UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA



TESIS DOCTORAL

Análisis de terminación de programas con controles de flujo complejos

Termination analysis of programs with complex control-flow

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Jesús Javier Doménech Arellano

Director

Samir Genaim

Madrid

© Jesús Javier Doménech Arellano, 2021

Termination Analysis of Programs with Complex Control-Flow

Análisis de Terminación de Programas con Controles de Flujo Complejos



TESIS DOCTORAL

Jesús Javier Doménech Arellano

Director: **Samir Genaim**

Facultad de Informática
Universidad Complutense de Madrid

Madrid, octubre de 2020

Análisis de Terminación de Programas con Controles de Flujo Complejos



TESIS DOCTORAL

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por

Jesús Javier Doménech Arellano

Dirigida por el profesor:

Samir Genaim

Facultad de Informática
Universidad Complutense de Madrid

Madrid, octubre de 2020

Termination Analysis of Programs with Complex Control-Flow



Ph.D. Thesis Dissertation

Dissertation presented to obtain the degree of
Ph.D. in Computer Science by
Jesús Javier Doménech Arellano

Supervised by:
Samir Genaim

Facultad de Informática
Universidad Complutense de Madrid

Madrid, October 2020

Financial Support. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services, by the EU project RTI2018-094403-B-C31, by the Spanish MICINN/FEDER, by the MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, by the CM projects S2013/ICE-3006 and S2018/TCS-4314, and by the pre-doctoral UCM CT27/16-CT28/16 grant.

Resumen

El problema de la terminación de un programa es fundamental en la informática y ha sido objeto de estudio de numerosas investigaciones. La técnica mejor conocida, y más frecuentemente utilizada, para demostrar terminación es la del uso de funciones de clasificación (*ranking functions*). Estas funciones relacionan los estados del programa con los elementos de un conjunto ordenado bien-fundado, tal que el valor desciende en estados consecutivos del programa. Como descender en un conjunto ordenado bien-fundado no se puede hacer de manera infinita se demuestra la terminación del programa. En esta tesis, abordamos el problema de terminación para Sistemas de Transiciones (*Transition Systems*) con valores numéricos, que son una representación de programas muy comúnmente utilizada en los análisis de programas. Los Sistemas de Transiciones están definidos por Grafos de Control de Flujo (*Control-Flow Graphs*) donde las aristas están anotadas con fórmulas describiendo las transiciones que hay entre los nodos correspondientes.

A diferencia de la terminación de programas en general, que es indecidible, los problemas algorítmicos de detección y generación de funciones de clasificación pueden ser resueltos dadas ciertas elecciones de la representación del programa y la clase de funciones de clasificación. Numerosos investigadores han propuesto dichas clases; en algunos casos, los problemas algorítmicos han sido completamente resueltos, y se provee de algoritmos eficientes, mientras que en otros casos permanecen todavía como problemas abiertos. Además de demostrar terminación, algunas clases de funciones de clasificación también sirven para determinar la cota de la longitud de la ejecución, muy útil en aplicaciones como el análisis de coste (*cost analysis*).

Debido a consideraciones prácticas, las herramientas de análisis de terminación suelen centrarse en clases de funciones de clasificación que puedan ser sintetizadas eficientemente. Normalmente, el análisis comienza con una clase simple y, si falla, se usan clases más ricas y, posiblemente, más caras en ejecución. Aún así, entender los aspectos teóricos de estas clases es importante incluso en la práctica porque comprender los límites y las propiedades de los problemas subyacentes ayuda en el diseño de mejores algoritmos.

El tema de esta tesis se basa en el campo de la demostración de terminación de sistemas de transiciones numéricos usando funciones de clasificación, en concreto, sistemas de transiciones con *controles de flujo complejos* y funciones de clasificación que son lineales o tuplas de funciones lineales. Los principales objetivos son: (i) el estudio de propiedades de las clases de funciones de clasificación que no han recibido todavía suficiente atención y, así, promover su uso para mejorar la precisión de los análisis de terminación en la práctica; y (ii) desarrollar otras técnicas para mejorar la precisión de los análisis de terminación, en particular, usando transformaciones de programas.

Las funciones de clasificación multi-fase (*Multiphase ranking functions*) son una clase importante que no se ha estudiado lo suficiente en la literatura, y en esta tesis, nuestro objetivo es estudiarlas. Este tipo de funciones de clasificación es usado para demostrar la terminación de programas en los que la ejecución progresa por medio de un número de fases. Estas funciones son tuplas de funciones lineales $\langle \rho_1, \ldots, \rho_d \rangle$ donde ρ_i decrece durante la fase i-ésima. Nuestro trabajo proporciona nuevos conocimientos importantes sobre las funciones de clasificación multi-fase para bucles definidos por la conjunción de restricciones lineales, que son un caso especial de los sistemas de transiciones. Concretamente, para el problema de decisión de la *existencia de una función de clasificación multi-fase* que busca determinar si dado un bucle, este admite una o no; y el correspondiente problema *acotado* de decisión que restringe la búsqueda a funciones de clasificación

multi-fase con una profundidad d, donde d es parte de la entrada. La decibilidad y complejidad del problema cuando está restringido a d como parámetro de entrada han sido resueltas, mientras que, en esta tesis, progresamos respecto al problema de existencia sin una cota de profundidad dada.

Nuestro nuevo enfoque, aunque no llega a ser un procedimiento de decisión para el caso general, revela información importante sobre la estructura de estas funciones. Curiosamente, relaciona el problema de buscar una función de clasificación multi-fase con el problema de encontrar conjuntos de recurrencia (usados para demostrar no terminación). También, ayuda a identificar las clases de bucles para los que las funciones de clasificación multi-fase son suficientes, y así tener cotas en tiempo de ejecución lineales. Para el problema de existencia acotado por profundidad, obtenemos un nuevo método de tiempo polinómico que puede proporcionar también pruebas para respuestas negativas. Para obtener este método, introducimos una nueva representación para los bucles que otorga nueva información importante sobre las funciones de clasificación multi-fase y la terminación de los bucles en general, información que sería difícil de ver usando las representaciones estándar.

Otro enfoque habitual para mejorar la precisión en análisis de programas en general, no solo en terminación, está basado en el uso de transformadores de programas para simplificar el flujo de control, lo que también se conoce como técnicas de Refinamiento de Controles de Flujo (*Contro-Flow Refinement*). Varias de estas técnicas han sido sugeridas para diferentes modelos de programación, pero están hechas a medida para análisis concretos. En esta tesis, sugerimos el uso de técnicas de transformación de programas de uso general para el refinamiento de controles de flujo, en particular sugerimos el uso de Evaluación Parcial (*Partial Evaluation*).

Usar evaluación parcial para el refinamiento de los controles de flujo tiene la clara ventaja de que la validez de la transformación se obtiene de las propiedades generales de la evaluación parcial. Usamos un algoritmo de evaluación parcial que incorpora abstracción basada en propiedades, y vemos cómo la correcta elección de propiedades nos permite demostrar terminación e inferir el coste de programas complicados que no pueden ser tratados por las herramientas del estado-del-arte. Aportamos una (profunda) integración y evaluación de esta técnica en un analizador de terminación, y la usamos como un paso de pre-proceso para varios análisis de coste. Además, proveemos de una implementación independiente que puede ser usada con poco esfuerzo para añadir refinamiento del control de flujo a herramientas de análisis de programas existentes.

Aportamos también iRankFinder, un analizador de terminación que implementa, entre otras cosas, todas las técnicas desarrolladas en esta tesis. Así como, una evaluación experimental de iRankFinder que demuestra la utilidad de las técnicas. iRankFinder puede ser usado por medio de la línea de comandos o vía interfaz web. Sin embargo, en lugar de construir una interfaz web hecha a medida para iRankFinder, en esta tesis, hemos desarrollado un conjunto de herramientas de código abierto, llamado EasyInterface, que simplifica el proceso de construir interfaces de usuario para prototipos de herramientas de investigación, y así mejorar la difusión de la correspondiente investigación.

Palabras Clave: Análisis de Terminación y de No terminación. Funciones de Clasificación. Conjuntos Recurrentes. Refinamiento de Controles de Flujo. Evaluación Parcial.

Abstract

The problem of program termination is fundamental in Computer Science and has been the subject of voluminous research. The best known, and often used technique for proving termination is that of *ranking functions*. These are functions that map the program states to the elements of a well-founded ordered set, such that the value descends on consecutive program states. Since descent in a well-founded set cannot be infinite, this proves termination. In this thesis, we address the termination problem for *Transition Systems* with numerical variables, which is a very common program representation that is often used in program analysis. They are defined by *Control-Flow Graphs* where edges are annotated with formulas describing transitions between corresponding nodes.

Unlike termination of programs in general, which is undecidable, the algorithmic problems of detection or generation of a ranking function can well be solvable, given certain choices of the program representation, and the class of ranking functions. Numerous researchers have proposed such classes; in some cases, the algorithmic problems have been completely settled, and efficient algorithms provided, while other cases remain as open problems. Besides proving termination, some classes of ranking functions also serve to bound the length of the computation, useful in applications such as *cost analysis*.

Due to practical considerations, termination analysis tools typically focus on classes of ranking functions that can be synthesised efficiently. Typically, the analysis starts with a simple class of ranking functions, and on failure, other richer classes, and possibly more expensive, are used. Understanding the theoretical aspects of such classes is still important even in practice because understanding the limits and properties of the underlying problems helps in designing better algorithms.

The topic of this thesis lies in the field of proving termination of numerical transition systems using ranking functions, in particular, transition systems with *complex control-flow* and ranking functions that are linear or tuples of linear functions. The main goals are: (i) study properties of classes of ranking functions that have not received enough attention yet, and thus promote their use for improving the precision of termination analysis in practice; and (ii) develop other techniques for improving the precision of termination analysis, in particular using program transformations.

