldsymbol{a_i}) \bowtie cd(\boldsymbol{a_1}) + q \cdot iv_{its}[1..i]$$

Proof. We prove it by induction over i.

- Base case: For $i = 1$, the interval in $iv_{its}[1..1]$ is empty, $iv_{its}[1..1] = 0$ and $cd(\boldsymbol{a_1}) \bowtie cd(\boldsymbol{a_1}) + 0$.
- Inductive case: We assume $cd(\boldsymbol{a_i}) \bowtie cd(\boldsymbol{a_1}) + q \cdot iv_{its}[1..i]$ and prove it for $i+1$. We distinguish two cases:
  - If $label(T_i) \in CntT$, we have:

$$\begin{aligned} cd(\boldsymbol{a_{i+1}}) &\bowtie cd(\boldsymbol{a_i}) + q \\ &\bowtie^{(IH)} cd(\boldsymbol{a_1}) + q \cdot iv_{its}[1..i] + q \\ &= cd(\boldsymbol{a_1}) + q \cdot (iv_{its}[1..i] + 1) = cd(\boldsymbol{a_1}) + q \cdot (iv_{its}[1..(i+1)]) \end{aligned}$$

  - If $label(T_i) \in Nop$, we have:

$$\begin{aligned} cd(\boldsymbol{a_{i+1}}) &= cd(\boldsymbol{a_i}) =^{(IH)} cd(\boldsymbol{a_1}) + q \cdot iv_{its}[1..i] \\ &= cd(\boldsymbol{a_1}) + q \cdot iv_{its}[1..i+1] \end{aligned}$$

□

According to the definition of *CntT* (Table 6.1), we have:

$$\sum_{c_e \in CntT} cntt_e \bowtie \sum_{1 \leq j < f \wedge label(T_j) \in CntT} cd(\boldsymbol{a_j})$$

We prove that the right side of the constraint that we generate is a valid approximation of this constraint:

$$\begin{aligned}
\sum_{1\le j<f\wedge label(T_j)\in CntT} cd(\boldsymbol{a_j}) &\bowtie^{(1)} \sum_{1\le j<f\wedge label(T_j)\in CntT} \big(cd(\boldsymbol{a_1}) + q\cdot iv_{its}[1..j]\big)\\
&=^{(2)} iv_{its}[1..f\,]\cdot cd(\boldsymbol{a_1}) + \sum_{\substack{1\le j<f\wedge label(T_j)\in CntT\\ iv_{its}[1..f\,]-1}} (q\cdot iv_{its}[1..j])\\
&=^{(3)} iv_{its}[1..f\,]\cdot cd(\boldsymbol{a_1}) + q\sum_{j=0}^{} j\\
&=^{(4)} iv_{its}\cdot cd(\boldsymbol{a_1}) + \sum_{j=0}^{iv_{its}-1} j\\
&=^{(5)} cd(\boldsymbol{a_1})\cdot(iv_{its}) + \tfrac{q}{2}\cdot(iv_{its}^2 - iv_{its})
\end{aligned}$$

1. Because of Lemma 6.57
2. Definition of $iv_{its}[1..n]$ and distributivity
3. Express sum as indexed sum
4. Definition of $iv_{its}[1..n]$: $iv_{its} = iv_{its}[1..f\,]$
5. Solve arithmetic sequence

6.10.7 Theorem 6.43: Inductive Sum Strategy for Multiple Chains

Let $ch = ph\cdot CH$ be a multiple chain. We have to prove that, given a candidate $cd(\boldsymbol{xy})$ such that we could classify every $c_e \in ph$ and every $ch' \in CH$ into the classes *Cnt*, *Dc* and *Ic* according to their definitions in Table 6.3. The following constraints are valid for any evaluation of the chain.

$$\sum_{B\in Cnt} cnt_B \bowtie iv_{cd+} - iv_{cd-} + \sum_{B\in Ic} smiv_{ic_B} - \sum_{B\in Dc} smiv_{dc_B} \qquad \begin{array}{l} iv_{cd+} \bowtie \|cd(\boldsymbol{x_s y_s})\| \\ iv_{cd-}! \bowtie \|{-cd}(\boldsymbol{x_s y_s})\| \end{array}$$

As in previous proofs, we merge the constraints and instantiate the result for an evaluation T. Given an evaluation $T = t(c(\boldsymbol{a:b}), [T_1, \cdots, T_n])$, we define its corresponding constraint:

$$\sum_{B\in Cnt} cnt_B(T) \bowtie \quad cd(\boldsymbol{ab}) + \sum_{B\in Ic} smiv_{ic_B}(T) - \sum_{B\in Dc} smiv_{dc_B}(T)$$

We prove that this constraint holds for any evaluation $T \in [\![C[ph\cdot CH]]\!]_{fc}$ or $T \in [\![C[ch']]\!]_{fc}$ with $ch' \in CH$ by induction on $n = |CEinst(T)|$, that is, the number of evaluation nodes that belong to the phase.

Base Case

We consider the case $n = 0$, that is $T \in [\![C[ch']]\!]_{fc}$ for a chain $ch' \in CH$. We distinguish cases according to whether ch' has been classified in *Cnt*, *Dc* or *Ic*:

- If $ch' \in Cnt$, we have

$$\begin{aligned}
\sum_{B\in Cnt} cnt_B(T) &\bowtie cd(\boldsymbol{ab}) + \sum_{B\in Ic} smiv_{ic_B}(T) - \sum_{B\in Dc} smiv_{dc_B}(T)\\
\sum_{smiv_k\in cnt_{ch'}} iv_k(T) &\bowtie cd(\boldsymbol{ab})
\end{aligned}$$

  This is guaranteed by the classification condition of *Cnt*: $\sum \boldsymbol{iv'} \bowtie \|l'\| \bowtie cd(\boldsymbol{xy})$ where $cnt_{ch'}$ is defined as $\sum \boldsymbol{smiv'}$.

- If $ch' \in Dc$, we have

$$\begin{aligned} \sum_{B \in Cnt} cnt_B(T) &\bowtie cd(\boldsymbol{ab}) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\ 0 &\bowtie cd(\boldsymbol{ab}) - iv_{dc_{ch'}} \\ iv_{dc_{ch'}} &\bowtie cd(\boldsymbol{ab}) \end{aligned}$$

  This is guaranteed by the classification condition of Dc.

- If $ch' \in Ic$, we have

$$\begin{aligned} \sum_{B \in Cnt} cnt_B(T) &\bowtie cd(\boldsymbol{ab}) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\ 0 &\bowtie cd(\boldsymbol{ab}) + iv_{ic_{ch'}} \\ -cd(\boldsymbol{ab}) &\bowtie iv_{ic_{ch'}} \end{aligned}$$

  This is guaranteed by the classification condition of Ic.

Inductive Case

For the inductive case, we assume the expression holds for every evaluation of size smaller than n and prove it for size n. Let $T = t(c_i(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![C[ph \cdot CH]]\!]_{fc}$. We assume without loss of generality that $T_1, \cdots, T_m$ correspond to the recursive calls, that is, for each T_j $1 \leq j \leq m$ either $T_j \in [\![C[ph \cdot CH](\boldsymbol{a}_j : \boldsymbol{b}_j)]\!]_{fc}$ or $T_j \in [\![C[ch'](\boldsymbol{a}_j : \boldsymbol{b}_j)]\!]_{fc}$ for a $ch' \in CH$. $CEinst(T_j)$ is smaller than $CEinst(T)$ so the induction hypothesis can be applied to every T_j. We distinguish cases depending on which CE is evaluated. In particular, whether c belongs to Cnt, Dc or Ic. In each case, we reduce the constraint on T to the constraints on T_j plus some additional summands and prove that the additional summands maintain the inequality.

- If $c_i \in Cnt$, the left-hand side of the constraint is:

$$\sum_{B \in Cnt} cnt_B(T) = \sum_{smiv_k \in cnt_{c_i}} iv_k(T) + \sum_{j=1}^{m} \left(\sum_{B \in Cnt} cnt_B(T_j) \right)$$

  The right-hand side of the constraint is:

$$\begin{aligned} & cd(\boldsymbol{ab}) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\ = \; & cd(\boldsymbol{ab}) + \left(\sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) \right) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\ = \; & cd(\boldsymbol{ab}) + \left(\sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) \right) + \sum_{B \in Ic} \left(\sum_{j=1}^{m} smiv_{ic_B}(T_j) \right) - \sum_{B \in Dc} \left(\sum_{j=1}^{m} smiv_{dc_B}(T_j) \right) \\ = \; & cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) + \sum_{j=1}^{m} \left(cd(\boldsymbol{a_j b_j}) + \sum_{B \in Ic} smiv_{ic_B}(T_j) - \sum_{B \in Dc} smiv_{dc_B}(T_j) \right) \end{aligned}$$

  If we apply the induction hypothesis for all T_j, we are left to prove:

$$\sum_{smiv_k \in cnt_{c_i}} iv_k(T) \bowtie cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j})$$

This constraint is directly guaranteed by the classification condition of *Cnt*.

- If $c_i \in Dc$, the left side can be completely reduced to the cases of the evaluations T_j:

$$\sum_{B \in Cnt} cnt_B(T) = \sum_{j=1}^{m} \Big(\sum_{B \in Cnt} cnt_B(T_j) \Big)$$

And the right-hand side is:

$$\begin{aligned}
& cd(\boldsymbol{ab}) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\
= \;& cd(\boldsymbol{ab}) + \left(\sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) \right) + \sum_{B \in Ic} \Big(\sum_{j=1}^{m} smiv_{ic_B}(T_j) \Big) \\
& - \sum_{B \in Dc} \Big(\sum_{j=1}^{m} smiv_{dc_B}(T_j) \Big) - iv_{dc_{c_i}}(T) \\
= \;& cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - iv_{dc_{c_i}}(T) + \sum_{j=1}^{m} \Big(cd(\boldsymbol{a_j b_j}) + \sum_{B \in Ic} smiv_{ic_B}(T_j) - \sum_{B \in Dc} smiv_{dc_B}(T_j) \Big)
\end{aligned}$$

If we apply the induction hypothesis for all T_j, we are left to prove:

$$0 \quad \bowtie cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - iv_{dc_{c_i}}(T)$$

This constraint is directly guaranteed by the classification condition of *Dc*.

- If $c_i \in Ic$, the left side can be completely reduced to the cases of the evaluations T_j as in the previous case. The right-hand side of the constraint can be decomposed as follows:

$$\begin{aligned}
& cd(\boldsymbol{ab}) + \sum_{B \in Ic} smiv_{ic_B}(T) - \sum_{B \in Dc} smiv_{dc_B}(T) \\
= \;& cd(\boldsymbol{ab}) + \left(\sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) \right) + \sum_{B \in Ic} \Big(\sum_{j=1}^{m} smiv_{ic_B}(T_j) \Big) + iv_{ic_{c_i}}(T) \\
& - \sum_{B \in Dc} \Big(\sum_{j=1}^{m} smiv_{dc_B}(T_j) \Big) \\
= \;& cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) + iv_{ic_{c_i}}(T) + \sum_{j=1}^{m} \left(cd(\boldsymbol{a_j b_j}) + \sum_{B \in Ic} smiv_{ic_B}(T_j) - \sum_{B \in Dc} smiv_{dc_B}(T_j) \right)
\end{aligned}$$

If we apply the induction hypothesis for all T_j, we are left to prove:

$$0 \quad \bowtie cd(\boldsymbol{ab}) - \sum_{j=1}^{m} cd(\boldsymbol{a_j b_j}) + iv_{ic_{c_i}}(T)$$

This constraint is directly guaranteed by the classification condition of *Ic*.

6.10.8 Theorem 6.44: Inductive Strategy with Resets for Multiple Recursion

Let $ch = ph \cdot CH$ be a multiple chain where ph might be divergent, we have to prove that, given a candidate $cd(\boldsymbol{x}\,\boldsymbol{y})$ such that we could classify every $c_e \in ph$ and every $ch' \in CH$ into the classes *CntR*, *DcR*, and *IcR* according to their definitions in Table 6.4. The following constraints are valid for any evaluation of the chain.

$$\sum_{B \in CntR} cntr_B \leq iv_{cd} + \sum_{B \in IcR} smiv_{icr_B} \qquad iv_{cd} \leq \|cd(\boldsymbol{x_s})\|$$

As in previous proofs, we merge the constraints and instantiate the result for an evaluation T. Given an evaluation $T = t(c(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n])$, we define its corresponding constraint:

$$\sum_{B \in CntR} cntr_B(T) \leq \quad \|cd(\boldsymbol{a})\| + \sum_{B \in IcR} smiv_{icr_B}(T)$$

We prove that this constraint holds for any evaluation $T \in [\![C[ph \cdot CH]]\!]_{fc}$, $T \in [\![C[ch']]\!]_{fc}$ with $ch' \in CH$, or $T = t(\bot(\boldsymbol{a} : \boldsymbol{b}), [])$ by induction on $n = |CEinst(T)|$, that is, the number of evaluation nodes that belong to the phase.

Base Case

Consider the case $n = 0$, that is $T \in [\![C[ch']]\!]_{fc}$ for a chain $ch' \in CH$ or $T = t(\bot(\boldsymbol{a} : \boldsymbol{b}), [])$. We distinguish cases:

- If $T = t(\bot(\boldsymbol{a} : \boldsymbol{b}), [])$, we have

$$\begin{aligned} \sum_{B \in CntR} cntr_B(T) \leq \quad & \|cd(\boldsymbol{a})\| + \sum_{B \in IcR} smiv_{icr_B}(T) \\ 0 \leq \quad & \|cd(\boldsymbol{a})\| \end{aligned}$$

  which is trivially satisfied.

- If $ch' \in CntR$, we have

$$\begin{aligned} \sum_{B \in CntR} cntr_B(T) \quad & \leq \|cd(\boldsymbol{a})\| + \sum_{B \in IcR} smiv_{icr_B}(T) \\ \sum_{smiv_k \in cntr_{ch'}} iv_k(T) \quad & \leq \|cd(\boldsymbol{a})\| \end{aligned}$$

  This is guaranteed by the classification condition of *CntR*: $\sum \boldsymbol{iv'} \bowtie \|l'\| \leq \|cd(\boldsymbol{x})\|$ where $cntr_{ch'}$ is defined as $\sum \boldsymbol{smiv'}$.

- If $ch' \in DcR$ or $ch' \in IcR$, we obtain respectively

$$\begin{aligned} 0 \leq \quad & \|cd(\boldsymbol{a})\| && \text{for } ch' \in DcR \\ 0 \leq \quad & \|cd(\boldsymbol{a})\| + iv_{ic_{ch'}} && \text{for } ch' \in IcR \end{aligned}$$

  which are both trivially satisfied.

Inductive Case

For the inductive case, we assume the expression holds for every evaluation of size smaller than n and prove it for size n. Let $T = t(c_i(\boldsymbol{a} : \boldsymbol{b}), [T_1, \cdots, T_n]) \in [\![C[ph \cdot CH]]\!]_{fc}$. We assume without loss of generality that $T_1, \cdots, T_m$ correspond to the recursive calls, that is, for each T_j $1 \leq j \leq m$ either $T_j \in [\![C[ph \cdot CH](\boldsymbol{a}_j : \boldsymbol{b}_j)]\!]_{fc}$, or $T_j \in [\![C[ch'](\boldsymbol{a}_j : \boldsymbol{b}_j)]\!]_{fc}$ for a $ch' \in CH$, or $T_j = t(\bot(\boldsymbol{a}_j : \boldsymbol{b}_j), [])$. $CEinst(T_j)$ is smaller than $CEinst(T)$ so the induction hypothesis can be applied to every T_j. We distinguish cases

depending on whether c belongs to *CntR*, *DcR*, or *IcR*. In each case, we reduce the constraint on T to the constraints on T_j plus some additional summands and prove that the additional summands maintain the inequality. Many of these reductions are very similar to the ones in proof of Section 6.10.3 so they are presented summarized.

- If $c_i \in CntR$, the left-hand side of the constraint is:

$$\sum_{B \in CntR} cntr_B(T) = \sum_{smiv_k \in cntr_{c_i}} iv_k(T) + \sum_{j=1}^{m} \left(\sum_{B \in CntR} cntr_B(T_j) \right)$$

  The right-hand side of the constraint is:

$$\|cd(\boldsymbol{a})\| + \sum_{B \in IcR} smiv_{icr_B}(T) = \|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\| + \sum_{j=1}^{m} \left(\|cd(\boldsymbol{a_j})\| + \sum_{B \in IcR} smiv_{icr_B}(T_j) \right)$$

  If we apply the induction hypothesis for all T_j, we are left to prove:

$$\sum_{smiv_k \in cntr_{c_i}} iv_k(T) \quad \leq \|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\|$$

  This constraint is directly guaranteed by the classification condition of *CntR*.

- If $c_i \in DcR$ or $c_i \in IcR$ the left side can be completely reduced to the cases of the evaluations T_j:

$$\sum_{B \in CntR} cntr_B(T) = \sum_{j=1}^{m} (\sum_{B \in CntR} cntr_B(T_j))$$

  And the right-hand side can be reduced respectively to:

$$\begin{aligned} &\|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\| + \sum_{j=1}^{m} \left(\|cd(\boldsymbol{a_j})\| + \sum_{B \in IcR} smiv_{icr_B}(T_j) \right) && \text{for } c_i \in DcR \\ &\|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\| + iv_{icr_{c_i}}(T) + \sum_{j=1}^{m} \left(\|cd(\boldsymbol{a_j})\| + \sum_{B \in IcR} smiv_{icr_B}(T_j) \right) && \text{for } c_i \in IcR \end{aligned}$$

  If we apply the induction hypothesis for all T_j, we obtain:

$$\begin{aligned} &0 \leq \|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\| && \text{for } c_i \in DcR \\ &0 \leq \|cd(\boldsymbol{a})\| - \sum_{j=1}^{m} \|cd(\boldsymbol{a_j})\| + iv_{icr_{c_i}}(T) && \text{for } c_i \in IcR \end{aligned}$$

  which are directly guaranteed by the classification conditions of *DcR* and *IcR*.

7 Evaluation

An analysis to obtain upper and lower bounds for cost relation systems has been presented. The analysis is sound and can obtain precise bounds for programs with complex features, in particular, it overcomes many of the limitations of previous approaches. Back in the introduction, it was argued that despite any additional power, there will always be programs for which any technique fails because cost analysis is an undecidable problem. Consequently, the main objective of this research has been to extend the kinds of programs that can be automatically analyzed in practice. Whether this objective has been attained or not is an empirical question. In order to answer such question, the analysis has been implemented into a prototype tool, tested against various benchmarks, and compared to existing tools. The complete results of the experiments can be found at CoFloCo's website[1].

7.1 Implementation

The analysis has been implemented into a tool called CoFloCo. CoFloCo is written in Prolog, it is open source and publicly available[2]. The tool uses the Parma Polyhedra Library (PPL) [BHZ08] to reason about linear constraint sets.

CoFloCo can be tried online through a web interface. In the web interface, it can be used alone to solve cost relation systems or combined with Llvm2kittel (https://github.com/s-falke/llvm2kittel) and a translation script to analyze single functions written in C. CoFloCo has also been integrated to be used as a backend in SACO [AAF+14] for the analysis of ABS programs. The COSTA tool [AAG+12] also generates cost relations from Java bytecode but no integration has taken place so far. Finally, CoFloCo includes scripts to generate cost relations from KoAT's integer transition system format [BEF+16] and from a subset of first order pure Lisp programs.

More recently, CoFloCo has been integrated in AProVE [GAB+17] to obtain upper bounds of term rewrite systems [NFB+17], Java programs [FG17], and of C programs with bitvector arithmetics [HGFS17].

The implementation follows closely the analysis presented here with a few exceptions. At the moment, the implementation only supports cost equations with a single constraint in the input format. This is not a problem because that is the format generated by the existing frontends as long as the input-output size analysis is not performed (see Section 1.2.2). In addition to that, the implementation does not make an explicit case distinction for non-terminating chains in the case of non-tail recursion or multiple recursion (as seen in Chapter 5, the case of tail recursive definitions is simpler). That is, if there is a CR with non-tail recursion or multiple recursion and non-termination cannot be completely discarded, the analysis simply fails.

Note that these limitations of the current implementation do not affect the validity of the results nor the precision of the experiments in practice. Imperative programs are transformed into tail-recursive cost relations which are fully supported. For the analysis of functional programs and term-rewrite systems, only the number of steps (time) cost model is considered and for such a cost model, non-terminating programs cannot have a finite cost upper bound.

Appendix A contains a description of the parameters of CoFloCo and some additional details of its implementation.

[1] http://aeflores.github.io/CoFloCo/experimentsPhD/
[2] https://github.com/aeflores/CoFloCo

7.2 Experiments on Imperative programs

The first part of the experiments compares CoFloCo to other state-of-the-art tools for analyzing single-function integer programs written in C. In this setting, CoFloCo is compared with the highest number of tools because many tools can read C code directly or there are automatic translation tools that generate the required input format. On the other hand, these experiments do not contain programs with non-tail recursion, multiple recursion, or data structures. The first comparison is composed by challenging examples from the literature. The second and the third are replications of the experiments performed in [SZV17] in which a larger benchmark of real world code and a smaller set of challenging loop patterns were analyzed. All these experiments were run on a Linux system with an Intel i7-3667U 2.00GHz processor with 8GB memory and with a timeout of 60 seconds per example.

7.2.1 Examples from the Literature

A set of examples extracted from the literature have been analyzed to infer upper and lower bounds. For upper bounds, the benchmark contains a total of 122 challenging programs written in C extracted from the evaluations of [ADFG10, CHS15] and from the papers [SZV17, SZV14, GMC09, Flo16, GZ10, GJK09, ZGSV11]. CoFloCo is compared to the last version of Loopus[3] [SZV17], KoAT[4] [BEF+16], C4B [CHS15], PUBS-A (the implementation of the analysis presented in [ABAG13] which is the most powerful) and Rank[5] [ADFG10]. Each tool has a different input format and the target programs had to be translated accordingly:

C4B C4B analyzes the source code directly.

Loopus Loopus analyzes the LLVM intermediate representation (LLVM-IR). This representation was generated with the Clang compiler.

KoAT The tool Llvm2kittel [FKS11] was used to transform the LLVM-IR programs into integer trasition systems (ITS) for KoAT. For the translation, Llvm2kittel was executed with the parameters `-uniform-complexity-tuples` and `-division-constraint exact`.

CoFloCo Llvm2kittel was also used to generate ITS which were later translated to cost relations systems (CRS) by a dedicated script[6]. In this case, Llvm2kittel was executed with the parameters `-complexity-tuples -division-constraint exact`[7]. The translation script from ITS to cost relations was executed with the option `loop_cost_model` which assigns a cost model that counts the number of loop iterations. Note that Loopus and C4B also count the number of loop iterations but KoAT counts the number of transition steps in the ITS representation. This affects the multiplicative factors in the resulting bounds but not the asymptotic complexity. In this evaluation, CoFloCo was executed with the following options: `-v 3 -solve_fast -compute_lbs no -stats` (see Appendix A for a description of each option).

PUBS-A Generating a valid input for PUBS-A was more challenging. PUBS-A also receives CRS as input, but these CRS have to be expressed only in terms of the input variables. The CRS for CoFloCo can be easily transformed to exclude the output variables. However, PUBS-A expects CRS that have already been enriched with input-output relations (see Section 1.2.2). Unfortunately, the current implementation of PUBS-A only supports CEs with a single constraint set and adding input-output

[3] Binary taken from `http://forsyte.at/software/loopus/` on March 15, 2017.
[4] Commit $b8618f7$ on January 15, 2017.
[5] Downloaded from `http://compsys-tools.ens-lyon.fr/rank/` on March 15, 2017.
[6] The script to translate ITS to CRS can be found in the CoFloCo repository.
[7] The different Llvm2kittel options used for KoAT and CoFloCo come from the fact that KoAT requires ITS where all transitions have the same number of variables whereas that is not necessary for the translation to CRS.

Table 7.1.: Upper bounds of examples from the literature: The number of examples with a given complexity, failed or timeout (T/O). The number of examples in which CoFloCo reports a better or worse bound than each tool and the average (Avg.) and median (Md.) analysis times per example in seconds.

	# Examples (Total 122)							CoFloCo is		Time(s)	
Tool	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$> \mathcal{O}(n^3)$	Fail	T/O	Better	Worse	Avg.	Md.
CoFloCo	3	62	34	2	1	19	1	-	-	1.44	0.48
PUBS-A	3	41	29	3	1	43	2	35	0	0.92	0.71
Loopus	2	57	28	0	2	33	0	19	4	0.05	0.02
KoAT	3	46	41	8	3	16	5	27	5	6.11	1.34
C4B	1	42	-	-	-	79	0	60	1	1.19	0.07
Rank	1	53	25	1	1	41	0	28	5	0.35	0.08

relations to such a constraint set is unsound if the CRS is non-terminating. On the other hand, performing the analysis without adding any input-output relations yields very weak results.

In order to obtain the best results for PUBS-A, while maintaining soundness, the following approach was taken. First, CoFloCo was executed with the option `-only-termination`. That option tries to prove termination of the cost relation system by performing the refinement. If all the non-terminating chains are discarded during the refinement, the CRS is terminating and input-output relations can be safely added to the CRS. The input-output relations were computed using the implementation of SACO. If CoFloCo failed to prove termination, no input-output relations were added to the CRS. The analysis time of PUBS-A includes the input-output relation generation (if it takes place) and the running time of PUBS-A but it does not include the auxiliary call to CoFloCo.

Rank Two options were considered for generating the input files for Rank. The first one is using a script available in KoAT's repository that translates ITS to Rank's representation. Unfortunately, Rank failed to analyze most of the examples generated this way, including most of the examples that come precisely from Rank's experimental evaluation[8]. The second option (which is the one adopted here) consists on generating Rank's input files using C2fsm and Aspic [FG10]. However, C2fsm supports only a limited subset of C which means that some examples had to be adapted and Rank could not be included in the remaining evaluations.

Table 7.1 contains a summary of the results of the analysis. It contains how many examples were reported in each complexity category, failed or timed out. Note that C4B can only compute linear bounds so its columns for non-linear bounds are empty. The right-hand side of the table contains the number of examples in which each CoFloCo computed a better or worse asymptotic bound than each of the other tools. For instance, CoFloCo computed a better bound than KoAT in 27 examples and Loopus computed a better bound than CoFloCo in 4 examples. Compared to all the other tools, CoFloCo was better in more examples than it was worse. Finally, the average and median times in seconds needed per program are reported. These times do not include the translation times between formats.

The second evaluation compared CoFloCo to PUBS-M (the implementation of the analysis presented in [AGM13]) for computing lower bounds. None of the other tools can compute lower bounds. The analyzed examples include the 122 examples from the first evaluation plus the examples of PUBS's evaluation and the examples of the evaluation in [ABAG13] making a total of 192. These additional

[8] This can be seen for example in the results of the experimental evaluation of [BEF+16]. Its cause is not clear, but it might be related to the way loops are encoded.

Table 7.2.: Lower bounds of examples from the literature: The number of examples with a given complexity, failed or timeout (T/O) and the average (Avg.) and median (Md.) times per example in seconds.

	# Examples (Total 192)								Time(s)	
Tool	$\Omega(1)$	$\Omega(\log(n))$	$\Omega(n)$	$\Omega(n\log(n))$	$\Omega(n^2)$	$\geq\Omega(n^3)$	T/O	Fail	Avg.	Md.
CoFloCo	64	0	97	0	26	4	1	0	1.89	1.10
PUBS-M	88	2	35	1	17	5	0	44	2.33	1.87

Table 7.3.: Replication of real world experimental evaluation of [SZV17]: The number of examples with a given complexity, failed or timeout (T/O). The number of examples in which CoFloCo reports a better or worse bound than each tool and the average (Avg.) and median (Md.) analysis times per example in seconds.

	# Examples (Total 1650)							CoFloCo is		Time(s)	
Tool	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)>\mathcal{O}(n^3)$		Fail	T/O	Better	Worse	Avg.	Md.
CoFloCo	211	144	39	0	0	1242	14	-	-	1.70	0.66
PUBS-A	195	138	36	0	0	1218	63	25	0	3.45	0.38
Loopus	205	486	97	12	2	839	9	18	426	0.75	0.04
KoAT	204	135	41	0	1	1144	125	21	3	7.72	0.69
Loopus *	194	138	40	0	0	1274	4	27	5	0.68	0.05

examples were not included in the previous evaluation because they are only available as cost relation systems. These CRS were already expressed only in terms of the input variables so they could be directly analyzed by CoFloCo and PUBS-M. The input files generated from C programs were transformed in the same way as in the previous evaluation. The results can be found in Table 7.2. In this case, the table includes columns for the complexities $\Omega(\log(n))$ and $\Omega(n\log(n))$ as PUBS-M can obtain this kind of bounds. In addition, the table distinguishes between examples where a trivial bound is obtained "$\Omega(1)$" and examples where the tool fails to return any bound "Fail".