Multiphase ranking functions are important ranking functions that have not been studied enough in the literature, and in this thesis, we aim to study this class. This kind of ranking functions is used to prove termination of programs in which the computation progresses through a number of phases. They consist of linear functions $\langle \rho_1, \ldots, \rho_d \rangle$ where ρ_i decreases during the ith phase. Our work provides new important insights regarding multiphase ranking functions for loops described by a conjunction of linear constraints, which are a special case of transitions systems. In particular, for the decision problem *existence of a multiphase ranking function* which asks to determine whether a given loop admits a multiphase ranking function; and the corresponding *bounded* decision problem that restricts the search to multiphase ranking functions of depth d, where d is part of the input. The decidability and complexity of the problem when d is restricted by an input parameter have been settled, while in this thesis, we make progress regarding the existence problem without a given depth bound.

Our new approach, while falling short of a decision procedure for the general case, reveals some important insights into the structure of these functions. Interestingly, it relates the problem of seeking multiphase ranking functions to that of seeking recurrent sets (used to prove non-termination). It also helps in identifying classes of loops for which

multiphase ranking functions are sufficient, and thus have linear runtime bounds. For the depth-bounded existence problem, we obtain a new polynomial-time procedure that can provide *witnesses* for negative answers as well. To obtain this procedure, we introduce a new representation for loops, which yields new important insights on multiphase ranking functions and termination loops in general, that are very difficult to see when using the standard representation.

Another common approach for increasing the precision in program analysis in general, not only termination, is based on using program transformations to simplify the control-flow, which is also known as *Control-Flow Refinement*. Several such techniques have been suggested for different programming models, but they are typically tailored for a particular analysis. In this thesis, we suggest the use of general-purpose program transformation techniques for control-flow refinement, in particular partial evaluation.

Using partial evaluation for control-flow refinement has a clear advantage over other approaches in that soundness follows from the general properties of partial evaluation. We use a partial evaluation algorithm incorporating property-based abstraction, and show how the right choice of properties allows us to prove termination and to infer the complexity of challenging programs that cannot be handled by state-of-the-art tools. We report on the (deep) integration and evaluation of the technique in a termination analyser, and its use as a pre-processing step for several cost analysis. We also provide a stand-alone implementation that can be used with little effort to add control-flow refinement to existing program analysis tools.

We also report on iRankFinder, a termination analyser that implements, among other things, all techniques developed in this thesis. We report on an experimental evaluation of iRankFinder that demonstrates the usefulness of the techniques developed in this thesis. iRankFinder can be used from a command-line or via a web-interface. However, instead of building a web-interface that is tailored for iRankFinder, in this thesis, we have developed an open-source toolkit, called EasyInterface, that simplifies the process of building GUIs for research prototypes tools, and thus improve the dissemination of the corresponding research.

Keywords: Termination and Non-termination Analysis. Ranking Functions. Recurrent Sets. Control-Flow Refinement. Partial Evaluation.

Agradecimientos

Esta tesis no habría sido posible sin la compañía y colaboración de muchas personas que han ido aportando granitos de arena e incluso grandes ladrillos en la edificación de este trabajo.

Por eso, no puedo nombrar primero a otro que mi director Samir Genaim que ha llevado sus funciones más allá del deber, ha sido paciente, accesible, dispuesto a gastar horas y a no perder un segundo, ha sido compañero y guía desde el principio. Simplemente gracias.

En segundo lugar, quiero agradecer todo el tiempo y dedicación otorgados por Elvira Albert, la Investigadora Principal del grupo COSTA que decidió apostar por un estudiante de cuarto año y enseñarle los entresijos de la investigación en la universidad. Junto a ella, quiero agradecer al resto de miembros del grupo que siempre que han tenido oportunidad han aportado su granito de arena, ya sea preparando transparencias, explicando los conceptos más básicos o tomando un café.

Quiero agradecer a John P. Gallagher, que me acogiera durante tres meses, y a Amir M. Ben-Amram, dos investigadores con los que he tenido el honor de poder trabajar y publicar.

El Aula 16 ese espacio donde tan pronto eramos cinco que cincuenta, ahí he descubierto a verdaderos compañeros entre los que querría mencionar especialmente a Marta Caro, Cristina Alonso, Antonio Calvo y Joaquín Gayoso. Ellos han sido una constante en el día a día de estos cuatro años. Gracias también a Alicia Merayo, no ha sido una ni dos peleas con su trabajo y el mío las que hemos batallado. Especialmente gracias a Pablo Gordillo y Miguel Isabel compañeros de todo, carrera, máster y esta etapa que vamos cerrando. Juntos nos hemos enfrentado a todo: clases, trabajo, papeleo imposible, copas y cafés, sin vosotros esto habría sido más difícil. Quiero mencionar aparte a Luisma Costero que, siendo compañero como el resto, siempre ha sido más testigo de mi boda, cervezas interminables, planes de cabras y montañas, un amigo.

Para el final dejo a quienes lo son todo, mis padres, que me han dado todo y me han convertido en quien soy, y hermanos, los incansables, siempre juntos los que me aguantaban día a día y me acompañaron cuando formé mi propia familia. Ana, esposa, gracias por ser tú, por estar ahí, por quererme y aguantarme, por decirme SI, por iniciar una aventura mayor conmigo, por ser mi esposa y madre de nuestros hijos, por tu paciencia en las largas noches y cortos días.

A Quien lo baña todo y a todos ellos, gracias por hacer de esta tesis una experiencia inolvidable.

Acknowledgments

This thesis would not have been possible without the collaboration of many people who have supported me along these years.

I cannot name first other than my advisor Samir Genaim who has gone above and beyond the duty, he has been patient, accessible, willing to spend hours but not to waste a second. He has been a colleague and a guide from the beginning.

Secondly, I would like to thank all the time and dedication given by Elvira Albert, the IP of the research COSTA group. She decided to bet on a fourth-year student and teach him the ins and outs of the research at the university. I would also want to thank the rest of the members of the group who have contributed with their expertise, whenever they have had the opportunity, either by preparing slides, explaining the most basic concepts or having a coffee.

I am also grateful to John P. Gallagher, who hosted me for three months, and Amir M. Ben-Amram, two researchers with whom I have had the honor of being able to work and publish results.

Mention apart requires Aula 16, that space where I have discovered trustworthy colleagues, among whom I would especially like to mention Marta Caro, Cristina Alonso, Antonio Calvo and Joaquin Gayoso. They have been a constant in the day a day of these four years. Thanks also to Alicia Merayo, it has not been one or two fights against our works that we have battled. Thanks to Pablo Gordillo and Miguel Isabel colleagues from everything degree, master, and this stage that we are closing now. Together we have faced everything: classes, work, impossible paperwork, drinks and coffees, without you, this would have been really difficult.

Thanks to Luisma Costero who, being a colleague in the studies, has always been more. He was a witness of my wedding, the pair of endless beers, the crazy dreamer planning to leave all and go to the mountains to care goats, a friend.

For the end, I have left those who are everything, my parents, who have given me all and have made me who I am, and my siblings, the tireless and always together those who supported me day by day. Ana, my wife, thank you for being you, for being there, for loving me and taking me up, for saying YES and starting the most extraordinary adventure building out own family, thank you for being my wife and mother of our children, for your patience in the long nights and the short days, for your love.

Who permeates everything and all of them, thank you for making this thesis and unforgettable experience.

Contents

Resumen i

Abstract iii

Contents vii

List of Figures x

List of Tables xi

1 Introduction 1
- 1.1 Classes of (Linear-based) Ranking Functions . . . 2
- 1.2 Control-Flow Refinement . . . 4
- 1.3 Objectives and Methodology . . . 4
- 1.4 Summary of Contributions and Outline . . . 5

2 Preliminaries 9
- 2.1 Polyhedra . . . 9
- 2.2 Program Representations . . . 11
  - 2.2.1 Transition Systems . . . 11
  - 2.2.2 Single-path Linear-Constraint Loops . . . 13
- 2.3 Termination and Non-termination Analysis . . . 14
  - 2.3.1 Termination Analysis Using Ranking Functions . . . 16
  - 2.3.2 Non-termination Analysis Using Recurrent Sets . . . 22

3 Control-Flow Refinement of Transition Systems via Partial Evaluation 25
- 3.1 Partial Evaluation of Transition Systems . . . 26
  - 3.1.1 Constraint Horn-Clauses . . . 26
  - 3.1.2 From Transition Systems to Constraint Horn-Clauses . . . 26
  - 3.1.3 Partial Evaluation of Constraint Horn-Clauses . . . 28
  - 3.1.4 Choice of Properties . . . 30
- 3.2 Application to Termination Analysis . . . 32
  - 3.2.1 Control-Flow Refinement Schemes . . . 33
  - 3.2.2 Incorporating CFR into a Termination Algorithm . . . 34
  - 3.2.3 How CFR Benefits Termination Analysis . . . 36
- 3.3 Application to Cost Analysis . . . 40
  - 3.3.1 How CFR Benefits Cost Analysis . . . 41
  - 3.3.2 Using Ranking Functions as Properties . . . 45

4 Multi-Phase Ranking Functions and Their Relation to Recurrent Sets 47
4.1 Multi-Phase Ranking Functions for SLC loops . . . 49
4.2 Inferring Multi-Phase Ranking Functions . . . 49
4.2.1 Deciding Existence of MΦRFs . . . 50
4.2.2 Inference of Recurrent Sets . . . 56
4.2.3 Recurrent Sets for Transition Systems . . . 59
4.2.4 Cases in which Algorithm 3 does not Terminate . . . 60
4.3 The Displacement Polyhedron . . . 61
4.3.1 Witnesses for Non-existence of MΦRFs of a Given Depth . . . 64
4.3.2 Conditional Termination . . . 66
4.3.3 Termination and Non-termination of Bounded SLC Loops . . . 67
4.3.4 New Directions for the General MΦRF Problem . . . 67
4.4 Loops for Which MΦRFs are Sufficient . . . 68
4.4.1 Finite Loops . . . 68
4.4.2 The Class $\texttt{RF}(b)$ . . . 68
4.4.3 Loops with Affine-linear Updates . . . 69

5 Implementation 73
5.1 iRankFinder . . . 73
5.1.1 Input Syntax . . . 73
5.1.2 Invariant Generation . . . 75
5.1.3 Control-Flow Refinement . . . 76
5.1.4 Termination and Non-termination Analysis . . . 76
5.1.5 Handling Strict Inequalities . . . 77
5.1.6 Using iRankFinder . . . 77
5.2 EasyInterface . . . 79
5.2.1 General overview . . . 80
5.2.2 Using EasyInterface . . . 82