For lower bounds CoFloCo was executed with the following options `-v 3 -compute_ubs no -conditional_lbs -stats` (see Appendix A for a description of each option). It is worth pointing out that the option `-conditional_lbs` generates piece-wise defined lower bound functions (see Section 6.9) and we consider the complexity of a piece-wise lower bound to be defined as the maximum complexity appearing in some of its partitions. This, together with the refinement, contributed greatly to the precision of CoFloCo. CoFloCo obtained a better result than PUBS-M (a higher complexity order) in 74 examples. In contrast, PUBS-M obtained better bounds in 4 examples. In 2 of these examples, PUBS-M obtained an exponential and a $n\log(n)$ bound which are not yet supported by CoFloCo. The other 2 examples are instances of a class of problem that is discussed in the future work Chapter 10.

7.2.2 Loopus's Real World Experimental Evaluation

In the recent work [SZV17], an extensive experimental evaluation was conducted. In that evaluation, 1659 functions from a compiler optimization benchmark (cBench)[9] were analyzed. This benchmark contains a total of 1027 different C files with 211892 lines of code. Table 7.3 contains a replication of this

[9] http://ctuning.org/wiki/index.php/CTools:CBench

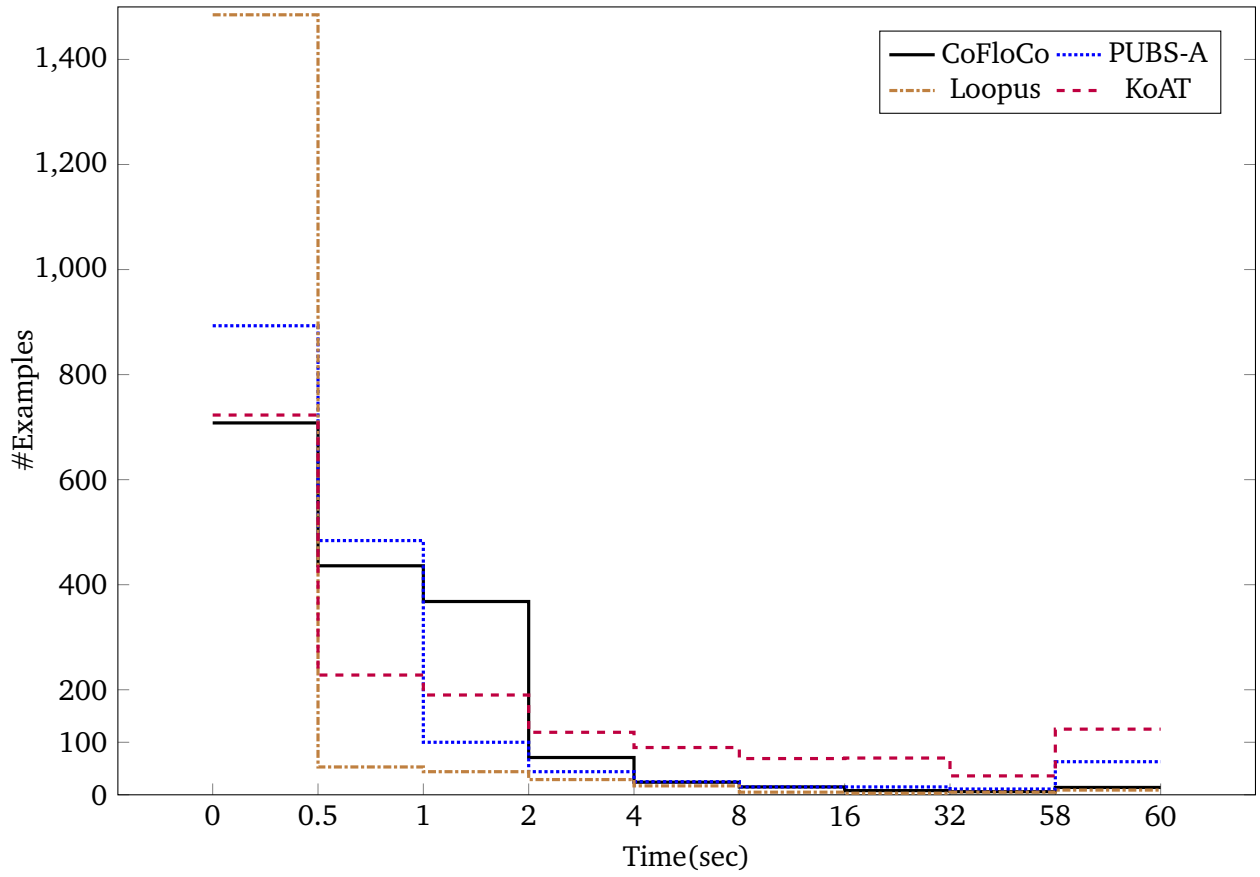


Figure 7.1.: Analysis time histogram of real world experimental evaluation: Number of examples in each time range and for each tool.

evaluation (with 1650 examples)[10] with the tools CoFloCo, KoAT, and PUBS-A. The tools C4B and Rank could not be evaluated on this benchmark as they support only a limited subset of C. This benchmark contains bigger examples so in this case Llvm2kittel was called with the following additional parameters `-multi-pred-control -only-loop-conditions` that perform slicing and simplify the generated ITS. CoFloCo was also executed with the additional parameter `-compress_chains 2`. The rest of the setup was as in the previous evaluations.

By examining the results, it became evident that CoFloCo, KoAT and PUBS-A were failing to compute a bound in many examples because the translation using Llvm2kittel does not consider structs and simple pointer references whereas these elements are better handled by Loopus. In order to isolate the effect of the translation, the examples generated by Llvm2kittel were translated back into C programs[11] and Loopus was executed on the resulting programs. This corresponds to the row Loopus * in Table 7.3. The results indicate that the translation plays a major role in the results. Factoring out the translation, Loopus *, KoAT and CoFloCo report similar results in terms of number of examples analyzed successfully. CoFloCo is better in more examples but Loopus is considerably faster.

The analysis times are not uniformly distributed. On the contrary, most of the tools worked reasonably fast for most of the examples and took a long time for a few of them. Therefore, in addition to the average and median times, a histogram of the analysis times is reported in Figure 7.1. The horizontal axis contains the analysis times in logarithmic scale (except for the last interval $58-60$ which is used for time-outs) and the vertical axis contains the number of examples analyzed in each time range. Here the differences between tools become more evident. Loopus analyzed most of the examples (1485) in

[10] Some examples were excluded because the translation tools failed.
[11] Using the script available at https://github.com/s-falke/kittel-koat.

Table 7.4.: Replication of challenging loop patterns experimental evaluation of [SZV17]: The number of examples with a given complexity, failed or timeout (T/O). The number of examples in which each tool reports a tight bound (Tight) or a finite over-approximation (Ov). The average analysis times per example in seconds with and without timeouts and the median analysis time (Md.).

	# Examples (Total 23)								Time(s)		
Tool	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	Fail	T/O	Tight	Ov	w TO	w/o TO	Md.
CoFloCo	14	6	1	1	0	1	20	2	17.57	4.73	1.33
PUBS-A	6	3	0	1	12	1	8	2	17.30	4.45	1.89
Loopus	16	5	0	2	0	0	21	2	0.09	0.09	0.05
KoAT	7	9	3	0	1	3	10	9	62.20	26.51	9.98

less than 0.5 seconds whereas CoFloCo took less than 2 seconds for 1512 examples and KoAT had more examples on the higher time intervals (it needed more than 2 seconds for 509 examples). PUBS-A was slightly faster than CoFloCo in many examples but it also timed out more (63 versus 14). Note that this distribution is likely to reflect the fact that, in practice, most functions are small and simple and only a few of them are really complex.

7.2.3 Loopus's Challenging Loop Patterns Evaluation

The work [SZV17] also contains a selection of 23 challenging integer loop patterns taken from the previous and other benchmarks that present an amortized cost. Table 7.4 contains a replication of this evaluation with the latest version of CoFloCo. In this case, the timeout is set to 300 seconds. CoFloCo was run with the additional options `-compress_chains 1` and `-n_candidates 2`. The rest of the tools were run with the same options as in the previous evaluation. This table does not include a column for programs with constant cost as all examples have at least linear complexity. Additionally, Table 7.4 does not contain a comparison with CoFloCo. Instead, the columns *Tight* and *Over-app* indicate the number of examples in which each tool computed an asymptotically tight bound or an over-approximation. Note that the columns *Tight* and *Over-app* add up to the number of examples in which the tools return a finite bound, that is, the cases where they did not fail nor timeout.

CoFloCo obtained a bound for all examples except one in which it times out. This example has many nested ifs inside loops which result in a very high number of paths. In fact, CoFloCo timed out while performing the preprocessing (Chapter 4). From the examples where CoFloCo obtained a bound, it over-approximated two, the same number as Loopus. However, the examples that were over-approximated by CoFloCo and Loopus are different. This results are significantly better than those of PUBS-A and KoAT. Note that KoAT also found a bound for most examples but it was unable to obtain amortized bounds. Consequently, it over-approximated the complexity in more examples.

In this case, the timeout is much higher (300 instead of 60) so in addition to the average times, the average times without counting timeouts are included. In this case, the behavior of the tools in terms of analysis times is similar to the previous evaluations. Loopus was the fastest, KoAT the slowest and CoFloCo and PUBS-A were in between with similar analysis times.

7.3 Evaluation on Functional Programs

In the second part of the evaluation, CoFloCo was used to analyze functional programs. In the case of functional programs, arguably one of the best existing tools is RAML [HDW17]. RAML analyzes pro-

Table 7.5.: Evaluation on Functional Programs benchmark: The number of examples with a given complexity, failed or timeout (T/O) and the average and median analysis time per file.

	# Examples (Total 182)								Time(s)/File	
Tool	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$> \mathcal{O}(n^3)$	Success	Fail	T/O	Avg	Med
CoFloCo-S	8	120	39	5	0	172	10	0	14.57	11.86
CoFloCo-M	8	93	40	9	0	150	25	7	26.47	11.74
RAML	8	93	44	23	5	173	8	1	32.05	2.09

grams written in OCaml and at the moment, there is no frontend that can generate cost relations from OCaml programs. This makes the comparison challenging. However, SACO [AAF+14] can analyze ABS programs which have a functional sub-language very similar to OCaml. Therefore, for this evaluation a subset of examples in the evaluation of paper [HDW17] was manually translated to ABS. This translation is almost one-to-one (Programs 4 and 8 are examples of functional ABS programs). However, the examples that contain references and higher order functions have been left out of the evaluation[12]. In total the benchmark contains 27 files with a total of 182 functions. In this case, the functions in each file depend on each other so they have to be analyzed together. Therefore, each file was run with a timeout of 300 seconds.

SACO implements the technique described in [AGG13] to generate cost relations from ABS programs. This technique supports two different size abstractions that produce different cost relation systems and result in different bounds. The *Sum* size abstraction considers the sum of all constructors of a term of a certain type. For example, if we have a list of lists the sum size abstraction generates a norm that represents the length of the outer list and another norm that represents *the sum of the lengths* of all the inner lists. In contrast, the *Max* size abstraction generates a norm that represents the length of the outer list and another norm that represents *the maximum length* of the inner lists. This can result in cost relations with different asymptotic complexities.

Example 7.1. Given a list of lists of integers $l := [[1,2,3],[4,5,6],[7,8,9]]$ of type `List<List<Int>>` the Sum size abstraction will generate the following norms:

$$l_1 := \text{Size}_{\texttt{List<List<Int>>}}(l) = 3 \quad l_2 := \text{Size}_{\texttt{List<Int>}}(l) = 9 \quad l_3 := \text{Size}_{\texttt{Int}}(l) = 45$$

In contrast, the Max size abstraction will generate the norms:

$$l_1 := \text{Size}_{\texttt{List<List<Int>>}}(l) = 3 \quad l_2 := \text{Size}_{\texttt{List<Int>}}(l) = 3 \quad l_3 := \text{Size}_{\texttt{Int}}(l) = 9$$

Consider the cost of a function *inc* that simply increments all the integers in the list of lists producing $[[2,3,4],[5,6,7],[8,9,10]]$. Such a function has to visit all the elements of the inner lists once. If we apply the Sum size abstraction, the complexity of a typical implementation of *inc* will be $\mathcal{O}(l_2)$ (linear) whereas if we apply the Max size abstraction, the complexity will be $\mathcal{O}(l_1 \cdot l_2)$ (quadratic).

The results of the evaluation are reported in Table 7.5. CoFloCo was executed using both size abstractions CoFloCo-S (Sum size abstraction) and CoFloCo-M (Max size abstraction) with the parameters `-v 3 -n_candidates 2 -solve_fast -compute_lbs no -stats -compress_chains 2` (see Appendix A for a description of each option). The size abstraction typically generates several norms for each variable so it becomes more important to consider multiple candidates in the strategies. Hence, the option `-n_candidates 2`. In this case, the time needed to generate the cost relations was also taken into account. RAML was executed with the parameters `analyze steps 1 6 -m` which obtains bounds on the

[12] The evaluation in [HDW17] contains examples of code generated from Coq proofs which have been also left out.

number of steps and tries to infer polynomials of degrees from 1 to 6. A higher degree did not infer additional bounds but it resulted in more timeouts.

The size abstraction of RAML corresponds (roughly) to the Max size abstraction, thus the complexities obtained by CoFloCo-M and RAML can be compared but not the ones obtained by CoFloCo-S. In that case, it is only meaningful to compare the number of successful bounds. CoFloCo-S obtained bounds for almost as many examples as RAML and the results are partly orthogonal. RAML succeeded in 7 examples where CoFloCo-S failed and CoFloCo-S succeeded in 6 examples where RAML failed. In comparison to CoFloCo-M, RAML performed better. RAML obtained a better (asymptotic) bound than CoFloCo-M in 27 examples and CoFloCo-M obtained a better bound in 8 examples. It is important to keep in mind that these examples have been taken from RAML's evaluation and many of them have been written by its developers to show the power of RAML.

There were two main reasons of failure. The first reason was due to the imprecision of the norm abstraction, in particular the abstraction of tuples. In the current implementation of SACO a tuple of two lists is abstracted to a single norm that represents the sum (or the maximum) of the lengths of the lists. Tuples are often used in the benchmark to have functions return several values. With the current abstraction, essential information is lost in these cases.

The second reason of failure is the inability of CoFloCo to infer non-linear size relations for the output variables of a cost relation. This limitation has a much greater effect on the analysis with the max size abstraction as this abstraction tends to generate cost relations with higher asymptotic complexities (see Example 7.1). A detailed discussion of this limitation can be found in Chapter 10.

In terms of performance, the comparison is also interesting. Table 7.5 includes the average and median analysis time needed per file. RAML generally worked faster for most examples but for some of them it took a really long time, presumably for examples with many variables that require polynomials of high degree. In contrast, CoFloCo-S took longer in general but its behavior did not deteriorate so fast for this kind of examples. As a result, the median time is lower for RAML but the average time ends up being lower for CoFloCo-S. CoFloCo-M performed similarly to CoFloCo-S except for a few examples where it timed out. This happened because the max operators in the size abstraction are translated into explicit case distinctions which highly increases the number of chains in the refinement phase for some examples. Finally, it is worth mentioning that SACO [AGG13] has an option to preselect important norms from all the possible existing norms. This option generates simpler cost relations with fewer variables. Unfortunately, during the experiments, a bug in the preselection was found[13] and it had to be deactivated. Once this issue is solved, the analysis times for CoFloCo are likely to improve significantly.

7.4 Evaluation on Term-rewrite Systems

The third part of the experiments includes an extension of the experiments of [NFB+17]. In that work, 965 out of the 1022 examples of the category "Runtime Complexity - Innermost Rewriting" of the Termination Competition 2016[14] were transformed into *recursive natural transition systems* (RNTS) which in turn can be easily transformed into cost relations. The results of that transformation were analyzed with CoFloCo and further transformed into ITSs to be analyzed with KoAT. These approaches were compared with pre-existing complexity analyses for term rewrite systems (TcT [AMS16] and AProVE [GAB+17]).

Here, the benchmark was re-analyzed with CoFloCo and, as an additional comparison, it was also analyzed with PUBS-A following the same approach as in previous experiments. We refer to the original paper for a comparison to other approaches. Note though, that according to their results, the two best tools for analyzing the complexity of term rewrite systems TcT [AMS16] and AProVE [GAB+17] obtained a bound for 380 and 427 examples respectively. In comparison, CoFloCo obtained a bound for 382 examples.

[13] The authors have been notified of this bug.

[14] http://termination-portal.org/wiki/Termination_Competition/

Table 7.6.: Evaluation on Term-rewrite Systems benchmark: The number of examples with a given complexity, failed or timeout (T/O).The number of examples in which CoFloCo reports a better or worse bound than each tool and the average (Avg.) and median (Md.) analysis times per example in seconds.

	# Examples (Total 965)								CoFloCo is		Time(s)	
Tool	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3) > \mathcal{O}(n^3)$		Exp	Fail	T/O	Better	Worse	Avg.	Md.
CoFloCo	42	208	107	21	4	0	521	62	-	-	5.84	1
PUBS-A	21	166	60	10	3	12	635	22	148	16	2.09	0.48

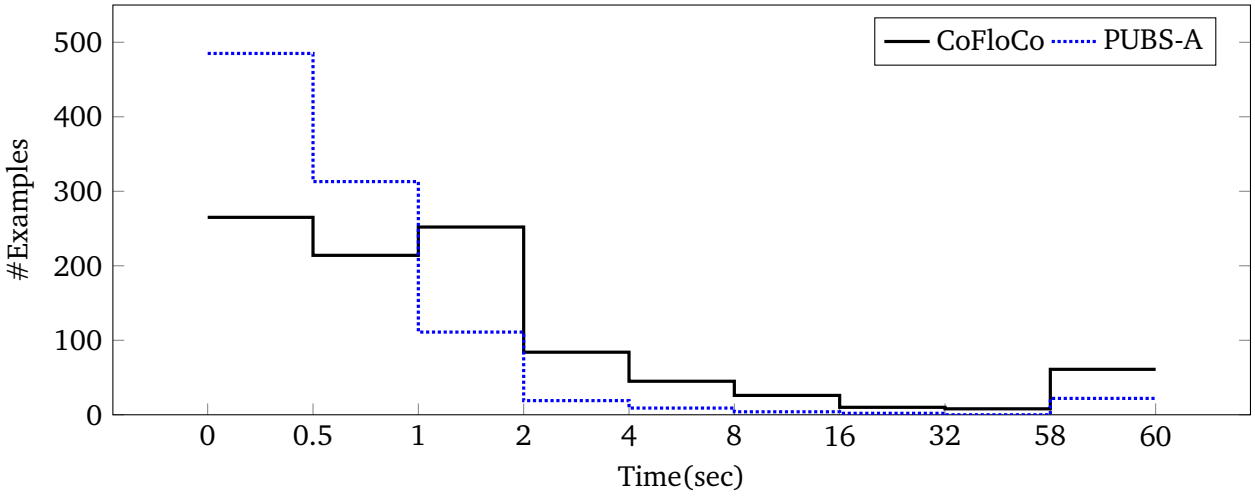


Figure 7.2.: Analysis time histogram of term-rewrite systems evaluation: Number of examples in each time range and for each tool.

In this evaluation, CoFloCo was executed with the options `-v 3 -solve_fast -compute_lbs no -stats -compress_chains 2`. An important aspect of term rewrite systems is that the evaluation can start at any point. This was simulated by having a special cost relation *start* that calls all the other cost relations. As a result, the CR *start* often contains many chains with affects the performance of the analysis negatively. This also means that the best case lower bounds of the generated CRS are trivial. Consequently, no evaluation with respect to lower bounds was performed.

This benchmark is interesting because it is significantly bigger than the evaluation on functional programs and it also contains many examples with non-tail recursion (328) and multiple recursion (300)[15]. Table 7.6 contains the results of the analysis and Figure 7.2 contains a histogram with the analysis times (with the same format as Figure 7.1). CoFloCo obtained a better bound in 148 examples and a worse bound in 15 examples. From these 15 examples, there are 11 examples where PUBS-A reported an exponential upper bound and 3 examples where CoFloCo timed out. Therefore, we can conclude that CoFloCo is much more powerful than PUBS-A for this kind of examples although it is also slower.

7.5 Limitations of the Evaluations

Achieving a meaningful comparison among different tools is challenging. Different tools expect different formats, analyze programs written in different languages, and make different assumptions on the program semantics. For example, the evaluations on imperative programs do not really show a comparison

[15] Some examples have both multiple recursion and non-tail recursion.

between CoFloCo and Loopus but rather a comparison between Llvm2kittel +CoFloCo and Loopus. Similarly, the evaluation on functional programs compares RAML to SACO+CoFloCo. How the translation between formats is done can also have a great effect on the results of the analysis. The same program can often be represented in many ways that are formally equivalent but not equally easy to analyze.

In the case of the comparison to RAML, the fact there is no frontend to generate cost relations from Ocaml programs limited the size of the benchmark that could be considered. There are other tools, such as SPEED [GMC09], that could not be included in the evaluation because they are not publicly available. However, the examples from its papers are all included in the first experimental evaluation.

Finally, these evaluations did not test all the capabilities of CoFloCo. For example, the analysis was not used to obtain net-cost bounds of CRS with non-cumulative resources even though CoFloCo has support for it. Besides, CoFloCo was executed in all evaluations using fixed settings. Further evaluations could be performed to test the effect of different options on the precision and performance of the results.

8 Related Work

There is an extensive body of work related to cost analysis. In the introduction, the approaches based on cost relations [AAGP11, AGM13, ABAG13] and their limitations were explained in detail. Also, a significant part of the thesis has been devoted to explain how CoFloCo overcomes these limitations and the experimental evaluation has shown how CoFloCo indeed obtains better bounds for many programs. Therefore, this section focuses on other (competing) approaches to cost analysis (Section 8.1) and on related research areas (Section 8.2).

8.1 Competing Approaches

8.1.1 Recurrence relations

Many early works on cost analysis are based on the extraction and solution of recurrence relations. The work of [Weg75] is one of the first ones that attempts to obtain cost upper bounds of programs automatically. METRIC, the developed system, obtains cost bounds of first order Lisp programs. The key idea is, given a recursive function, to extract a recurrence relation that represents the cost of the program in terms of the size of the input. Then, by solving (over-approximating) the recurrence relation, one can obtain a closed-form upper bound of the cost of the program.

A recursive function can contain calls to other functions whose cost has to be approximated first using the same mechanism. Additionally, the arguments of the recursive calls might depend on the result of some of those other calls. That means that in order to generate a recurrence relation for the cost, the size of those results has to be approximated first. These sizes can be approximated by solving additional recurrence relations that represent how the size of the output varies with respect to the size of the inputs in the intermediate call.

This idea of extracting and solving recurrence relations has been applied to different programming languages. It is worth mentioning [DLH90] which obtains upper bounds for deterministic logic programs. This work was later extended in [DL93, DLHL94] to deal with some of the additional challenges associated to logic programs such as non-determinism, multiple solutions, failure, etc., and in [DLHL97] to compute lower bounds.

Recurrence relations are also used to compute bounds for strongly typed functional programs [Ben01, Gro01, Vas08, VH04]. The approach in [Gro01] focuses on extracting cost recurrences (recurrence relations that represent the cost of the program) using dependent types, but not on solving the extracted cost recurrences. On the other hand, [Ben01] puts a greater focus of the recurrence relation solving and, in particular, in how to simplify the recurrence relations obtained from programs so they can be solved by existing tools such as Mathematica [Wol03]. In addition, specialized solvers, such as PURRS [BPZZ05], have been developed to solve wider classes of recurrence relations.

The works by Vasconcelos et al. [Vas08, VH04] are based on *sized-types*. In this approach, data types are annotated with an upper bound of their size and function types are annotated with an upper bound of their cost. The paper [VH04] presents a type inference system that collects a set of constraints and recurrence equations on the cost and the size annotations. One can obtain the cost of a program and the size of its output by solving those constraints and recurrence relations. Sized-types and recurrence relations are also used for the analysis of logic programs in [SLH14] within an abstract interpretation framework. The approaches do not try to obtain a single recurrence for the complete program but smaller recurrences to approximate how the sizes of different variables change along the execution. In this way, they decompose the problem in smaller and simpler sub-problems.

Nonetheless, these works have important limitations on the class of programs they can analyze. This is because both extracting and solving recurrence relations constitute challenging problems. Functions and loops in real programs often differ from the ideal recurrence patterns expressible with simple recurrence relations. On the contrary, they present characteristics that make the recurrence relations extraction and solving hard, especially in the case of imperative programs.

For example, a program's cost often depends on *multiple variables* but existing recurrence solvers have limited support for recurrences with multiple variables. Abstracted programs are also usually *non-deterministic* because of features of the programming language that cannot be precisely modeled, or because of the size abstractions. This non-determinism has to be solved in order to extract a recurrence relation that represents the worst case. Unfortunately, solving or approximating non-determinism becomes harder in the presence of *non-monotonic cost*. If a program has non-monotonic cost, we cannot obtain an upper bound of its cost by assuming variables take their maximum possible value. In fact, most works based on recurrence relations [DLH90, DL93, DLHL97, Vas08, VH04] work under the assumption that programs have size-monotonic cost. While this is often true for programs that manipulate algebraic data structures and for the usual size norms, it is certainly not true for programs with integers variables. For example, the cost of `while(i<n){i++}` is non-monotonic with respect to variable `i` (the smaller the initial value of `i`, the higher the cost). Finally, loops and recursive functions often have *multiple paths* that need to be abstracted into a single recurrence which can also be challenging.

The work by Albert et al. [AGM13] obtains cost bounds of imperative programs using recurrence relations. In fact, it tackles many of the problems of extracting recurrence relations and it does it using cost relations as an intermediate representation and with the help of ranking functions and invariants. Nonetheless, as described in the introduction, this work still suffers from important limitations that are addressed in this thesis.

8.1.2 SPEED

During the SPEED project, several cost analyses for imperative programs were developed [GG08, GJK09, GMC09, GZ10, ZGSV11]. In general, each of the analyses represents an improvement over the previous ones. This section provides a short description of all of them followed by a comparison of the last one [ZGSV11] to CoFloCo.

The paper [GG08] presents an extension of a linear abstract domain to support non-linear relations and the max operator. This abstract domain is then applied for computing upper bounds of some small programs including max and logarithms. The bound computation algorithm uses a single counter instrumentation. That is, a counter is added to the program and it is incremented in each loop iteration. Then, an invariant is inferred on the maximum value the counter can take. Such a value corresponds to an upper bound on the number of loop iterations. The algorithm relies on the power of the extended abstract domain to compute non-linear invariants and consequently non-linear bounds.

The approach of [GMC09] is based on multiple counter instrumentation. It annotates the program with multiple counter variables and uses linear invariant generation to relate the values of the counters to the input values of the program. In order to measure the total loop iterations, each back-edge of the control flow graph has to be annotated with a counter increment. Disjunctive non-linear bounds are obtained by having several counters that can depend on each other. A counter depends on another if it is reset whenever the other counter is incremented. All the counter dependencies have to form a DAG (directed acyclic graph) and the total bound of the program is obtained by combining the bounds of the individual counters. For example, the bounds of counters that depend on others are multiplied. This technique does not work well in the presence of nested loops in which the inner loop affects the number of iterations of the outer loop. It also fails when disjunctive reasoning is needed to obtain a bound but there is only one back-edge (and thus one counter) in the loop.

In the paper [GJK09] a bound analysis is presented based on control-flow refinement and progress invariants. The control-flow refinement presented in that work presents some similarities with the re-

finement of cost relations presented in Chapter 5. The bound computation procedure is based on progress invariants which relate the state of the program at a location between two visits to such location. Such an invariant can be used to conclude, for instance, that the counter of an inner loop has not been reset during the iteration of an outer loop.

The work [GZ10] focuses on obtaining bounds for the number of times a single program location is visited. This can be used to compute the number of iterations of a loop. The approach consists on generating a disjunctive invariant that relates the states of the program between two consecutive visits of a location[1]. In order to achieve that, inner loops are summarized with disjunctive input-output relationships. Once this invariant has been computed, ranking functions are inferred for each of the disjunctive components of the invariant. Each disjunctive component can be seen as an abstracted loop path. Then, these ranking functions are composed (by adding them, multiplying them or taking their maximum value) to obtain a final bound.