6 Experimental Evaluation 91
6.1 CFR for Termination Analysis . . . 91
6.2 CFR for Cost Analysis . . . 95
6.3 Non-Termination via Recurrent Sets (and CFR) . . . 96
6.4 Other Experiments with CFR . . . 99

7 Related Work 101
7.1 Terminating Analysis Using Ranking Functions . . . 101
7.2 Non-termination Analysis . . . 102
7.3 Control-Flow Refinement . . . 103

8 Conclusions and Future Work 105
8.1 Future Work . . . 106

Bibliography 109

Acronyms 117

List of Figures

1.1 A program and its corresponding transition system $\mathcal{T}$. 2

2.1 A polyhedron $\mathcal{P}$ and its integer hull $\mathcal{P}_I$. 10
2.2 A TS that corresponds to the SLC loop (1.2). 13
2.3 A loop that has a linear ranking function. 16
2.4 A loop that has a lexicographic ranking function, but not a linear ranking function. 18
2.5 A loop that has a LLRF according to the definition of [Bradley et al. 2005a]. 20
2.6 A loop that has a LLRF according to the definition of [Ben-Amram and Genaim 2014]. 20
2.7 A loop that has a LLRF according to the definition of [Larraz et al. 2013]. 21
2.8 A loop that has a MΦRF. 22
2.9 A non-terminating TS. 23

3.1 A CHC program and a corresponfing call graph. 26
3.2 A loop with 2 phases. 27
3.3 A loop with 2 phases that are never executed together. 37
3.4 A loop that searches in a circular queue. 38
3.5 A loop that implements a random walk process. 40
3.6 A non-terminating loop. 41
3.7 Iterative McCarthy. 43
3.8 A loop that calculates the *greatest common divisor* by iterative subtracting. 44
3.9 A SLC loop with 3 phases. 45

5.1 An example of TS in `flow-chart` syntax. It corresponds to TS $\mathcal{T}$ of Figure 3.2. 74
5.2 iRankFinder web-interface. 78
5.3 iRankFinder settings window. 79
5.4 The architecture of EasyInterface. 80
5.5 An example configuration file of a tool. 81
5.6 An example of a server request. 81
5.7 EasyInterface Web-Interface Client. 82
5.8 EIOL example. 83
5.9 EIOL stream command example. 83
5.10 EIOL download command example. 84
5.11 An example configuration file of two sets of examples. 85
5.12 EIOL general scheme. 87
5.13 Some EIOL commands. 88
5.14 EIOL actions examples. 89

List of Tables

3.1 Summary of applying cost analysis on the examples of Section 3.2.3 with and without CFR. 42

6.1 Summary of evaluation of CFR for termination analysis using LRFs. . . . 92
6.2 Summary of evaluation of CFR for termination analysis using LLRFs. . . 93
6.3 Total number of benchmarks for which each tool could prove termination. 95
6.4 Comparison of tools on individual benchmarks. Each cell indicates the number of benchmarks the tool at the corresponding row could prove terminating, and the tool at the corresponding column could not. 95
6.5 Evaluation of CFR for Cost Analysis. 96
6.6 Evaluation of the non-termination algorithm of Section 4.2.3. 97
6.7 Evaluation of CFR for Non-Termination. 98
6.8 The number of benchmarks for which each tool could prove Non-Termination. 98
6.9 Comparison of tools on individual benchmarks. Each cell indicates the number of benchmarks the tool at the corresponding row could prove non-terminating, and the tool at the corresponding column could not 98

Chapter 1

Introduction

Proving that a program will eventually terminate, i.e., that it does not go into an infinite loop, is one of the most fundamental tasks of program verification, and has been the subject of voluminous research. Perhaps the best known, and often used, technique for proving termination is that of *ranking functions*, which has already been used by Alan Turing in his early work on program verification [Turing 1949]. This consists of finding a function ρ that maps program states into the elements of a well-founded ordered set, such that $\rho(s) \succ \rho(s')$ holds for any consecutive states s and s'. This implies termination since infinite descent is impossible in a well-founded order. Besides proving termination, Turing [1949] mentions that ranking functions can be used to bound the length computations as well. This is useful in applications such as cost analysis and loop optimisation [Albert et al. 2011; Alias et al. 2010; Brockschmidt et al. 2016b; Feautrier 1992].

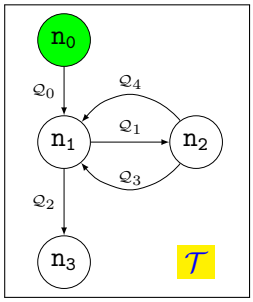
Termination of programs can be considered for all possible inputs, for a class of initial inputs, or for a single initial input. The later is known as the *halting problem*. Throughout this thesis, we will use the term *termination* for all these cases and explicitly state if an input, or a class of inputs, is provided when needed. Unlike termination of programs in general, which is undecidable, the algorithmic problems of detection (deciding the existence) or generation (synthesis) of a ranking function can well be solvable, given certain choices of the program representation, and the class of ranking functions. There is a considerable amount of research in this direction, in which different kinds of ranking functions for different kinds of program representations were considered. In some cases, the algorithmic problems have been completely settled, and efficient algorithms provided, while other cases remain open.

One of the program representations we study is *Single-path Linear-Constraint* (SLC) loop, where a state is described by the values of numerical variables, and the effect of a transition (one iteration) is described by a conjunction of *linear constraints*. Here is an example of this loop representation; primed variables x'_1 and x'_2 refer to the state following the transition:

$$\mathit{while}\ (x \leq y)\ \mathit{do}\ x' = x + 1,\ y' \leq y \tag{1.1}$$

Note that $x' = x + 1$ is an equation, not an assignment. The description of a loop may involve linear inequalities rather than equations, such as $y' \leq y$ above, and consequently, be non-deterministic. A more general program representation we study is *Transition Systems* (TSs). They are defined by *Control-Flow Graphs* (CFGs) with numerical variables consisting of nodes representing program locations, and edges annotated with linear constraints describing how values of variables change when moving from one location to

```
1 void phases1(int x, int y, int z) {
2   while( x >= 1 ) {
3     if ( y <= z - 1 ) {
4       y = y + 1;
5     } else {
6       x = x - 1;
7     }
8   }
9 }
```



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_1 \equiv\{x \geq 1, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_2 \equiv\{x \leq 0, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_3 \equiv\{y \leq z - 1, & x' = x, & y' = y + 1, & z' = z & \} \\
\mathcal{Q}_4 \equiv\{y \geq z, & x' = x - 1, & y' = y, & z' = z & \}
\end{array}
$$

Figure 1.1: A program and its corresponding transition system $\mathcal{T}$.

another[1]. An example of a TS is depicted in Figure 1.1 and it corresponds to the program to its left. Primed variables in the linear constraints refer to the state following the transition, exactly as in the case of SLC loop. Note SLC loops are also TSs.

In both program representations mentioned above, the domain of variables is also important as it typically affects the complexity of the underlying decision and synthesis problems. We consider the settings of integer-valued variables and rational-valued (or real-valued) variables, however, for simplicity, we mostly omit discussions on the domain of variables in our examples, and when it is important we state it explicitly. Although our program representations allow only numerical variables and linear constraints, data structures can be handled using *size abstractions*, e.g., length of lists, depth of trees, etc. [Bruynooghe et al. 2007; Giesl et al. 2017; Lee et al. 2001; Lindenstrauss and Sagiv 1997; Magill et al. 2010; Spoto et al. 2010]. In such case, variables represent sizes of corresponding data structures.

1.1 Classes of (Linear-based) Ranking Functions

Due to practical considerations, termination analysis tools typically focus on classes of ranking functions that can be synthesised efficiently. This, however, does not mean that theoretical aspects of such classes are put aside, as understanding theoretical limits and properties of the underlying problems is necessary for developing practical algorithms, after all, *"there is nothing as practical as a good theory"*.

The most popular class of ranking functions in this context is probably that of *Linear Ranking Functions* (LRFs). A LRF is a function $\rho(x_1, \ldots, x_n) = a_1x_1 + \cdots + a_nx_n + a_0$ such that any transition from $\vec{x}$ to $\vec{x}'$ satisfies (i) $\rho(\vec{x}) \geq 0$; and (ii) $\rho(\vec{x}) - \rho(\vec{x}') \geq 1$. For example, $\rho(x, y) = y - x$ is a LRF for Loop (1.1). Several polynomial-time algorithms to find a LRF using linear programming exist [Alias et al. 2010; Colón and

[1]We use CFG and TS interchangeably, however, formally a CFG is part of the definition of a TS.

Sipma 2001; Feautrier 1992; Mesnard and Serebrenik 2008; Podelski and Rybalchenko 2004; Sohn and Gelder 1991]. These algorithms are complete[2] for TSs with rational-valued variables, but not with integer-valued variables. Ben-Amram and Genaim [2013] showed how completeness for the integer case can be achieved, and also classified the corresponding decision problem as co-NP complete.

Despite their popularity, LRFs do not suffice for all TSs, and a natural question is what to do when an LRF does not exist; and a natural answer is to try a richer class of ranking functions. Of particular importance is the class of *Lexicographic Linear Ranking Function*s (LLRFs). These are tuples of linear functions that decrease lexicographically over a corresponding well-founded ordered set. For example, the program depicted in Figure 1.1 does not have a LRF, but it has the LLRF $\langle z-y, x\rangle$ since: for the *then* branch $z-y$ decreases and for the *else* branch x decrease while $z-y$ does not change. LLRFs might be necessary even for the simple loops. For example, the following SLC loop

$$\mathit{while}\ (x+z\geq 0)\ \mathit{do}\ x'=x+y,\ y'=y+z, z'=z-1 \tag{1.2}$$

does not have a LRF, but can be proved terminating using the LLRF $\langle z, y, x\rangle$.