Finally, [ZGSV11] provides a bound algorithm based on the Size-change abstraction. The algorithm has four steps:

1. First, a set of norms is selected for the program. Norms are functions from the program state to integers and they are chosen heuristically so they decrease in some loop path of the program. For instance, in a program `while(x<y)x++`, the expression `y-x` is a norm that bounds the number of iterations of the program. By choosing these norms, one can get rid of large parts of the program that do not affect its cost and make the later analysis simpler and more scalable.

2. Global invariants are computed using standard abstract domains such as polyhedra or octagon. These invariants are later used to relate bounds at a certain program location to the input values of the program.

3. The program is abstracted with respect to the chosen norms into a size-change graph. To compute the bound of a loop (or location), disjunctive summaries are computed for the inner loops and all the loop paths are enumerated. This step is similar to the one performed in [GZ10] but in the context of size-change graphs disjunctive summaries can be efficiently computed.

4. Finally, given a set of loop paths, *contextualization* is applied. Contextualization checks whether a path can be followed by another and creates a "call-graph" over loop paths. Then bounds are computed and composed for each of the SCC of the call-graph and expressed in terms of the input variables using the global invariants.

This approach shares several ideas with the work presented here. The refinement propagation in Chapter 5 effectively incorporates disjunctive summaries of the inner loops in the cost relations. However, the disjunctiveness in CoFloCo is guided by the refinement into chains (there is one summary per chain). Besides, chain summaries are much more expressive as they are general linear constraint sets instead of size-change constraints.

Also, *contextualization* is similar to the control flow refinement of cost relations. In the cost relation approach, a CE call-graph is computed and SCCs become phases that later are combined into chains (remember that for imperative code each CE in a CR represents a loop path). However, in [ZGSV11] there is nothing equivalent to the chain enumeration, the computation of calling contexts and chain summaries, and CE strengthening.

The bound computation differs considerably. The algorithm in [ZGSV11] computes the number of visits to a single location. In contrast, the presented algorithm computes the overall cost of the program and, as such, it also has to compose the bounds of the inner loops precisely. Besides, CoFloCo does not rely on a single global invariant to express bounds in terms of the input variables but performs this transformation incrementally as it composes the bounds of the different program parts.

[1] Progress invariants from [GJK09] are valid for two visits that do not have to be consecutive.

None of these approaches described above can infer amortized bounds for programs such as Program 3, and with the exception of [GMC09], they do not support recursive programs. They are also not compositional. Unfortunately, none of these approaches is publicly available so no experimental comparison could be performed. However, our experimental evaluation includes all examples from these papers and it includes a comparison to Loopus which is, in many aspects, a successor of the approach of [ZGSV11].

8.1.3 Loopus

There are several implementations of Loopus [SZV14] and [SZV15] (extended in [SZV17]) that implements slightly different techniques. This section focuses on the last (and most powerful) version described in [SZV17]. The approach presented in [SZV17] has some conceptual similarities to the work of [ZGSV11]. As in the analysis of [ZGSV11], a set of norms is heuristically selected from the program. Given this set of norms, the program is abstracted into a representation where an efficient analysis is possible. In contrast to [ZGSV11], this representation is not a size-change graph but a graph annotated with difference constraints.

A difference constraint has the form $x' \leq y + c$ where x' and y are variables, c is an integer constant, and it represents that the new value of x is bound by $y + c$ from above. Difference constraints can represent increments (e.g. $x' \leq x + 1$) and resets (e.g. $x' \leq y$) which are the most typical operations over loop counters.

Once the abstract representation has been computed, the bound algorithm is given as a mutually recursive definition between transitions bounds (how many times a transition can be taken) and variable bounds (the maximum value of a variable at a program location). Given an expression that represents a local transition bound, it checks how often this expression is incremented and reset by checking the bounds of the transitions that increment or reset such expression. It also checks to which value an expression can be reset by obtaining the variable bounds of the expression at the locations where it is reset.

This idea has been adopted in how cost structures for phases are computed in Section 6.5. There, there is a similar interplay between the computation of constraints for *smiv* and $\lceil iv \rceil / \lfloor iv \rfloor$. However, in [SZV17] this is done at a global level, that is, no summaries of the inner loops are computed and the algorithm considers the complete abstracted program at once. On the other hand, in the present work this interplay between transition bounds and variable bounds (here *smiv* and $\lceil iv \rceil$) is only done at the level of phases or chains but it considers arbitrary linear expressions instead of a set of prefixed norms.

Loopus can compute amortized bounds of programs with challenging loop patterns, as demonstrated in the experimental evaluation. Moreover, the early abstraction using norms, which is mostly based on symbolic execution and syntactic checks, makes it very fast. On the downside, the approach is not compositional, i.e. the complete program is considered at once. In principle, it can deal with programs formed by multiple functions by inlining them but no experiments have been done in this respect. Moreover, the approach does not support recursive functions at the moment.

8.1.4 KoAT

Another significant approach is the one implemented in KoAT [BEF+14] (extended in [BEF+16]). It obtains complexity bounds of integer programs, represented as integer transition systems (ITS), by alternating size and bound analysis. This idea is similar to the one adopted by Loopus (note that the publication [BEF+14] predates [SZV15]). KoAT uses polynomial ranking functions to find bounds on the number of iterations of transitions. Then, it obtains bounds on the size of variables by checking how often they are incremented or multiplied using the transition bounds.

This approach computes transition bounds and size bounds incrementally, considering only part of the program at once and following a top-down strategy. In order to be able to compose transition bounds

Program 17

```
while (x>0){
 y=x;
 while(y>0){
  x--;
  y--;
 }
}
```

Figure 8.1.: Program 17: Challenging example for KoAT's bottom-up approach

and size bounds in such a way, the approach considers weakly monotonic bounds expressed in terms of the absolute values of the variables. Unfortunately, this can result in an important loss of precision in some cases. In contrast, CoFloCo can infer polynomial bounds that are combinations of arbitrary linear expressions on the program variables (which might have negative coefficients). Conversely, Loopus relies on its norm preselection and abstraction to obtain non-monotonic bounds.

In contrast to Loopus and CoFloCo which only obtain polynomial bounds, KoAT can obtain exponential bounds arising from two situations: It can obtain exponential bounds caused by multiple recursion in a way similar to PUBS [AAGP11] and also it can obtain exponential bounds caused by loops that grow the size of a variable to an exponential size in terms of the input. The latter case is, to the best of my knowledge, not supported by any other tool.

In [BEF+16] KoAT has been extended to support a bottom-up approach which is more similar to CoFloCo's. This bottom-up approach can obtain amortized bounds in some cases. However, it cannot obtain the precision of CoFloCo's analysis because it does not perform any kind of control-flow refinement. An example of this is Program 17 (taken from [BEF+16]) in Figure 8.1. In that program, the bottom-up approach of KoAT fails to even prove termination of the outer loop because it cannot guarantee that x decreases in the inner loop. In contrast, CoFloCo obtains a linear bound because CoFloCo's refinement guarantees that after executing the inner loop both x and y are zero. Therefore, CoFloCo concludes that the outer loop can only iterate once. KoAT resorts to heuristics to decide when to apply the bottom-up or the top-down approach.

With respect to recursion, KoAT analyzes an extended representation of ITS that can represent recursive programs. However, this representation is quite limited as it ignores the results of the recursive calls. In the very recent work [NFB+17], a similar approach is adopted to analyze *recursive natural transition systems* (RNTS) which are a generalization of ITS that fully support recursion but are limited to natural numbers. In this case, the approach follows a bottom-up modular strategy. At each step, small ITS are generated that represent individual symbols of the RNTS (similar to cost relations) that can be solved by KoAT. This is similar to CoFloCo's strategy but instead of relying on chain summaries, it uses KoAT to compute size bounds of the result of the inner calls. Note though that chain summaries are useful not as size relations but also for the control-flow refinement of cost relations. Extending CoFloCo to support non-linear size relations compositionally is part of the future work (see Chapter 10).

Finally, KoAT only computes upper bounds. There is a related tool called LoAT [FNH+16] that infers lower bounds of ITS. However, LoAT infers worst case lower bounds which are not comparable to the best case lower bounds inferred by CoFloCo.

8.1.5 Rank

The paper [ADFG10] presents a technique to infer multi-dimensional ranking functions for programs represented as ITS. Using the computed ranking functions and the invariants at each location, it is sometimes possible to over-approximate the number of visits to each location by counting the integral points of a polyhedron. The tool Rank (included in the first experimental evaluation) implements this

approach. The technique worked reasonably well in the evaluation. However, it has no support for recursion, it is not compositional, and it relies on other tools to provide the invariants for the analysis.

8.1.6 RAML

The RAML project computes cost upper bounds of programs written in RAML (Resource aware ML) and, more recently, of programs written in OCaml. It is based on the potential method for cost analysis [Tar85] in which one assigns a potential to the data structures of the program and such a potential is used to pay for the operations on such data structures. If the potential can be assigned in such a way that all operations can be paid, the initial potential at the beginning of the execution represents an upper bound on the cost of the program.

The key aspects of the approach are the following:

- The potential of each variable is given by a polynomial template with unknown coefficients.
- A type system is given that generates constraints for the coefficients of the potential. The typing rules are mostly syntax directed.
- The considered polynomial templates have special format (resource polynomials) that allows the easy composition and decomposition of the potential of data structures and can represent a wide variety of bounds while having only positive coefficients. The typical operations over data structures in functional programs, i.e. pattern matching, generate only linear constraints over the coefficients. Therefore, obtaining the potential expression amount to solving a linear programming problem, which can be done very efficiently.

The paper [HH10b] introduces polynomial potentials for the first time. [HH10a] complements this work by defining a partial semantics to compute bounds of possibly non-terminating programs and dealing with polymorphism. In [HAH12], the work is extended to obtain multivariate polynomial bounds, that is, bounds that might depend on the product of several variables. The work [HS15] extends these techniques to support naturals and arrays to some extent. Finally, in the recent work [HDW17], these techniques are extended to deal with user-defined data structures and higher-order functions to analyze Ocaml programs.

This analysis can obtain precise amortized bounds and it is generally quite efficient, as demonstrated in the experimental evaluation. However, it also has some limitations. Its support for integers is very limited and it cannot obtain bounds that depend on relations between data structures, for example, a bound that depends on the difference in length of two lists. Moreover, the efficiency of the approach can be severely affected as the number of variables and the degree of the polynomials increases. It is also not clear how it can be extended to obtain other kinds of bounds such as logarithmic or exponential.

The system C4B [CHS15] (included in the first experimental evaluation) adapts this approach for C programs with integers. Instead of assigning a potential to individual variables, it assigns a potential to variable differences, that is, it assigns a potential to expressions of the form $\|x - y\|$. Its results are promising but it can only infer linear bounds at the moment.

8.1.7 Lower Bounds

There are few systems focused on obtaining best case lower bounds. There are some analyses for logic programs [DLHL97, SLH14] and a version of PUBS [AGM13] for cost relations (present in the experimental evaluation with the name PUBS-M). It is hard to ascertain the power of the analyses [DLHL97, SLH14] without an experimental evaluation. They analyze logic programs which are quite different from imperative programs. However, they are likely to suffer from similar limitations as [AGM13] as they are also based on solving recurrence relations and they do not perform any kind

of control-flow refinement. Thus, they will fail to produce non-trivial lower bounds for programs with complex control flow. They are also unable to compute non-trivial bounds for programs that present amortized cost. Note that the analyses in [DLHL97, SLH14] focus on other aspects of cost analysis that are specific of logic programs, such as counting failures or predicting the number of solutions.

8.2 Related Research Areas

8.2.1 Term Rewriting

Another active area of research focuses on the analysis of term rewrite systems (TRS). This line of work has traditionally focused on termination and its most widely used technique is based on the *dependecy pair* framework [AG00]. Recently, complexity analysis has received greater attention and many of the techniques initially designed for termination have been adapted for complexity analysis using *weak dependency pairs* [HM08] or *dependency tuples* [NEG13].

The complexity of a term rewrite system is obtained incrementally by applying *processors* to a *complexity problem*. Processors typically obtain a well-founded ordering of the terms that provides a bound on the number of evaluation steps (for example using polynomial interpretations or matrix interpretations), transform the problem into a simpler one, or subdivide it into smaller complexity problems. For example, several processors use the notion of dependency graph, which is similar to a control flow graph or a call graph in the context of TRS, to simplify the complexity problem. The final result of the analysis is a proof tree[2] where each node corresponds to the application of one processor.

The paper [AM16] describes a general framework for inferring complexity bounds of TRS that is general enough to cover different notions of complexity, and types of term rewrite systems. This framework has been implemented in the tool Tyrolean Complexity Tool (TcT) [AMS16] and it supports a certain degree of certification [AST15]. Another powerful tool that computes the complexity of TRS is AProVE [GAB+17].

These complexity analyses for TRS focus only on obtaining monovariant asymptotic bounds. That is, they compute a complexity on the overall size of the initial terms. In contrast, CoFloCo aims at obtaining precise bounds up to the constant level that can depend on several arguments of the entry cost relation. In addition to that, most approaches for the complexity analysis of TRS cannot compute the complexity of fragments of the TRS and compose them but have to consider the complete TRS at once. The work [AM16] is an exception as it presents an approach to decompose TRS according to the strongly connected components of the call-graph. However, this decomposition is not complete and it cannot obtain amortized complexity bounds.

Some work has been done to apply this technology for the analysis of higher-order functional programs. For example, in [ALM15] a complexity preserving transformation is proposed to generate TRS from pure higher order Ocaml programs.

Finally, there has been some effort [HM14, HM15] to adapt the amortized resource analysis based on the potential method to analyze the complexity of term rewrite systems. However, to the best of my knowledge, this approach has not yet been implemented.

8.2.2 COSTA and SACO

There are several works that generate cost relations from a variety of programming languages. In [AAG+12], cost relations are generated from programs written in Java bytecode. This approach is implemented in the COSTA system [AAG+08].

[2] Or a proof chain if there are no processors that split the problem into several sub-problems.

This work has been adapted to analyze concurrent ABS programs in [AAG+11]. This analysis has been implemented as part of the tool SACO [AAF+14] (Static Analyzer of Concurrent Objects) which includes a wide range of static analyses for ABS programs. The approach presented in [AAG+11] is quite limited when the termination or the cost of a program depends on the concurrent interleaving of loops. In [AFGM17], we developed an improved analysis that can obtain bounds for loops that are executed in parallel and influence each other.

Within SACO, a series of advanced cost analyses [ACMMRD14, ACJRD15, ACRD14] have been implemented. These analyses focus on concurrency-specific aspects of cost analysis. [ACMMRD14] presents an analysis to compute bounds on the amount of data transmitted over a network in a distributed application. [ACRD14] and [ACJRD15] focus on the peak cost and parallel cost consumption of concurrent programs respectively. The analysis in [GLL15] is also closely related. It implements a cost analysis to count the number of virtual machines used by a program that can allocate and release virtual machines. There has also been some work focused on obtaining bounds for memory use [AGGZ13] and for generic non-cumulative resources [ACRD15].

Finally, an essential aspect of cost analysis based on cost relations is to abstract data structures to integer norms precisely. This is the focus of the paper [AGG13], which uses type information to abstract complex data structures to (possibly) several integer norms. This is the frontend that has been used generate cost relations from ABS programs to compare CoFloCo to RAML (Section 7.3). All these works are part of SACO or COSTA and are complementary to CoFloCo, which they can use as a backend.

8.2.3 WCET

WCET (Worst Case Execution Time) analysis is a related research area that attempts to obtain precise time bounds on the execution time of a program in a given computer architecture. Instead of considering a simple cost model, WCET analysis focuses on modeling low level aspects of the behavior of hardware components such as processor pipelines and cache memories. [WEE+08] contains a survey. Often, WCET tools focus on programs where loops have a fixed number of iterations or they assume that loop bounds are provided as an input. In this context, Knoop et al. [KKZ11] propose a technique to obtain symbolic bounds for loops using recurrence relations. However, this technique is limited to a certain class of for-loops.

Traditionally, a WCET problem is reduced to integer linear programming (ILP) that encodes the dependencies between the paths of the program. The solution to this problem gives the WCET of the program. In [CHK+15], the authors provide a different approach that represents programs as abstract segment trees. This allows splitting the problem into smaller ILP problems instead of having to solve a single one. Abstract segment trees present some similarities to the CR representation. They represent the program hierarchy and they contain “segments” which represent program paths as regular expressions over program locations (or other segments) similarly to how chains encode the possible behaviors of a program. However, the refinement in [CHK+15] is performed in a counter-example guided style.

8.2.4 Cost Bounds Verification

There has been some work concerned with the verification of the bounds generated by cost analysis tools. This is an important problem because cost analysis tools grow quite complex and it is hard to ensure they are bug free. For some applications, such as certified software, cost bounds are of little use if they cannot be highly trusted. There has been some work on certification of cost bounds based on type systems [CW00] and quantitative Hoare logic [CHS15, CHRS14]. Avanzini et al. [AST15] present a verified checker developed in Isabelled that certifies that the complexity proofs generated by TcT [AMS16] are sound. Finally, the cost bounds generated by COSTA (using its backend PUBS) can be verified with the KeY system [ABG+16].

8.2.5 Termination and Ranking Functions

Termination and cost analysis are closely related problems. In fact, time cost analysis can be seen as a quantitative version of termination analysis. Consequently, many of the techniques originally developed for termination analysis can be useful for cost analysis. There is an enormous body of work on termination analysis. This section does not try to give an exhaustive account of all but rather provide some pointers to the most relevant developments concerning cost analysis.

The tool T2 [BCI+16] uses lexicographical ranking functions and cooperation programs [BCF13] to prove termination. CppInv [LORR13] makes use of Max-SMT technology to generate ranking functions and supporting invariants for termination. Max-SMT allows obtaining useful information that can serve to further refine the program even when the ranking function generation fails. The Ultimate Büchi Automizer [HHP14] decomposes programs into modules that can be proved terminating using techniques for termination of lasso programs. There are also very powerful tools for term rewrite systems such as AProVE [GAB+17] and TTT2 [KSZM09]. AProVE can also analyze the termination of programs written in multiple languages by transforming them into a term rewrite system with builtin integers.

A key element for many termination and cost analyses is the inference of ranking functions. Ranking functions not only provide a termination argument, but also a specific bound on the number of iterations of a loop (or recursive program). In [PR04], Podelski et al. present a complete method to infer linear ranking functions over the rationals for single path loops. This method is based on template inference using Farkas' lemma. Note that this idea is used to generate candidates for final constraints in the bound computation algorithm (see Section 6.5.2). In [ADFG10], a complete method is presented for lexicographical ranking functions over the rationals for arbitrary control-flow graphs. This method is then applied to obtain cost bounds in the tool Rank. The work of Ben-Amram et al. [BAG14] provides a complete method for linear ranking functions over the integers and lexicographic ranking functions over the integers and rationals[3] for multiple-path constraint loops. In [LH15], a template based method is developed to generate linear ranking functions, multi-phase, piece-wise and lexicographic ranking function. This method is based on Motzkin's transposition theorem instead of Farkas' lemma to support strict inequalities.

Finally, Urban et al. [UM14] have developed a parametric abstract domain in which the elements of the abstract domain are decision trees with linear constraints on the nodes and elements of a numerical abstract domain at the leaves. This can be instantiated to define piece-wise ranking functions and it can be used to prove termination (and probably for cost analysis as well). It is worth pointing out that in Section 6.9, a similar decision tree is built to obtain piece-wise defined bounds. However, in Section 6.9 this is done as a post-processing step after the main analysis has been completed and not within an abstract interpretation framework.

8.2.6 Verification of Programs with Constrained Horn Clauses

Cost relations are very similar to constrained Horn clauses (CHC) annotated with cost. In recent years, there has been a significant amount of work that uses constraint Horn clauses as an abstract representation for program verification (see [BGMR15] for some references). The techniques used to translate programs into CHC could be adapted to generate cost relations, although additional care has to be taken with cost relations to make sure that the cost of the program and the non-terminating behaviors are preserved. Also, Horn clause solvers could be used to assist cost analysis tools based on cost relations to refine the program and infer better supporting invariants.

[3] The definition of lexicographic ranking functions in [BAG14] is more general than in [ADFG10].

9 Conclusion

This thesis presents a new cost analysis based on an extended notion of cost relations. These cost relations include input and output variables and can contain multiple constraint sets that are considered sequentially. A formal semantics for cost relations has been developed that considers non-terminating evaluations explicitly. In addition, other restrictions have been (partially) lifted, for example, cost relations can now have positive and negative cost annotations. The analysis can infer upper and lower bounds on the net-cost of a program and on the peak-cost of programs with cumulative resources.

The analysis contains three parts. The first part reduces any mutually recursive cost relations to cost relations that only have direct recursion using unfolding. The second part consists of a refinement of cost relations that partitions all possible executions of the program into execution patterns, represented as chains. At the same time, it uses this information to infer precise invariants, input-output relationships, discard unfeasible execution patterns, and prove termination. In the third part of the analysis, cost bounds are inferred compositionally with the help of cost structures. Cost structures reduce the computation of complex bounds to the inference of simple constraints (implemented through strategies) using linear programming. They can represent polynomial upper and lower bounds of programs with max and min operators. They can also maintain multiple bound candidates in a single representation. On top of that, cost structures are very flexible and can be easily extended to handle other classes of bounds such as exponential, logarithmic and bounds with binomial coefficients (see future work in Chapter 10). At the end of the analysis, the bounds from the different chains can be combined to obtain a piece-wise defined bound. No other tool can produce this kind of bounds. Both the refinement and the bound computation have been proved sound with respect to the given cost relation semantics.

The analysis has been implemented in the tool CoFloCo. CoFloCo overcomes many of the limitations existing in previous approaches based on cost relations. In particular, it can deal much better with programs that have a complex control flow. For example, programs with nested loops that influence each other, loops with multiple phases and programs that present amortized cost.

CoFloCo has been evaluated against other state-of-the-art tools and with respect to a variety of benchmarks. These benchmarks include: a set of small but challenging C programs taken from the literature, an extensive set of real world C programs taken from a compiler optimization benchmark, a small set of C programs with challenging iteration patterns, a medium-sized benchmark of functional programs and a large benchmark of term rewrite systems. The approach is highly competitive with other cost analyses, many of them developed in parallel e.g., KoAT, Loopus and C4B. In some of the categories such as “examples from the literature” it obtains the best results, and in other categories such as “challenging loop patterns” or “functional programs from the RAML evaluation”, it performs almost as well as the leading tools which are language specific (Loopus and RAML respectively).

The use of cost relations makes CoFloCo very versatile i.e. it can be easily applied to different programming languages. This is also illustrated in the experimental evaluation. At the same time, CoFloCo does not suffer from some limitations present in some of the other analyses, such as the inability to analyze recursive programs (Loopus), or obtain non-monotonic bounds (KoAT). It is also one of the few existing tools that compute best case lower bounds, supports both positive and negative cost annotations and is able to obtain amortized bounds despite following a compositional bottom-up approach.

CoFloCo has also some limitations. However, most of them are not fundamental to the analysis. That is, many of the current limitations could be solved by extending the analysis while maintaining the general approach. In the next chapter, some of these limitations are explained in detail together with possible solutions.

10 Limitations and Future Work

Despite the progress made in this thesis, CoFloCo still presents some limitations that should be addressed in order to make it applicable in practice.

10.1 Logarithmic and Exponential Bounds

The cost structures presented in this thesis can only represent polynomial bounds (with max and min operators). However, it should be straightforward to extend them and their corresponding strategies to support constraints with exponentials and logarithms following as similar approach as in [ABAG13]. The approach of [BEF+16] could also be adapted for bounding Max and Min variables in the cases where they grow exponentially.

10.2 Non-linear and Non-local Size Change

Within a cost relation, the mechanism to compute constraints for Max and Min variables can obtain non-linear size relations of linear expressions. However, at the moment this is only done at the local level (within a phase) whereas chain summaries constitute the size relations that are used to compose the costs of different cost relations. Unfortunately, chain summaries are linear and cannot reflect non-linear cost relations.

Example 10.1. Program 16 (in Page 98) is an example that requires a non-linear size bound. Section 6.7 contains an approach that can successfully obtain a bound for this program with the CR representation of Figure 6.7. However, this is not the usual CR representation. In that representation, the second loop is encoded as a continuation of the first one thus breaking modularity. The usual representation, in which each loop represents an independent cost relation, is presented in Figure 10.1. Under this representation CoFloCo successfully obtains a bound for the chains $(3)^+(2)$ and $(5)^+(4)$. However, in order to compose these bounds, it uses the constraint set in CE 1.1 which does not capture that $b' \leq a^2$. Consequently, CoFloCo fails to obtain a bound for CE 1.1.

Adopting the continuation style representation of Figure 6.7 can work in cases where there is a simple sequential composition, but it is not a general solution.

Example 10.2. Consider Program 18 in Figure 10.2 (the function *append* has been omitted). This function generates a list of all the possible pairs of the elements in the input list. The recursive cost equation of *pairs* is:

$$pairs(l : lo) = \{l > 0, l = 1 + x + xs\},\ 1,\ pairs(xs : xso),\ attach(x, xs : xs'),\ append(xso, xs' : lo)$$

1.1: $koat(a) = \{a > 0, b = 0, b' > 0\},\ wh3[(3)^+(2)](a, b : b'),\ wh7[(5)^+4](b')$
1.2: $koat(a) = \{a \leq 0, b = 0, b' = 0\},\ wh3[(2.2)](a, b : b'),\ wh7[4](b')$

2: $wh3(a, b : bo) = \{a \leq 0, bo = b\}$
3: $wh3(a, b : bo) = \{a > 0, a' = a - 1, b' = b + a\},\ 1,\ wh3(a', b' : bo)$

4: $wh7(b) = \{b \leq 0\}$
5: $wh7(b) = \{b > 0, b' = b - 1\},\ 1,\ wh7(b')$

Figure 10.1.: Refined cost relations of Program 16: Modular representation

Program 18

```
def List<Pair<A,A>> attach(A n,List<A> l) =
        case l {
          | Nil => Nil;
          | Cons(x,xs) =>  Cons(Pair(n, x),attach(n,xs));
        };
def List<Pair<A,A>> pairs(List<A> l)=
        case l {
          | Nil => Nil;
          | Cons(x,xs) =>  append(pairs(xs), attach(x,xs));
        };
```

Figure 10.2.: Program 18: Example that requires non-linear non-local size relations

In this CE, the cost of *append* depends on *xso* which is quadratic with respect to *xs* ($xso \leq xs^2$). Unfortunately, the chain summary of *pairs* cannot capture this non-linear relation.

CoFloCo can be extended to compute non-linear size relations using Max and Min variables and to use cost structures to represent the sizes of output variables in addition to costs.

10.3 Cost Structures with Binomial Coefficients

The Triangular Sum strategy (presented in Chapter 6) obtains precise bounds for certain kinds of programs (Program 2). This strategy is particularly interesting for lower bounds given that without it, the analysis would obtain only a linear bound for Program 2 instead of a quadratic bound[1]. However, the Triangular Sum strategy does not compute a precise lower bound if there are three or more nested loops of this kind. This is in fact what happens in two of the examples where PUBS-M obtains a better lower bound than CoFloCo in the experimental evaluation.

Example 10.3. Program 19 in Figure10.3 illustrates this situation. The lower bound complexity of that program is cubic $\Omega(n^3)$ but the current approach only obtains a quadratic lower bound complexity $\Omega(n^2)$. The Triangular Sum strategy can be applied to obtain a precise cost structure (for lower bounds) of the second loop (Chain $[(4)^+(3)]$):

$$\left\langle\ iv_1\ ,\ \left\{\begin{array}{ll} iv_1 \geq iv_2 - \frac{1}{2}iv_3 + \frac{1}{2}iv_4 & \\ iv_2 \geq iv_5 \cdot iv_4, & iv_3 \leq iv_4 \cdot iv_4 \end{array}\right\},\ \left\{\begin{array}{l} iv_5 \geq \|n-y\| \\ iv_4 = \|n-y\| \end{array}\right\}\right\rangle$$

However, when transforming this cost structure to obtain the cost of the phase $(2)^+$, the constraint $iv_2 \geq iv_5 \cdot iv_4$ generates the approximated constraint $smiv_2 \geq smiv_5 \cdot \lfloor iv \rfloor_4$ and $\lfloor iv \rfloor_4$ can take the value 1 in the last iteration. This loss of precision prevents CoFloCo from obtaining a cubic bound, even if the Triangular Sum strategy can be successfully applied to $smiv_5$. This comes from the fact that the non-linear expression $\|n-y\|^2$ is split into different constraints that are later treated independently.