There are several definitions for LLRFs in the literature [Alias et al. 2010; Ben-Amram and Genaim 2014; Bradley et al. 2005a; Larraz et al. 2013] and they have different power, i.e., some can prove termination of a program while others fail (see Section 2.3.1 for a detailed discussion). They have corresponding polynomial-time synthesis algorithms for the case of rational variables, and the underlying decision problems are co-NP complete for the case of integer variables [Ben-Amram and Genaim 2014; Ben-Amram and Genaim 2015]. The definition of Larraz et al. [2013] is the most general in this spectrum of LLRFs, its complexity classification is not known yet and corresponding complete synthesis algorithms do not exist.

The LLRF $\langle z, y, x\rangle$ used above for Loop (1.2) belongs to a class of LLRFs that is know as *Multi-Phase Linear Ranking Function*s (MΦRFs). Ranking functions in this class are characterised by the following behaviour: the first component always decreases, and when it becomes negative the second component starts to decrease, and when it becomes negative the third component starts to decrease, and so on. This behaviour defines phases through which executions pass. This is different from LLRFs in general since once a component becomes negative it cannot be used anymore.

Many problems related to algorithmic and complexity aspects of MΦRFs are still open, even for the special case of SLC loops. They have mainly been studied for a special case in which an upper-bound d on the length of the tuple is given as input. Leike and Heizmann [2015] showed how to synthesise bounded MΦRFs by solving corresponding non-linear constraints. Ben-Amram and Genaim [2017] settled the complexity of synthesising and deciding the existence of bounded MΦRFs for SLC loops and showed that for SLC loops MΦRFs are as powerful as the most general definition of LLRFs [Larraz et al. 2013]. Ben-Amram and Genaim [2017] show as well that, for SLC loops, MΦRFs induce a linear bound on the number of iterations, which makes this class attractive for applications such as cost analysis.

[2]Complete means that if there is a LRF, they will find one.

1.2 Control-Flow Refinement

In the previous section, we have seen that when failing to obtain a termination proof using a simple class of ranking functions, one might resort to a richer class which typically comes at the price of performance. This is common in program analysis in general, not only in termination analysis. For example, in abstract interpretation [Cousot and Cousot 1977] when we fail to prove a property using some abstract domain, e.g., octagons [Miné 2006], we might resort to a less abstract one, e.g., polyhedra [Cousot and Halbwachs 1978].

Loss of precision is often caused by complex or implicit control-flow since for such cases, abstract properties of several execution paths are merged. There are techniques that overcome the loss of precision using program transformations to simplify the control-flow, and thus makes it possible to infer the desired properties even with a weaker version of the analysis. This is known as *Control-Flow Refinement* (CFR).

CFR can be useful to improve the precision of termination analysis as well. Consider the program of Figure 1.1 again, and recall that it does not have a LRF and that for proving termination we used LLRFs. Examining this program carefully, we can see that any execution passes in two phases: in the first one, `y` is incremented until it reaches the value of `z`, and in the second phase `x` is decremented until it reaches `0` (this is a bit different from the notion of phases for MΦRFs since y does not continue decreasing during the second phase). Lets us transform the program of Figure 1.1 into a semantically equivalent one such that the two phases are separate and explicit:

```
1  while (x >= 0 and y <= z-1) y = y + 1;
2  while (x >= 0 and y >= z) x = x - 1;
```

Now we can prove termination of this program using LRFs only: for the first $z - y$ is a LRF, and for the second x is a LRF. Moreover, cost analysis tools that are based on bounding loop iterations using LRFs would infer a linear bound for this program while they would fail on the original one.

Apart from simplifying the termination proof, there are also cases where it is not possible to prove termination without such transformations even when using LLRFs (see Chapter 3). In addition, CFR can help in inferring more precise invariants, without the need for expensive abstract domains, which can benefit any analysis that uses such invariants, e.g., termination and cost analysis.

CFR has been considered before by Gulwani et al. [2009] and Flores-Montoya and Hähnle [2014] to improve the precision of cost analysis, and by Sharma et al. [2011] to improve invariants in order to prove assertions. While all these techniques can automatically obtain the transformed program above, they are developed from scratch and tailored to some analysis of interest. Recently, CFR has been considered by Albert et al. [2019] to improve cost analysis as well, however from a different perspective that uses termination witnesses to guide CFR.

1.3 Objectives and Methodology

Section 1.1 and 1.2 discussed two approaches to improve the precision of termination analysis on failure. The first one relies on resorting to sophisticated classes of ranking functions and more precise abstract domains for invariants, and the second relies on using CFR techniques to simplify complex control-flow. The objectives of this thesis are guided by these two approaches.

Our first objective is to explore the use of CFR in the context of termination analysis, as well as other related areas such as invariant generation and cost analysis. However, instead of developing dedicated new techniques from scratch, our methodology is based on exploring the use of well-studied program specialisation techniques for this purpose. We aim at adapting these techniques for TSs, and allow configuring them automatically to suit different purposes. We also aim at exploring different strategies for applying them, not only as a pre-processing step that transforms the whole program but also at different levels of granularity. In addition, apart from using CFR for termination analysis, we aim at providing independent tools for applying CFR on TSs, and thus to promote integrating CFR in program analysis tools without additional effort.

Our second objective is to study the class of MΦRFs, in particular the general case that does not impose a bound on the length of corresponding tuples. As we have mentioned before, this class of ranking functions has not received enough attention from the community. We are interested in exploring theoretical properties of MΦRFs, as well as their practical use for termination analysis of TSs with complex control-flow. We are particularly interested in the case of SLC loops, which are important for theoretical aspects of termination, but also in generalising the results for TSs. We anticipate and note, as we will see in Chapter 4, that this objective led us to unexpected research direction in which we explored the relationship between MΦRFs and non-termination, and thus parts of this thesis will also address non-termination analysis. The tools and methodology we use to address this objective are similar to those used by Ben-Amram and Genaim [2017], which are mainly based on using different aspects of linear programming theory.

Our third objective, which is a direct consequence of the other objectives, is to develop a termination analyser that includes the techniques developed for the first two objectives, as well as other state-of-the-art termination analysis techniques. We will also use this tool to evaluate the practical effect of these techniques.

1.4 Summary of Contributions and Outline

In Chapter 2, we start this thesis by providing necessary background material which includes: definitions and some results related to polyhedra; formal definition of the program representations that we use throughout this thesis; and a general termination analysis algorithm and different notions of ranking functions and recurrent sets which are used in the context of non-termination – the discussion on ranking functions, in particular LLRFs, is made such that different existing notions of LLRFs fit in the same algorithmic framework which makes describing and comparing them easier.

For program representations, we do not use programs directly, e.g., Java or C, but rather all our research is presented using TSs that we informally described at the beginning of this chapter. We prefer this representation since it is popular in the program analysis community, and also can represent programs written in different programming languages, and thus, in principle, analyses develop for TSs can be used in the context of different programming languages. In addition, in some parts of the thesis, we use the special case of SLC loops, which is common for studying theoretical aspects of termination analysis.

Control-Flow Refinement via Partial Evaluation

Since CFR, as discussed in Section 1.2, is in principle a program transformation that specialises programs to distinguish different execution scenarios, in Chapter 3 we explore on the use of general-purpose specialisation techniques for CFR, in particular the partial evaluation techniques of Gallagher [2019]. Basing CFR on such well-studied techniques has the clear advantage that soundness comes for free because partial evaluation guarantees semantic equivalence between the original program and its transformed version. Moreover, this way we obtain a CFR procedure that is not tailored for a particular purpose, but rather can be tuned depending on the application domain.

We suggest heuristics for automatically configuring partial evaluation (i.e., inferring properties to guides specialisation) in order to achieve the desired CFR, and also suggest the use of termination witnesses (ranking functions) as properties for CFR in the context of cost analysis. We integrate such a CFR procedure in the termination analysis algorithm described in Chapter 2 in a way that allows applying CFR at different levels of granularity, and thus controlling the trade-off between precision and performance. This is done by suggesting different schemes for applying CFR, not only as a pre-processing step but also on specific parts of the TS which we could not prove terminating. We discuss, using an extensive series of examples, how CFR can benefit both termination and cost analysis. This work has been reported in

> Jesús J. Doménech, John P. Gallagher, and Samir Genaim. Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis. *Theory and Practice of Logic Programming*, 19(5-6):990–1005, 2019. **DOI:** 10.1017/S1471068419000310

and has been presented at the 35$^{\text{th}}$ *International Conference on Logic Programming* as well. Implementation and experimental evaluation, that demonstrates the usefulness of our approach to CFR, are reported in corresponding dedicated chapters (see below).

A New Look at Multi-Phase Ranking Functions

As we have discussed in Section 1.1, many questions related to algorithmic and complexity aspects of MΦRFs are still open. In Chapter 4, we turn our attention to MΦRFs for SLC loops, and make progress towards solving the problem of *existence of MΦRFs*, i.e., seeking (or deciding existence) MΦRFs without a given bound on the depth. This is how our research on MΦRFs has started, however, it has led to unexpected directions that we summarise below.

We present an algorithm for seeking MΦRFs that reveals new insights on the structure of these ranking functions. Our algorithm starts from the set of transitions of the given SLC loop, constructs the set of all linear functions that are *non-negative on all enabled states*, and remove all transitions for which there is such a *non-negative* function that is decreasing. This is applied iteratively since, after removing some transitions, more functions may satisfy the non-negativity condition, and they may eliminate additional transitions in the next iteration. When all transitions are eliminated in a finite number of iterations, we can construct a MΦRF using the corresponding *non-negative* functions; and when reaching a situation where no transition can be eliminated, we have actually reached a recurrent set that witnesses non-termination. This reveals a surprising relation between ranking functions and recurrent sets. We also show how it can be used to compute recurrent sets for TSs in general. Our algorithm is not complete as it might not

terminate in some cases, however, even in this case, it provides important insights that raise some new questions and research directions.