A possible improvement is to extend cost structures with a new kind of final constraint that contains binomial coefficients of the form:

$$\binom{\|l\|}{k}$$

[1] For upper bounds the strategy obtains better precision at the constant level but it does not improve the asymptotic complexity.

Program 19	Refined cost relations
1 for(int x=0;x<n;x++) 2 for(int y=x;y<n;y++) 3 for(int z=y;z<n;z++) 4 tick(1);	1: $for_1(x,n) = \{x \geq n\}$ 2: $for_1(x,n) = \{y = x, x < n, x' = x+1\}, for_2[(4)^+3](y,n), for_1(x',n)$ 3: $for_2(y,n) = \{y \geq n\}$ 4: $for_2(y,n) = \{z = y, y < n, y' = y+1\}, for_3[(6)^+5](z,n), for_2(y',n)$ 5: $for_3(z,n) = \{z \geq n\}$ 6: $for_3(z,n) = \{z < n, z' = z+1\}, 1, for_3(z',n)$

Figure 10.3.: Program 19: Motivating example for binomial coefficients

where l is a linear expression in terms of the program variables and k is a positive integer. Binomial coefficients can be decomposed (or composed) using the formula

$$\binom{\|l\|+1}{k} = \binom{\|l\|}{k-1} + \binom{\|l\|}{k}$$

which exactly matches the behavior of the loops in Program 19. The Triangular Sum strategy could be extended to infer this kind of constraints[2]. Then, with this kind of constraints, the cost of $(4)^+(3)$ can be represented as

$$\left\langle\ iv_1\ ,\ \emptyset,\ \left\{\ iv_1 \geq \binom{\|n-y\|}{2}\ \right\}\right\rangle$$

And the cost of chain $(2)^+(1)$ is

$$\left\langle\ smiv_1\ ,\ \emptyset,\ \left\{\ smiv_1 \geq \binom{\|n-x\|}{3}\ \right\}\right\rangle$$

If the Triangular Sum strategy fails, a final constraint with a binomial coefficient can always be approximated to several regular final and non-final constraints to be processed by other strategies.

10.4 Scalability

An essential aspect of any static analysis that aspires to be widely applicable is scalability. In order to achieve it, the analysis needs to be compositional and efficient. Note that if the analysis is compositional and can process the functions of a program independently, it "only" has to be efficient enough to analyze the biggest function. This is a much more realistic objective than being able to analyze a complete program. Moreover, compositionality can facilitate the parallelization of the analysis.

The approach of CoFloCo is partly compositional. It follows a bottom-up approach and composes the results of each of the components following the structure of the call-graph. However, the analysis can suffer from performance issues and suffer a combinatorial explosion at several points.

First, the preprocessing of cost relations unfolds them to transform indirect recursion into direct recursion (see Chapter 4). In the case of loops, this can generate a CE per loop path and the number of loop paths can be exponential in the size of the program[3]. This can be partially alleviated by applying CE subsumption during the unfolding process. However, more aggressive simplifications (merging similar CEs) could be applied if the number of CEs grows very large. Additionally, it is important to slice the

[2] Note that using binomial coefficients to capture the cost of programs is not new. In fact, binomial coefficients are the building blocks of the resource polynomials used in RAML [HDW17].

[3] This is in fact what happens in the example where CoFloCo times out in the evaluation of challenging loop patterns (Section 7.2.3).

program in advance so the CR representation contains only variables that are likely to influence the cost. This can be done by checking which variables can influence the loop conditions. If much of the program behavior is abstracted away, many loop paths can have the same behavior with respect to the remaining variables and they can be represented by a single CE. The norm selection in Loopus [SZV17] has a similar effect and it is one of the factors that makes it so fast. On top of that, many operations on constraint sets (polyhedra) are, in the worst case, exponential on the number of variables so keeping that number low is essential for the performance of the analysis. The performance of the polyhedral operations can also be increased by applying the techniques in [SPV17].

Second, a combinational explosion can occur in the enumeration of the chains of a cost relation. Given a cost relation with n CEs, it is theoretically possible to have $\mathcal{O}(2^n)$ chains. However, this rarely happens in practice.

The third case where a combinational explosion can occur is in the cost equation specialization (Section 5.4). Given a CE that contains n calls to other CRs where each CR has m_i chains for $1 \leq i \leq n$, in the worst case the CE can be specialized into $\prod_{i=1}^{n} m_i$ CEs. The technique of chain compression (see Appendix A.2) partially alleviates this problem by reducing the number of chains (the m_i) that have to be considered. However, the number of specialized CEs can still grow very large even if the number of chains of each called CR is small. Consider a CE with n calls to other CRs where each of them has 2 chains. The number of specialized CE can be at most 2^n. A possible solution could involve applying folding transformations (the inverse operation of unfolding) to group calls together combined with a more aggressive chain compression.

10.5 Other Challenges: Pointers, Partial Failures and Contracts

In order for cost analysis to be effective on real programs, there are still multiple challenges that need to be addressed. As shown by the results of the real world evaluation (see Section 7.2.2), much work needs to be done in the abstraction of data structures and pointers and the integration of shape analyses. Loopus obtains much better results in that benchmark thanks (mainly) to its better treatment of data structures. The authors of [SZV17] also point out that with optimistic assumptions for pointers and data structures, they can obtain a bound for 1185 functions out of 1659 (instead of 806)[4]. In the case of functional programs, data structures tend to be easier to abstract, but there are other challenges such as higher-order functions and lazy evaluation.

In the majority of the existing cost analyses, once the analysis fails to obtain a bound of some part of the program, this failure is propagated to the rest of the analysis making the overall result useless. In order for an analysis to be useful in practice, it is important to be able to partially fail and still generate information that is useful. For example, there could be a scenario in which there is a program with a function `foo` whose cost cannot be successfully analyzed, but it can still be concluded that the function is called n times. In such a case, a bound of the form $n \cdot \mathit{Cost}(\texttt{foo})$ would still provide useful information about the behavior of the program. This could be combined with user annotations or contracts that enable users to specify the cost of program parts that fail to be analyzed or whose code is simply not available. Such user annotations could also help the tool analyze fragments of programs that require invariants that cannot be inferred automatically.

[4] In the replication of the experiments some examples were excluded and Loopus obtained 802 bounds for 1650.

Bibliography

[AAF+14] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.

[AAG+08] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[AAG+11] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Cost analysis of concurrent OO programs. In Hongseok Yang, editor, *Proceedings of Programming Languages and Systems - 9th Asian Symposium, APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[ABAG13] Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. Precise cost analysis via local reasoning. In Dang Van Hung and Mizuhito Ogawa, editors, *Proceedings of Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2013.

[ABG12] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In Antoine Miné and David Schmidt, editors, *Proceedings of Static Analysis - 19th International Symposium, SAS 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, September 2012.

[ABG+16] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. A formal verification framework for static analysis. *Software and Systems Modeling*, 15(4):987–1012, 2016.

[ACJRD15] Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. Parallel cost analysis of distributed systems. In Sandrine Blazy and Thomas Jensen, editors, *Proceedings of Static Analysis - 22nd International Symposium, SAS 2015*, volume 9291 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2015.

[ACMMRD14] Elvira Albert, Jesús Correas, Enrique Martín-Martín, and Guillermo Román-Díez. Static inference of transmission data sizes in distributed systems. In Tiziana Margaria and

Bernhard Steffen, editors, *Proceedings of 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISOLA 2014*, volume 8803 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2014.

[ACRD14] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Peak cost analysis of distributed systems. In Markus Müller-Olm and Helmut Seidl, editors, *Proceedings of Static Analysis - 21st International Symposium, SAS 2014*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014.

[ACRD15] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-cumulative resource analysis. In Christel Baier and Cesare Tinelli, editors, *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2015.

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *Proceedings of Static Analysis - 17th International Symposium, SAS 2010*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2010.

[AFG12] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Analysis of may-happen-in-parallel in concurrent objects. In Holger Giese and Grigore Rosu, editors, *Proceedings of Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012*, volume 7273 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2012.

[AFG15] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. May-happen-in-parallel analysis with condition synchronization. In Marko C. J. D. van Eekelen and Ugo Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis - 4th International Workshop, FOPARA 2015, Revised Selected Papers*, volume 9964 of *Lecture Notes in Computer Science*, pages 1–19, 2015.

[AFGM13] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Termination and cost analysis of loops with concurrent interleavings. In Dang Van Hung and Mizuhito Ogawa, editors, *Proceedings of Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013*, volume 8172 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2013.

[AFGM16] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. May-happen-in-parallel analysis for actor-based concurrency. *ACM Transactions on Computational Logic*, 17(2):11:1–11:39, 2016.

[AFGM17] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Rely-guarantee termination and cost analyses of loops with concurrent interleavings. *Journal of Automated Reasoning*, 59(1):47–85, 2017.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[AGG13] Elvira Albert, Samir Genaim, and Raúl Gutiérrez. A transformational approach to resource analysis with typed-norms. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2013.

[AGGZ13] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.

[AGM13] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, 2013.

[ALM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: higher-order meets first-order. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 152–164. ACM, 2015.

[AM16] Martin Avanzini and Georg Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016.

[AMS16] Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean Complexity Tool. In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016*, volume 9636 of *Lecture Notes in Computer Science*, pages 407–423. Springer, 2016.

[AST15] Martin Avanzini, Christian Sternagel, and René Thiemann. Certification of complexity proofs using CeTA. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23–39. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[BAG14] Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, 2014.

[BCF13] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 413–429. Springer, 2013.

[BCI+16] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In Marsha Chechik and Jean-François Raskin, editors, *Proceeding of Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016*, volume 9636 of *Lecture Notes in Computer Science*, pages 387–393. Springer, 2016.

[BEF+14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[BEF+16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38(4):13:1–13:50, 2016.

[Ben01] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[BK96] Florence Benoy and Andy King. Inferring argument size relationships with CLP(R). In John P. Gallagher, editor, *Proceedings of Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 1996.

[BPZZ05] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. PURRS: towards computer algebra support for fully automatic worst-case complexity analysis. *CoRR*, abs/cs/0512056, 2005.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages, POPL 1978*, pages 84–96. ACM Press, 1978.

[CHK+15] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment abstraction for worst-case execution time analysis. In Jan Vitek, editor, *Proceedings of Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *Lecture Notes in Computer Science*, pages 105–131. Springer, 2015.

[CHRS14] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 270–281. ACM, 2014.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*, pages 467–478. ACM, 2015.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In Mark N. Wegman and Thomas W. Reps, editors, *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00*, pages 184–198. ACM, 2000.

[DH17] Ankush Das and Jan Hoffmann. ML for ML: learning cost semantics by experiment. In Axel Legay and Tiziana Margaria, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 190–207. Springer, 2017.

[DL93] Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[DLH90] Saumya K. Debray, Nai-Wei Lin, and Manuel Hermenegildo. Task granularity analysis in logic programs. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN 1990*

Conference on Programming Language Design and Implementation, PLDI 1990, pages 174–188. ACM, 1990.

[DLHL94] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Estimating the computational cost of logic programs. In *Proceedings of Static Analysis: First International Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 255–265. Springer, 1994.

[DLHL97] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In Jan Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium*, pages 291–305. MIT Press, 1997.

[FAG13] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In Dirk Beyer and Michele Boreale, editors, *Proceedings of Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2013.

[FG10] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with Aspic and C2fsm. *Electronic Notes in Theoretical Computer Science*, 267(2):3 – 13, 2010.

[FG17] Florian Frohn and Jürgen Giesl. Complexity analysis for java with AProVE. In *Proceedings of the 13th International Conference on integrated Formal Methods, iFM '17*, Lecture Notes in Computer Science. Springer, 2017. To appear.

[FH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems, APLAS 2014*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[FKS11] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41–50. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

[Flo16] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *Proceedings of Formal Methods - 21st International Symposium, FM 2016*, volume 9995 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2016.

[FNH+16] Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. Lower runtime bounds for integer programs. In Nicola Olivetti and Ashish Tiwari, editors, *Proceedings of Automated Reasoning: 8th International Joint Conference. IJCAR 2016*, volume 9706 of *Lecture Notes in Computer Science*, pages 550–567. Springer, 2016.

[FPR09] Paola Festa, Panos M. Pardalos, and Mauricio G. C. Resende. Feedback set problems. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1005–1016. Springer, 2009.

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and

complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[GG08] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and Max operator with application in timing analysis. In Aarti Gupta and Sharad Malik, editors, *Proceedings of Computer Aided Verification, 20th International Conference, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[GGP+15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for LLVM IR programs. In Henk Corporaal and Sander Stuijk, editors, *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2015*, pages 12–21. ACM, 2015.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 375–385. ACM, 2009.

[GLL15] Abel Garcia, Cosimo Laneve, and Michael Lienhardt. Static analysis of cloud elasticity. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, PPDP 2015*, pages 125–136. ACM, 2015.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 127–139. ACM, 2009.

[Gro01] Bernd Grobauer. Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming. ICFP 2001*, pages 253–264. ACM, 2001.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pages 292–304. ACM, 2010.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012.

[HDW17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 359–373. ACM, 2017.

[HGFS17] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. Termination analysis for programs with bitvector arithmetic by symbolic execution. Technical report, LuFG Informatik 2, RWTH Aachen University, Germany, 2017.

[HH10a] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In Kazunori Ueda, editor, *Proceedings of Programming Languages and Systems: 8th Asian Symposium, APLAS 2010*, volume 6461 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2010.

[HH10b] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Proceedings of Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[HHP14] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination analysis by learning terminating programs. In Armin Biere and Roderick Bloem, editors, *Proceeding of Computer Aided Verification - 26th International Conference, CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.

[HM08] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, IJCAR '08*, volume 5195 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2008.

[HM14] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In Gilles Dowek, editor, *Proceedings of Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014*, volume 8560 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2014.

[HM15] Martin Hofmann and Georg Moser. Multivariate amortised resource analysis for term rewrite systems. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 241–256. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[HS15] Jan Hoffmann and Zhong Shao. Type-based amortized resource analysis with integers and arrays. *Journal of Functional Programming*, 25, 2015.

[JHS+11] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proceedings of 9th International Symposium on Formal Methods for Components and Objects, FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[KKZ11] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund M. Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011.