Our research has, besides, led to a new representation for SLC loops that provides us with new tools for studying the termination of SLC loops in general, and the existence of a MΦRF in particular. As evidence on the usefulness of this new representation, we also show that some non-trivial observations on termination of bounded SLC loops are made straightforward in this representation, while they are not easy to see in the normal representation. The results of this part have been published in

> Amir M. Ben-Amram, Jesús J. Doménech, and Samir Genaim. Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets, In Bor-Yuh Evan Chang, editor, *Proceedings of the 26th International Symposium on Static Analysis (SAS'19)*, volume 11822 of *Lecture Notes in Computer Science*, pages 459–480. Springer, 2019. **DOI:** 10.1007/978-3-030-32304-2_22

Besides, in a recent work that has not been published before and that we report for the first time in Section 4.4, we show that the insights made by this work are useful for finding classes of SLC loops for which, when terminating, there is always a MΦRF and thus have linear runtime bound.

Implementation, Evaluation, and Dissemination

In Chapter 5, we present iRankFinder, a termination analyser that has been developed in the context of this thesis and implements, among other things, the techniques developed in chapters 3 and 4. In Chapter 6, we report an experimental evaluation of iRankFinder, in particular, we evaluate the benefits of its CFR component and the use of the techniques of Chapter 4 for non-termination analysis of TSs.

During PhD studies or the lifetime of a research project in general, providing the community with easy access to research prototype tools is crucial to promote the research or get feedback. This can be achieved by building *Graphical User Interfaces* (GUIs) that facilitate trying tools; in particular, tools with web-interfaces can be tried without the overhead downloading and installing them. However, researchers typically avoid developing GUIs until tools are fairly stable since the tools change continuously, and also because programming plug-ins for sophisticated frameworks and building web-interfaces from scratch are tedious tasks.

In this thesis, we also wanted to make iRankFinder available via a web-interface right from the beginning, but instead of building a dedicated web-interface we have opted at solving a general problem and developed an open-source toolkit, called EasyInterface, that aims at simplifying the process of building GUIs for research prototypes tools, and thus improve the dissemination of the corresponding research. EasyInterface provides an easy and almost immediate way to make existing (command-line) applications available via a web-interface. This toolkit has been presented in the following publication

> Jesús J. Doménech, Samir Genaim, Einar Broch Johnsen, and Rudolf Schlatte. EasyInterface: A Toolkit for Rapid Development of GUIs for Research Prototype Tools, In Marieke Huisman and Julia Rubin, editors, *20th International Conference on Fundamental Approaches to Software Engineering (FASE'17)*, volume 10202 of *Lecture Notes in Computer Science*, pages 379–383. Springer, 2017. **DOI:** 10.1007/978-3-662-54494-5_22

Apart from the web-interface of iRankFinder, this toolkit has been used for building web-interfaces for other tools developed in our research group, and also for tools developed by other teams in the context of research projects in which our research group is involved.

Finally, in Chapter 7, we survey related work and, in Chapter 8, we draw our conclusions and discuss possible future work.

Chapter 2

Preliminaries

In this chapter, we give necessary background on polyhedra, formally define the different program representations that we will be using throughout this thesis, and provide some necessary background related to termination and non-termination analysis.

2.1 Polyhedra

In this section, we give some necessary background on polyhedra. The sets of integer and rational numbers are denoted by $\mathbb{Z}$ and $\mathbb{Q}$ respectively. We use $\mathbb{D}$ to refer to either $\mathbb{Z}$ or $\mathbb{Q}$ when the domain is not fixed. Column and row vectors of n elements (such as variable, numbers, etc.) are written as follows

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad\qquad \vec{x} = \begin{pmatrix} x_1 & \dots & x_n \end{pmatrix}$$

We sometimes write $\mathbf{x}$ as a transposition of $\vec{x}$, denote as $\vec{x}^{\mathrm{T}}$, and vice versa. A *linear constraint* (or linear inequality) ψ is of the form

$$\sum_{i=1}^{n} a_i x_i \leq b$$

where x_i are variables, and $a_i, b \in \mathbb{Q}$. We call $\vec{a} = (a_1, \dots, a_n)$ the coefficients and b the free constant. A mapping $\sigma : \vec{x} \mapsto \mathbb{D}$ is a satisfying assignment for ψ if the inequality $\sum_{i=1}^{n} a_i \sigma(x_i) \leq b$ is *true*. For simplicity, strict inequalities $\sum_{i=1}^{n} a_i x_i < b$ are not allowed explicitly, however, in Section 5.1.5 we discuss how they are handled in practice.

A (linear) *formula* φ is a Boolean formula where the atoms are linear constraints. The set of variables of φ is denoted by $\texttt{VARS}(t)$. A formula φ is satisfiable if there is $\sigma : \texttt{VARS}(\varphi) \mapsto \mathbb{D}$ such that the corresponding Boolean formula is satisfiable wrt. σ (each linear constraint evaluates to *true* or *false* wrt. σ). We call σ a solution for φ.

We are particularly interested in formulas that are conjunctions of linear constraints, i.e., of the form $\psi_1 \wedge \dots \wedge \psi_m$, which we also write as a set $\varphi = \{\psi_1, \dots, \psi_m\}$. The matrix representation of φ is $A\mathbf{x} \leq \mathbf{b}$ where $A \in \mathbb{Q}^{m \times n}$ is a rational matrix of n columns and m rows such that the ith row consists of the coefficients of ψ_i, and $\mathbf{b} \in \mathbb{Q}^m$ such that the ith row consists of the free constant of ψ_i. For simplicity, when writing linear constraints explicitly, we use $\geq$ and $=$ as well. Both cases can be normalised to a form with $\leq$ only.

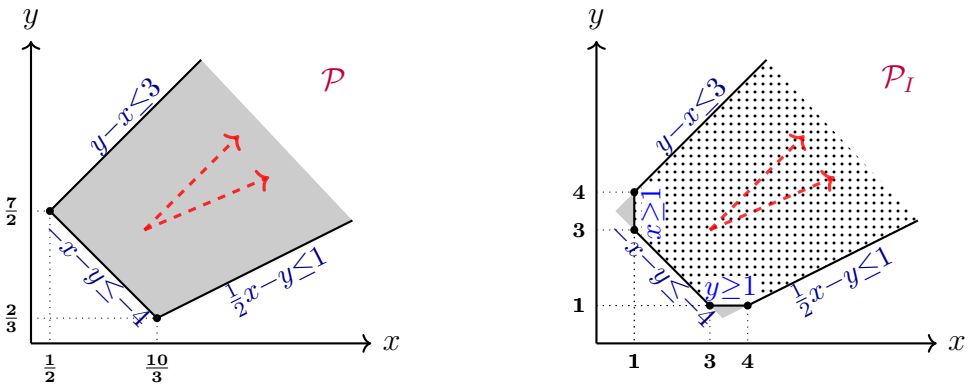


Figure 2.1: A polyhedron $\mathcal{P}$ and its integer hull $\mathcal{P}_I$.

A *rational convex polyhedron* $\mathcal{P} \subseteq \mathbb{Q}^n$ (*polyhedron* for short) is the set of solutions of a (linear) formula $A\mathbf{x} \leq \mathbf{b}$ with rational variables, namely:

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$$

where $\mathbf{x} \in \mathbb{Q}^n$. We say that $\mathcal{P}$ is specified by $A\mathbf{x} \leq \mathbf{b}$. If $\mathbf{b} = \mathbf{0}$, then $\mathcal{P}$ is called a *cone*. The set of the *recession directions* of a polyhedron $\mathcal{P}$ specified by $A\mathbf{x} \leq \mathbf{b}$, also known as its *recession cone*, is defined by the following cone:

$$\texttt{rec.cone}(\mathcal{P}) = \{\mathbf{y} \in \mathbb{Q}^n \mid A\mathbf{y} \leq \mathbf{0}\}$$

EXAMPLE 2.1. Consider the polyhedron $\mathcal{P}$ of Figure 2.1 (on the left). Points defined by the grey area and the black borders are solutions to the system of linear constraints

$$\{y - x \leq 3\,,\ -x - y \leq -4\,,\ \frac{1}{2}x - y \leq 1\}$$

Note that we consider all rational points. □

For a given polyhedron $\mathcal{P} \subseteq \mathbb{Q}^n$ we let $I(\mathcal{P})$ be $\mathcal{P} \cap \mathbb{Z}^n$, i.e., the set of integer points of $\mathcal{P}$. The *integer hull* of $\mathcal{P}$, commonly denoted by $\mathcal{P}_I$, is defined as the convex hull of $I(\mathcal{P})$, i.e., every rational point of $\mathcal{P}_I$ is a convex combination of integer points. It is known that $\mathcal{P}_I$ is also a polyhedron and that $\texttt{rec.cone}(\mathcal{P}) = \texttt{rec.cone}(\mathcal{P}_I)$. An *integer polyhedron* is a polyhedron $\mathcal{P}$ such that $\mathcal{P} = \mathcal{P}_I$. An integer polyhedron is also called *integral*.

EXAMPLE 2.2. The integer hull $\mathcal{P}_I$ of polyhedron $\mathcal{P}$ of Figure 2.1 (on the left) is given in the same figure (on the right). It is defined by the dotted area and the black border and is obtained by adding the inequalities $x \geq 1$ and $y \geq 1$ to $\mathcal{P}$. The two grey triangles next to the edges of $\mathcal{P}_I$ are subsets of $\mathcal{P}$ that were eliminated when computing $\mathcal{P}_I$. □

Polyhedra also have another important representation called the *generator representation*. It defines the polyhedron in terms of vertices and rays as follows:

$$\mathcal{P} = \texttt{conv.hull}\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} + \texttt{cone}\{\mathbf{y}_1, \ldots, \mathbf{y}_t\}\,.$$

This means that $\mathbf{x} \in \mathcal{P}$ iff

$$\mathbf{x} = \sum_{i=1}^{m} a_i \cdot \mathbf{x}_i + \sum_{j=1}^{t} b_j \cdot \mathbf{y}_j$$

for some rationals $a_i, b_j \geq 0$, where $\sum_{i=1}^{m} a_i = 1$. Note that $\mathbf{y}_1, \ldots, \mathbf{y}_t$ are the recession directions of $\mathcal{P}$, i.e., $\mathbf{y} \in \texttt{rec.cone}(\mathcal{P})$ iff

$$\mathbf{y} = \sum_{j=1}^{t} b_j \cdot \mathbf{y}_j$$

for some rationals $b_j \geq 0$. If $\mathcal{P}$ is integral, then there is a generator representation in which all $\mathbf{x}_i$ and $\mathbf{y}_j$ are integers. An empty polyhedron is represented by an empty set of vertices and rays.

EXAMPLE 2.3. The generator representations of $\mathcal{P}$ and $\mathcal{P}_I$ of Figure 2.1 are

$$\begin{aligned} \mathcal{P} &= \texttt{conv.hull}\{(\tfrac{1}{2}, \tfrac{7}{2}), (\tfrac{10}{3}, \tfrac{2}{3})\} + \texttt{cone}\{(1,1), (7,3)\} \\ \mathcal{P}_I &= \texttt{conv.hull}\{(1,3), (1,4), (3,1), (4,1)\} + \texttt{cone}\{(1,1), (7,3)\} \end{aligned}$$

The points in `conv.hull` are vertices, they correspond to the points marked with $\bullet$ in Figure 2.1. The rays are the vectors $(1,1)$ and $(7,3)$; they describe a direction, rather than a specific point, and are therefore represented in Figure 2.1 by the red arrows. Note that the vertices of $\mathcal{P}_I$ are integer points, while those of $\mathcal{P}$ are not. For example, in $\mathcal{P}$ the point $(3,2)$ is defined as

$$\frac{5}{17} \cdot (\frac{1}{2}, \frac{7}{2}) + \frac{12}{17} \cdot (\frac{10}{3}, \frac{2}{3}) + \frac{1}{2} \cdot (1,1) + 0 \cdot (7,3)$$

and in $\mathcal{P}_I$ as

$$0 \cdot (1,3) + \frac{1}{3} \cdot (1,4) + 0 \cdot (3,1) + \frac{2}{3} \cdot (4,1) + 0 \cdot (1,1) + 0 \cdot (7,3)$$

□

Let $\mathcal{P} \subseteq \mathbb{Q}^{n+m}$ be a polyhedron, and let $\left(\begin{smallmatrix}\mathbf{x}\\ \mathbf{y}\end{smallmatrix}\right) \in \mathcal{P}$ be such that $\mathbf{x} \in \mathbb{Q}^n$ and $\mathbf{y} \in \mathbb{Q}^m$. The *projection of* $\mathcal{P}$ onto the $\mathbf{x}$-space is defined as

$$\texttt{proj}_{\mathbf{x}}(\mathcal{P}) = \{\mathbf{x} \in \mathbb{Q}^n \mid \exists \mathbf{y} \in \mathbb{Q}^m \text{ such that } \left(\begin{smallmatrix}\mathbf{x}\\ \mathbf{y}\end{smallmatrix}\right) \in \mathcal{P}\}. \tag{2.1}$$

This operation simply restricts $\mathcal{P}$ to some coordinates of interest. Note that this operation is well defined for any subset of $\mathbb{Q}^n$, not only polyhedral subsets.

2.2 Program Representations

In this section, we describe the different program representations that we will be using throughout this thesis.

2.2.1 Transition Systems

Programs in this thesis are defined using TSs, which is a very common representation that is often used in program analysis. The advantage of using TSs is that they abstract away from the particularities of a specific programming language, and thus, in principle, analyses develop for such programs can be used in the context of different programming languages. A TS is typically defined by a corresponding CFG that defines the possible flows that executions can follow. We will use the terms CFGs are TSs interchangeably, though CFG is used when referring to the graphical representation, and TS when referring to the underlying mathematical object.

Definition 2.4 (Transition System)**.** A *transition system* $\mathcal{T}$ is a tuple

$$\mathcal{T} = \langle V, N, \mathtt{n}_0, E \rangle$$

where

1. $V = \{x_1, \ldots, x_n\}$ is a set of program variables over some fixed domain $\mathbb{D}$ (we will be using $\mathbb{Q}$ and $\mathbb{Z}$);
2. N is a set of nodes;
3. $\mathtt{n}_0 \in N$ is the entry node; and
4. E is a set of labelled edges. An edge has the form $\mathtt{n}_\mathtt{s} \xrightarrow{\mathcal{Q}} \mathtt{n}_\mathtt{t}$ such that $\mathtt{n}_\mathtt{s}, \mathtt{n}_\mathtt{t} \in N$ and $\mathcal{Q} \subseteq \mathbb{Q}^{2n}$ is a polyhedron over variables $V \cup V'$ where $V' = \{x'_i \mid x_i \in V\}$. We refer to $\mathcal{Q}$ as the *transitions polyhedron* of the edge.

We assume that the entry node $\mathtt{n}_0$ has no incoming edges. We say that a TS is a *rational* TS if the variables take rational values, i.e. $\mathbb{D} = \mathbb{Q}$, and *integer* if the variables take integer values, i.e., $\mathbb{D} = \mathbb{Z}$.

A *program state* is a pair $(\mathtt{n}, \sigma)$, where $\mathtt{n} \in N$ and $\sigma : V \mapsto \mathbb{D}$ is a mapping. We often view σ as a vector $\mathbf{x} = (\sigma(x_1), \ldots, \sigma(x_n))^\mathrm{T}$, and when it is clear from the context we call $\mathbf{x}$ a state as well. There is a *transition* (i.e. a valid execution step) from $s_1 = (\mathtt{n}_\mathtt{s}, \sigma)$ to $s_2 = (\mathtt{n}_\mathtt{t}, \sigma')$, denoted as $s_1 \to s_2$, if there is an edge $\mathtt{n}_\mathtt{s} \xrightarrow{\mathcal{Q}} \mathtt{n}_\mathtt{t} \in E$ such that the point $(\sigma(x_1), \ldots, \sigma(x_n), \sigma'(x_1), \ldots, \sigma'(x_n))^\mathrm{T}$ is in the transitions polyhedron $\mathcal{Q}$. We call $\mathtt{n}_\mathtt{s}$ and $\mathtt{n}_\mathtt{t}$ source and target nodes of the transition respectively. When it is clear from the context, we ignore the nodes and say that there is a transition from $(\sigma(x_1), \ldots, \sigma(x_n))^\mathrm{T}$ to $(\sigma'(x_1), \ldots, \sigma'(x_n))^\mathrm{T}$.

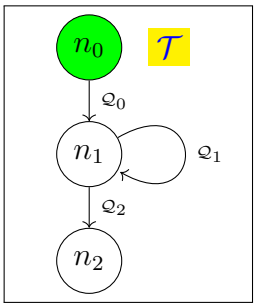
A transition can be seen also as a point $\binom{\mathbf{x}}{\mathbf{x}'} \in \mathcal{Q}$, where its first n components correspond to $\mathbf{x}$ and its last n components to $\mathbf{x}'$. For ease of notation, we denote $\binom{\mathbf{x}}{\mathbf{x}'}$ by $\mathbf{x}''$. States in $\mathtt{proj}_\mathbf{x}(\mathcal{Q})$ are called the *enabled states* of $\mathcal{Q}$, i.e., states from which we can make a transition. Note that the primed variables V' refer to the program state following the transition. A *trace* is a sequence of states $s_0 \to s_1 \to \cdots$ such that $s_i \to s_{i+1}$ is a transition. We write $s_i \to^* s_j$ to indicate that state s_j is reachable from state s_i. A trace might be of finite or infinite length.

The definition of a transition system, in related literature, sometimes include a formula over V that describes the set of valid initial states, however, for simplicity, we do not include such a formula in the above definition since for our purposes it can always be added to outgoing edges of $\mathtt{n}_0$.

EXAMPLE 2.5. The CFG $\mathcal{T}$ in Figure 1.1 defines a TS that corresponds to the program `phases1` to its left. The entry node is $\mathtt{n}_0$, from which we can move to $\mathtt{n}_1$ without changing the value of any variable as specified by $\mathcal{Q}_0$. Nodes $\mathtt{n}_1$ and $\mathtt{n}_2$, and the edges between them correspond to the loop body: edge $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_1} \mathtt{n}_2$ corresponds to entering the loop when the loop condition holds; edge $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_3} \mathtt{n}_1$ corresponds the *then* branch; and edge $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_4} \mathtt{n}_1$ corresponds to the *else* branch. Finally, edge $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_3$ corresponds to exiting the while loop. □

Throughout this thesis, we often rely on (polyhedral) invariants in order to improve the precision of termination or non-termination analysis. An *invariant* for a given node $\mathtt{n} \in N$

```
1 void phases_xyz(int x, int y, int z) {
2     while( x + z >= 0 ) {
3         x = x + y;
4         y = y + z;
5         z = z - 1;
6     }
7 }
```



$$\begin{array}{lllll} \mathcal{Q}_0 \equiv\{ & x' = x, & y' = y, & z' = z & \} \\ \mathcal{Q}_1 \equiv\{x + z \geq 0, & x' = x + y, & y' = y + z, & z' = z - 1 & \} \\ \mathcal{Q}_2 \equiv\{x + z \leq -1, & x' = x, & y' = y, & z' = z & \} \end{array}$$

Figure 2.2: A TS that corresponds to the SLC loop (1.2).

is a polyhedron $\mathcal{P} \subseteq \mathbb{Q}^n$, such that for any assignment σ_0 and trace $(\mathtt{n}_0, \sigma_0) \rightarrow^* (\mathtt{n}, \sigma)$ we have $(\sigma(x_1), \ldots, \sigma(x_n))^{\mathrm{T}} \in \mathcal{P}$, i.e., when viewing $\mathcal{P}$ as a formula, it is guaranteed to hold whenever node $\mathtt{n}$ is reached.

EXAMPLE 2.6. For the TS $\mathcal{T}$ of Figure 1.1, $\mathcal{P} \equiv \{x \geq 0\}$ is an invariant for node $\mathtt{n}_2$ and $\mathcal{P} \equiv \{x \leq 0\}$ is an invariant for node $\mathtt{n}_3$. □

Termination of TSs. A TS is *non-terminating* if there is a trace of infinite length starting in an initial state $(\mathtt{n}_0, \sigma_0)$ for some σ_0, and is *terminating* otherwise, i.e., if all possible traces are of finite lengths. Sometimes we restrict our interest to termination wrt. initial states that satisfy some conditions, however, this can be modelled by adding these conditions to the outgoing edges of $\mathtt{n}_0$.