[KSZM09] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In Ralf Treinen, editor, *Proceedings of Rewriting Techniques and Applications: 20th International Conference, RTA 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LH15] Jan Leike and Matthias Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1), 2015.

[LORR13] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using Max-SMT. In *Proceedings of Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 218–225. IEEE, 2013.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[NEG13] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.

[NFB+17] Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl. Complexity analysis for term rewriting by integer transition systems. In *Proceedings of the 11th International Symposium on Frontiers of Combining Systems, FroCoS 2017*, Lecture Notes in Artificial Intelligence. Springer, 2017. To appear.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[Sha79] Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

[SLH14] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming, TPLP*, 14(4-5):739–754, 2014.

[SPV17] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 46–59. ACM, 2017.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Armin Biere and Roderick Bloem, editors, *Proceedings of Computer Aided Verification - 26th International Conference, CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761. Springer, 2014.

[SZV15] Moritz Sinn, Florian Zuleger, and Helmut Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In Roope Kaivola and Thomas Wahl, editors, *Proceedings of Formal Methods in Computer-Aided Design, FMCAD 2015*, pages 144–151. IEEE, 2015.

[SZV17] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[UM14] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In Markus Müller-Olm and Helmut Seidl, editors, *Proceedings of Static Analysis - 21st International Symposium, SAS 2014*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.

[Vas08] Pedro B. Vasconcelos. *Space cost analysis using sized types*. PhD thesis, School of Computer Science, University of St Andrews, UK, November 2008.

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Phil Trinder, Greg J. Michaelson, and Ricardo Peña, editors, *Proceedings of the 15th International Conference on Implementation of Functional Languages, IFL' 03*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2004.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008.

[Weg75] Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

[Wol03] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, 5 edition, 2003.

[YK97] Ting Yu and Owen Kaser. A note on "on the conversion of indirect to direct recursion". *ACM Transactions on Programming Languages and Systems*, 19(6):1085–1087, 1997.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *Proceedings of Static Analysis - 18th International Symposium, SAS 2011*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2011.

A CoFloCo Implementation Details

This chapter contains a quick reference of the options that CoFloCo supports at the moment and some implementation details that have a significant effect on the efficiency of the tool.

A.1 CoFloCo Quick Reference

-h, -help Display help information.

-i, -input *filename*: Select the input file that contains the cost relation system.

-s, -stats Show some time statistics.

-debug (default no) Show debug information.

-v, -verbosity *0-3* : The level of verbosity.

-incremental (default no) The usual analysis performs the refinement of the complete cost relation system first and all the bound computation later. With this option, the refinement and the bound computation are done bottom-up one cost relation at a time.

-n_candidates *nat* : (default 1) Set the maximum number of candidates considered in the strategies.

-compute_ubs (default yes) Obtain closed-form upper bounds.

-compute_lbs (default yes) Obtain closed-form lower bounds (if disabled, additional simplifications can be made on cost structures).

-conditional_ubs (default no) Generate piece-wise upper bounds (as described in Section 6.9).

-conditional_lbs (default no) Generate piece-wise lower bounds (as described in Section 6.9).

-solve_fast (default no) All constraints of the form $\sum_{n=1}^{n} iv_i \leq e$ in cost structures are split into simple constraints $iv_i \leq e$ for $1 \leq i \leq n$. This loses precision at the constant level but not at the asymptotic level. On the other hand, this enables further simplifications on cost structures as detailed in Section A.3 and simplifies their solution (Section 6.8).

-compress_chains *0-2*: (default 0) This option activates chain compression as described in Section A.2. The parameter indicates the criterion to compress chains.

-only_termination Perform only the refinement but not the bound computation.

A.2 Refinement

This section contains some technical details about the implementation of the refinement procedure which have a significant effect on its efficiency and scalability.

Lazy Call-graph Computation

In order to compute the call-graph of a CR C, CoFloCo has to check for every $c, d \in C$ whether a CE c can call d. This check corresponds to checking the satisfiability of a constraint set (several if c has multiple recursive calls). This process can be costly, in a cost relation with n cost equations, $O(n^2)$ checks have to be performed to compute the complete call-graph.

However, in many cases we do not need to compute all the edges of the call-graph. Consider a CR with CEs $1, 2, \cdots, 10$ where 10 is the only base case, $1, 2, \cdots, 9$ are recursive CEs such that $i \to j$ for every $i \in [1..j]$ and $j \in [1..j]$. The chains of this CR are $(1 \vee 2 \vee \cdots \vee 9)^{\omega}$, $(1 \vee 2 \vee \cdots \vee 9)^{+}10$, and 10. In order to compute the complete call-graph $9 \cdot 10 = 90$ checks have to be performed but the given phases and chains can be obtained by only checking that $i \to i+1$ for $i \in [1..9]$ and $9 \to 1$ (in total 10 checks). In other words, it is enough to find one cycle that includes all CEs $1, \cdots, 9$ to conclude that $\{1, \cdots, 9\}$ is a strongly connected component.

Therefore, the edges of the call-graph are computed lazily as the algorithm checks for cycles. In the worst case, all the checks have to be performed but on average the number of checks is significantly lower.

Chain Compression

Cost equation specialization (see Section 5.4) can give rise to a combinatorial explosion. Let $c\colon C'(\boldsymbol{x} : \boldsymbol{y}) = \varphi_1, b_1, \cdots, \varphi_i, C(\boldsymbol{x}_i : \boldsymbol{y}_i), \cdots, b_n, \varphi_n$ be a cost relation that has to be specialized and let $ch_1, ch_2, \cdots, ch_n$ be the chains for C. The CE specialization generates n copies of c, each one with a call to $C[ch_i]$. If c has other calls that are specialized, each of the copies gets specialized and the number of CEs can grow very large.

This can be partly alleviated by grouping "similar" chains together. If for instance, chains are grouped in two sets $ch_1, ch_2, \cdots, ch_i$ and $ch_{i+1}, ch_2, \cdots, ch_n$, CE specialization will generate only 2 copies of C with calls to $C[ch_1 \vee ch_2 \vee \cdots \vee ch_i]$ and $C[ch_{i+1} \vee ch_2 \vee \cdots \vee ch_n]$ where the summary of a group is the convex hull of the summaries of its chains $summary(ch_1 \vee ch_2 \vee \cdots \vee ch_i) = \bigsqcup_{j=1}^{m} summary(ch_j)$. This technique is called *chain compression* and it reduces the number of specialized cost equations at the cost of possibly losing some precision in the summary. The only requirement is that terminating and non-terminating chains are not grouped together (in order to maintain Claim 5.1).

The similarity among chains can be defined by considering their summary. Different criteria can be adopted to strike a balance between efficiency and precision. The two criteria currently implemented are the following. Two chains are grouped together ($ch_1 \equiv ch_2$) if

1. $summary(ch_1) = summary(ch_2)$
2. $(summary(ch_1) \Rightarrow summary(ch_2)) \vee (summary(ch_2) \Rightarrow summary(ch_1))$

The first criterion does not suppose any loss in precision neither in the refinement or the bound computation. The second criterion is more agressive, that is, it will group more chains together and it might affect the precision of the latter bound computation.

Chain compression can be activated using the option `-chain_compression n` where n corresponds to the criterion applied.

A.3 Cost Structure Maintenance and Simplification

An important part of the bound computation algorithm is to represent and maintain cost structures efficiently. This involves several simplifications.

Joining Final Constraints

The operations over final constraints have a higher cost than the ones over non-final constraints (which are mainly syntactic). For this reason, the efficiency of the analysis can be increased by merging similar final constraints with the help of additional non-final constraints.

Let $\sum \boldsymbol{iv} \bowtie \|l\|$ and $\sum \boldsymbol{iv}' \leq \|l\|$ be two final constraints, they can be substituted by a final constraint $iv \leq \|l\|$ and two non-final constraints $\sum \boldsymbol{iv} \bowtie iv$ and $\sum \boldsymbol{iv}' \bowtie iv$ where iv is a fresh intermediate variable. More generally, let $\sum \boldsymbol{iv} \bowtie \|l\|$ and $\sum \boldsymbol{iv}' \leq \|l'\|$ such that $l' = k \cdot l$ for $k \in \mathbb{Q}^+$, they can be substituted by $iv \leq \|l\|$ and two non-final constraints $\sum \boldsymbol{iv} \bowtie iv$ and $\sum \boldsymbol{iv}' \bowtie k \cdot iv$.

Discarding Undefined and Unused Constraints

As mentioned at the end of Section 6.3, a dependency relation $\preceq$ can be established between constraints based on their used and defined intermediate variables. This dependency relation induces a partial order over the constraints of a cost structure which are always maintained topologically sorted. In the current implementation, dependency cycles are not allowed and when one is detected (because a new constraint has been added) it is resolved by merging or discarding some constraints.

Given a cost structure where the non-final constraints are topologically sorted, it is easy to detect constraints that can be removed. If a constraint uses variables that are not defined elsewhere, it can be discarded or relaxed. For example, if iv_2 is not defined anywhere, the constraint $iv \leq iv_1 - iv_2$ can be relaxed to $iv \leq iv_1$. If a constraint defines variables that are never used, it can also be discarded.

Discarding Redundant Constraints

Redundant constraints can also be detected and simplified. Let $ic = \sum \boldsymbol{iv} \leq SE$ and $ic' = \sum \boldsymbol{iv}' \leq SE$, the constraint ic is redundant if $defines(ic) \subseteq defines(ic')$. Conversely, let $ic = \sum \boldsymbol{iv} \geq SE$ and $ic' = \sum \boldsymbol{iv}' \geq SE$, the constraint ic is redundant if $defines(ic) \supseteq defines(ic')$.

Substituting Intermediate Variables and Factoring out

A constraint is *simple* if it has a single intermediate variable on the left side, that is, it has the form $iv \bowtie SE$. If an intermediate variable is defined in only one simple constraint, we can substitute its positive uses in constraints of the same operator ($\bowtie$) and its negative uses in constraints of the opposite operator ($!\bowtie$). For instance, the constraints $iv_1 \leq 2iv_2 + 1, iv_1 \geq iv_3 - iv_2$ and $iv_2 \leq iv_4 + 2$ can be transformed (as long as iv_2 is not defined in any other constraint) into $iv_1 \leq 2iv_4 + 5$ and $iv_1 \geq iv_3 - iv_4 - 2$. In the case of non-linear constraints, this can give us the opportunity to factor out components and reduce the number of constraints. For example, the constraints $iv_1 \leq iv_2 + iv_3$, $iv_2 \leq iv_4 \cdot iv_5$, and $iv_3 \leq iv_6 \cdot iv_5$, can be transformed to $iv_1 \leq iv_7 \cdot iv_5$, $iv_7 \leq iv_4 + iv_6$ where iv_7 is a fresh intermediate variable.

Merging Cost Structures for Chain Compression

In Section A.2, an optimization called *chain compression* was presented. This optimization reduces the number of refined CEs and chains generated during the refinement phase. It does so by generating a CE that calls a disjunction of chains $[ch_1 \vee ch_2 \vee \cdots \vee ch_n]$ instead of generating n CEs that call each single chain. In order to apply such optimization, we have to define the cost structure of a disjunction of chains $[ch_1 \vee ch_2 \vee \cdots \vee ch_n]$ and how to obtain it from the cost structures of each of the chains.

We define this operation for two chains but it can be easily generalized to any number of chains. Let $\langle E_1, IC_1, FC_1(\boldsymbol{x})\rangle$ and $\langle E_2, IC_2, FC_2(\boldsymbol{x})\rangle$ be cost structures of ch_1 and ch_2. The cost structure for $[ch_1 \vee ch_2]$ is

$$\left\langle iv, \quad \left\{ \begin{array}{ll} iv \leq \max(iv_1, iv_2), & iv_1 = E_1, \\ iv \geq \min(iv_1, iv_2), & iv_2 = E_2, \end{array} \right\} \cup IC_1 \cup IC_2, \quad FC_1(\boldsymbol{x}) \cup FC_2(\boldsymbol{x}) \right\rangle$$

where iv, iv_1 and iv_2 are fresh intermediate variables. This cost structure is precise but it generates constraints with max and min operators whose later treatment is not very precise (see Section 6.5.1). In many cases though, the resulting cost structures can be simplified applying the techniques described above.

Example A.1. Consider for instance the composition of the following cost structures $\langle iv_3 + 2, \emptyset, \{iv_3 \leq \|x\|\}\rangle$ and $\langle iv_4, \emptyset, \{iv_4 \leq \|x\|\}\rangle$. The merged cost structure is:

$$\Big\langle iv, \quad \{iv \leq \max(iv_1, iv_2), iv \geq \min(iv_1, iv_2), iv_1 \leq iv_3 + 2, iv_2 \leq iv_4\}, \quad \{iv_3 \leq \|x\|, iv_4 \leq \|x\|\}\Big\rangle$$

This cost structure can be simplified as follows:

- The lower bounds of iv_1 and iv_2 are not defined, so $iv \geq \min(iv_1, iv_2)$ can be discarded.
- The final constraints $iv_3 \leq \|x\|$ and $iv_4 \leq \|x\|$ can be joined generating $iv_5 \leq \|x\|$ and $iv_3 \leq iv_5$ and $iv_4 \leq iv_5$.
- Because iv_3 and iv_4 are only defined by one constraint, we can substitute their uses by their definitions and obtain $iv_1 \leq iv_5 + 2$ and $iv_2 \leq iv_5$.
- Finally, we can factor out $iv \leq \max(iv_5 + 2, iv_5) = iv_5 + \max(2, 0) = iv_5 + 2$
- The final cost structure is: $\langle iv, \{iv \leq iv_5 + 2\}, \{iv_5 \leq \|x\|\}\rangle$

Fast merging of Cost Structures

In many cases, we are only interested in the asymptotic complexity of the upper bounds (we do not want to obtain lower bounds). Then, we can simply add the cost structures. Let $\langle E_1, IC_1, FC_1(\boldsymbol{x})\rangle$ and $\langle E_2, IC_2, FC_2(\boldsymbol{x})\rangle$ be cost structures of ch_1 and ch_2. A valid over-approximation of the cost structure for $[ch_1 \vee ch_2]$ is

$$\langle E_1 + E_2, IC_1 \cup IC_2, FC_1(\boldsymbol{x}) \cup FC_2(\boldsymbol{x})\rangle$$

The resulting cost expression is easier to simplify, because it does not have extra constraints with max and min operators but it represents an over-approximation of the cost. This operation is only used if the option `compute_lbs` is deactivated and `solve_fast` is active.