Complexity of TSs. The *complexity* (also called cost) of a TS is typically defined in terms of the maximum length of its traces, where each execution step contributes 1 to the total cost. In order to model cost in a way that is amenable to program transformations, where edges/nodes might be eliminated/added, we can add an extra *auxiliary variable* x_{n+1} whose value is 0 in the initial state and is incremented by 1 in every transition (x_{n+1} can be modified in other ways to capture different cost models). The cost of a TS, wrt. to an initial state, is then defined as the supremum of all values of x_{n+1} in all reachable states. A function $f : \mathbb{D}^n \mapsto \mathbb{Q}$ is an upper-bound on the cost of $\mathcal{T}$ if for any possible trace $(\mathtt{n}_0, \sigma_0) \rightarrow^* (\mathtt{n}, \sigma)$, we have $f(\sigma_0(x_1), \ldots, \sigma_0(x_n)) \geq \sigma'(x_{n+1})$.

2.2.2 Single-path Linear-Constraint Loops

In this thesis, we are particularly interested in TSs that consist of a single node and single edge, in addition to the entry and exits nodes. This is the case of the TS depicted in Figure 2.2, which corresponds to Loop (1.2) that we have seen in Chapter 1. This kind of TSs is common when studying theoretical aspects of termination and non-termination, and is often represented as follows.

Definition 2.7 (Single-path Linear-Constraint Loop)**.** A *SLC* loop over n variables has the form

$$\textit{while } (B\mathbf{x} \leq \mathbf{b}) \textit{ do } A\mathbf{x} + A'\mathbf{x}' \leq \mathbf{c} \tag{2.2}$$

where for some $p, q > 0$, $B \in \mathbb{Q}^{p\times n}$, $A, A' \in \mathbb{Q}^{q\times n}$, $\mathbf{b} \in \mathbb{Q}^p$, $\mathbf{c} \in \mathbb{Q}^q$, $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$, and $\mathbf{x}' = (x'_1, \ldots, x'_n)^{\mathrm{T}}$.

The formula $B\mathbf{x} \leq \mathbf{b}$ is called *the loop condition* (a.k.a. the loop guard) and the formula $A\mathbf{x} + A'\mathbf{x}' \leq \mathbf{c}$ is called *the update*. It is easy to see that Loop (2.2) represents a TS with a single node $\mathtt{n_1}$ and an edge $\mathtt{n_1} \xrightarrow{\mathcal{Q}} \mathtt{n_1}$, where the transitions polyhedron $\mathcal{Q}$ is specified by a set of inequalities $A''\mathbf{x}'' \leq \mathbf{c}''$ such that

$$A'' = \begin{pmatrix} B & 0 \\ A & A' \end{pmatrix} \qquad\qquad \mathbf{c}'' = \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix}$$

We call $\mathcal{Q}$ the *transitions polyhedron* as in the case of TSs. The loop (and the corresponding TS) is fully represented by this polyhedron, and thus we ignore node $\mathtt{n_1}$ when writing a state, i.e., a state is simply $\mathbf{x}'' \in \mathcal{Q}$. For integer loops, the set of transitions is denoted by $I(\mathcal{Q})$, i.e., the set of integer points of $\mathcal{Q}$.

The update of Loop (2.2) is called *deterministic* if, for a given $\mathbf{x}$ satisfying the loop guard there is at most one $\mathbf{x}'$ satisfying the update formula. The update is called *affine linear* if it can be rewritten as

$$\mathbf{x}' = U\mathbf{x} + \mathbf{c} \tag{2.3}$$

for a matrix $U \in \mathbb{Q}^{n\times n}$ and vector $\mathbf{c} \in \mathbb{Q}^n$.

EXAMPLE 2.8. The TS depicted in Figure 2.2 corresponds to a SLC loop with *deterministic affine linear* update defined by:

$$B = \begin{pmatrix} -1 & 0 & -1 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{c} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

□

As in the case of TSs, we say that a loop is a *rational loop* if the variables take rational values, i.e., $\mathbf{x}$ and $\mathbf{x}'$ range over $\mathbb{Q}^n$, and that it is an *integer loop* if they take integer values, i.e., $\mathbf{x}$ and $\mathbf{x}'$ range over $\mathbb{Z}^n$. One could also allow variables to take any real-number value, but for the problems we study, where the constraints are expressed by rational numbers, this very rarely differs from the rational case (when it does, we comment on that explicitly).

2.3 Termination and Non-termination Analysis

In this section, we describe general techniques and algorithms for proving termination and non-termination that we will use throughout this thesis.

The pseudo-code depicted in Algorithm 1 corresponds to a general algorithm for termination and non-termination analysis – it will be mainly used in Chapter 3. It consists of procedure TerminationCFG that uses procedure TerminationSCC as a black-box (details of TerminationSCC will be given in sections 2.3.1 and 2.3.2).

Algorithm 1: Pseudocode of (Non)Termination Analysis.

```
   TerminationCFG(𝒯)
1  |  F := ∅
2  |  foreach SCC S of 𝒯 do
3  |  |  ⟨Ans, F_S⟩ := TerminationSCC(S,𝒯)
4  |  |  if Ans is NO then
5  |  |  |  return ⟨NO, F_S⟩
6  |  |  F := F ∪ F_S
7  |  if F ≠ ∅ then
8  |  |  return ⟨MAYBE, F⟩
9  |  else return ⟨YES, F⟩
```

Procedure TerminationCFG traverses the *Strongly Connected Component*s (SCCs) of the CFG of $\mathcal{T}$ and attempts to prove termination or non-termination of each SCC S by calling TerminationSCC. We may assume that only non-trivial SCCs, i.e., those with at least one edge, are handled as others are trivially terminating. Let us first describe the possible outcomes of calling TerminationSCC$(S, \mathcal{T})$ to analyse the termination behaviour of S:

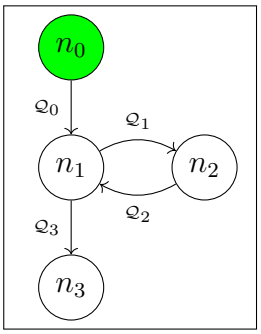
- $\langle \texttt{NO}, F_S \rangle$: in this case, procedure TerminationSCC succeeds to prove that S is non-terminating, and that, moreover, there is a valid execution that starts in the initial node $\texttt{n}_0$ of $\mathcal{T}$ and reaches S – this is why we pass $\mathcal{T}$ as well. The second component F_S is supposed to be a witness for non-termination in this case (the details are given in Section 2.3.2).

- $\langle \texttt{YES}, \emptyset \rangle$: in this case, procedure TerminationSCC succeeds to prove that S is terminating. Note that the second component is $\emptyset$ in this case. The parameter $\mathcal{T}$ might be used to understand how S is reached from the initial node $\texttt{n}_0$, to infer invariants, etc.

- $\langle \texttt{MAYBE}, F_S \rangle$: in this case, procedure TerminationSCC did not succeed to prove termination or non-termination of S, however, it might have succeeded to prove that some of the edges (or individual transitions) in S cannot be taken infinitely often, and, in such case, the second component F_S is supposed to be the set of all other edges (or reduced edges, i.e., after removing terminating transitions from corresponding transition polyhedra). This set is used in Chapter 3 to apply program transformations to those parts of S.

The output of TerminationCFG is similar to TerminationSCC, except in the case of `MAYBE` where it returns the set of all edges (i.e, among all SCCs) that it could not prove terminating (they are accumulated at Line 6).

In the next sections, we describe some common approaches used to implement procedure TerminationSCC, both for termination and non-termination analysis. In all discussions and examples, we avoid technical details that are related to the domain of variables ($\mathbb{Z}$ or $\mathbb{Q}$). Technically, the domain might affect the complexity (and in some case completeness) of the different techniques described below. Briefly, when variables range over $\mathbb{Q}$, the complexity is usually polynomial, but when they range over the $\mathbb{Z}$ it is usually exponential since it requires computing the integer-hull of the transition polyhedra of all



$$\begin{aligned}
\mathcal{Q}_0 \equiv\{ & \quad x' = x & \} \\
\mathcal{Q}_1 \equiv\{ & x \geq 0, \quad x' = x & \} \\
\mathcal{Q}_2 \equiv\{ & \quad x' \leq x - 1 & \} \\
\mathcal{Q}_3 \equiv\{ & x \leq -1, \quad x' = x & \}
\end{aligned}$$

Figure 2.3: A loop that has a linear ranking function.

edges [Ben-Amram and Genaim 2014; Ben-Amram and Genaim 2017]. In what follows, when we state complexity results we refer to the case of rational variables.

For the rest of this chapter, we fix a single SCC S that we are interested in analysing its termination behaviour, and we use E_S and N_S to denote the set of edges and nodes of S, respectively.

2.3.1 Termination Analysis Using Ranking Functions

Termination analysis is often based on the notion of *ranking functions*. These are functions that map the program states to the elements of a well-founded ordered set, such that the value descends on consecutive program states. Since descent in a well-founded set cannot be infinite, this proves that the program must terminate.

Definition 2.9 (Ranking function)**.** Let $\langle \mathcal{W}, \succ \rangle$ be a well-founded partial ordered set, we say that $\rho : N_S \times \mathbb{D}^n \mapsto \mathcal{W}$ is a ranking function for SCC S, if for every $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ and $\mathbf{x}'' \in \mathcal{Q}$ we have

$$\rho(\mathtt{n_s}, \mathbf{x}) \succ \rho(\mathtt{n_t}, \mathbf{x}') \tag{2.4}$$

Namely, function ρ decreases when applied to consecutive states $(\mathtt{n_s}, \mathbf{x})$ and $(\mathtt{s_t}, \mathbf{x}')$.

In what follows, for simplicity, we abuse notation and write $\rho_{\mathtt{n}}(\mathbf{x})$ instead of $\rho(\mathtt{n}, \mathbf{x})$, and, moreover, we treat $\rho_{\mathtt{n}}$ as a function from $\mathbb{D}^n$ to $\mathcal{W}$. We also abuse terminology and refer to each $\rho_{\mathtt{n}}$, as well as to the set of all such functions, as a ranking function.

EXAMPLE 2.10. Consider the TS depicted in Figure 2.3, in particular the SCC formed by nodes $\mathtt{n_1}$ and $\mathtt{n_2}$. This SCC has a ranking function defined by

$$\begin{aligned}
\rho_{\mathtt{n_1}}(x) &= 2x + 1 \\
\rho_{\mathtt{n_2}}(x) &= 2x
\end{aligned}$$

because $\rho_{\mathtt{n_1}}(x) - \rho_{\mathtt{n_2}}(x') \geq 1$ holds for the edge of $\mathcal{Q}_1$ and $\rho_{\mathtt{n_2}}(x) - \rho_{\mathtt{n_1}}(x') \geq 1$ holds for the edge of $\mathcal{Q}_2$. This proves termination since $\rho_{\mathtt{n_1}}(x) \geq 0$ holds for any enabled state of $\mathcal{Q}_1$, and $\rho_{\mathtt{n_2}}(x) \geq 0$ holds for any enabled state of $\mathcal{Q}_2$ (here we make use of an invariant $x \geq 0$ for $\mathtt{n_2}$). This example also demonstrates the importance of using different functions for the different nodes, since there is no ranking function for this SCC in which both nodes have the same *linear* function. □

The ranking functions used in the example above are called LRFs since they have a linear form $\rho_{\mathtt{n}}(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$. Formally, a set $\{\rho_{\mathtt{n}} : \mathbb{D}^n \mapsto \mathbb{Q} \mid \mathtt{n} \in S\}$ of linear functions is a LRF if the following conditions hold[1] for any $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ and $\mathbf{x}'' \in \mathcal{Q}$

$$\rho_{\mathtt{n_s}}(\mathbf{x}) \geq 0\,, \tag{2.5}$$

$$\rho_{\mathtt{n_s}}(\mathbf{x}) - \rho_{\mathtt{n_t}}(\mathbf{x}') \geq 1\,. \tag{2.6}$$

LRFs are very common in termination analysers since they have polynomial-time algorithms that are easy to implement [Alias et al. 2010; Ben-Amram and Genaim 2013; Colón and Sipma 2001; Feautrier 1992; Mesnard and Serebrenik 2008; Podelski and Rybalchenko 2004; Sohn and Gelder 1991]. Moreover, these algorithms are complete in the sense that if there is a LRF for S, they will find one.

As the reader might have noticed, the functions used in conditions (2.5-2.6) might not strictly satisfy the conditions of a ranking function as stated in Definition 2.9. This is because the range of ρ, in this case, is the set of rationals $\mathbb{Q}$ (or integers $\mathbb{Z}$), which is not well-founded. This is the case of the ranking function we used in Example 2.10 because $\rho_{\mathtt{n}_1}(x')$ can be arbitrarily negative on $\binom{x}{x'} \in \mathcal{Q}_2$. This, however, is just a technical detail that can be solved as follow: each $\rho_{\mathtt{n}}$ is lifted to $\tilde{\rho}_n(\mathbf{x}) = \max(0, \rho_{\mathtt{n}}(\mathbf{x}) + 1)$, whose range is the non-negative subset of $\mathbb{Q}$ which is well-founded when using the order $a \succ b$ iff $a \geq b + 1$; now the set of all $\tilde{\rho}_{\mathtt{n}}$ is a ranking function as in Definition 2.9. This technical issue arises in all types of ranking functions that we use in this thesis, and always can be solved in a similar way. Thus, we will abuse terminology and call them ranking functions and avoid lifting them as we just explained.

LRFs do not suffice for all SCCs, and in such case *lexicographic ranking functions* is a natural extension. A lexicographic ranking function for S consists of a tuple $\tau_{\mathtt{n}} = \langle \rho_{1,\mathtt{n}}, \ldots, \rho_{k_{\mathtt{n}},\mathtt{n}} \rangle$ for each node $\mathtt{n} \in N_S$, where each $\rho_{i,\mathtt{n}}$ is a function from $\mathbb{D}^n$ to a well-founded ordered set $\mathcal{W}$, such that for any $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ and $\mathbf{x}'' \in \mathcal{Q}$ there exists $0 < i \leq k_{\mathtt{n_s}}$ for which the following conditions hold

$$\rho_{i,\mathtt{n_s}}(\mathbf{x}) \succ \rho_{i,\mathtt{n_t}}(\mathbf{x}') \tag{2.7}$$

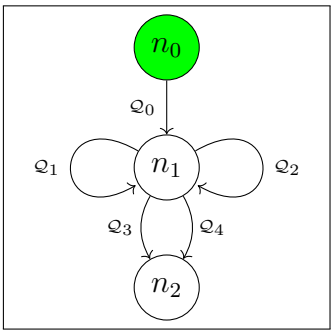
$$\forall j < i.\ \ \rho_{j,\mathtt{n_s}}(\mathbf{x}) = \rho_{j,\mathtt{n_t}}(\mathbf{x}') \tag{2.8}$$

This means that the ith component decreases for $\mathbf{x}''$, while all components to its left do not change their value. The tuples $\tau_{\mathtt{n}}$ can be used to define a ranking function as in Definition 2.9, simply by using a lexicographic extension of $\mathcal{W}$ which is well-founded.

EXAMPLE 2.11. Consider the TS depicted in Figure 2.4. The SCC formed by $\mathtt{n}_1$ does not have a LRF, but it has a lexicographic ranking function $\tau_{\mathtt{n}_1} = \langle x, y \rangle$: for the edge of $\mathcal{Q}_1$ the first component decreases, and for the edge of $\mathcal{Q}_2$ the second component decreases while the first does not change its value. □

The above example shows that lexicographic ranking functions are important in practice, however, seeking them *directly* can be technically challenging since, among other things, it might require solving non-linear constraints [Leike and Heizmann 2015]. Thus, instead of seeking them directly, there are lightweight approaches that construct them incrementally using the notion of *quasi-ranking function*.

[1] The constant 1 in (2.6) can be replaced by any fixed rational constant $\delta > 0$.



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & & x' = x, & y' = y & \} \\
\mathcal{Q}_1 \equiv\{ & x \geq 0, y \geq 0, & x' = x - 1, & y' = y & \} \\
\mathcal{Q}_2 \equiv\{ & x \geq 0, y \geq 0, & x' = x, & y' = y - 1 & \} \\
\mathcal{Q}_3 \equiv\{ & x \leq -1, & x' = x, & y' = y & \} \\
\mathcal{Q}_4 \equiv\{ & y \leq -1, & x' = x, & y' = y & \}
\end{array}
$$

Figure 2.4: A loop that has a lexicographic ranking function, but not a linear ranking function.

Definition 2.12 (Quasi-ranking function)**.** Let $\langle \mathcal{W}, \succ \rangle$ be a well-founded partial ordered set. We say that $\rho : N_S \times \mathbb{D}^n \mapsto \mathcal{W}$ is a quasi-ranking function for SCC S, if for any $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ and $\mathbf{x}'' \in \mathcal{Q}$ Condition (2.9) holds, and for at least one such $\mathbf{x}''$ Condition (2.10) holds as well

$$\rho(\mathtt{n_s}, \mathbf{x}) \succeq \rho(\mathtt{n_t}, \mathbf{x}') \tag{2.9}$$
$$\rho(\mathtt{n_s}, \mathbf{x}) \succ \rho(\mathtt{n_t}, \mathbf{x}') \tag{2.10}$$

Transitions that satisfy (2.10) are said to be ranked by ρ. The relation $a \succeq b$ is defined[2] as $a \succ b \vee a = b$.

A quasi-ranking function by itself does not imply termination, all it guarantees is that transitions satisfying (2.10) cannot be used infinitely often in a single trace. However, they can be used to incrementally construct a lexicographic ranking function by repeating the following actions

1. Find a quasi-ranking function ρ for S.
2. Remove all $\mathbf{x}''$ that satisfy (2.10) from S.

These actions should be repeated until all transitions are eliminated from S, or no progress is made, i.e., no quasi-ranking function is found at Step 1. If all transitions are eliminated, the quasi-ranking functions inferred in Step 1 form a lexicographic ranking function (when ordered from left to right). Note that during the iterations, S might be split into several non-trivial (sub) SCCs that are probably connected (since complete edges are removed), in such case it is also possible to seek a quasi-ranking function for each (sub) SCC separately.

The termination of this iterative process depends on the choice of ρ in each iteration, however, if there is a lexicographic ranking function for S then there is a choice of ρ that guarantees termination of this process. In what follows, we refer to this iterative process as the *incremental algorithm*, and, like the case of ranking functions, we will use $\rho_{\mathtt{n}}(\mathbf{x})$ instead of $\rho(\mathtt{n}, \mathbf{x})$ for quasi-ranking functions.

Termination analysers usually use quasi-*linear* ranking functions (QLRFs), i.e., each $\rho_{\mathtt{n}}$ has the form $\rho_{\mathtt{n}}(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$, and thus they infer LLRFs. This restriction allows inferring QLRFs using polynomial-time linear programming techniques. Different kinds

[2] We could use $=$ instead of $\succeq$ as well.

of QLRFs were suggested[3] in the literature [Alias et al. 2010; Ben-Amram and Genaim 2014; Bradley et al. 2005a; Larraz et al. 2013] and they have different power, i.e., some can prove termination of programs that others cannot. The main difference between these notions of QLRFs is on how they classify transitions to be *ranked* (i.e., satisfy (2.10)) and to be non-increasing (i.e., satisfy (2.9)). In what follows, we discuss these types of QLRFs and compare their relative power, ignoring differences such as simultaneous inference of invariants as done by Bradley et al. [2005a] and Larraz et al. [2013].

Alias et al. [2010] use a definition of QLRFs that requires[4]: (*i*) there is least one edge $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ such that for all $\mathbf{x}'' \in \mathcal{Q}$ conditions (2.11,2.13) hold; and (*ii*) for any other $\mathbf{x}''$, that comes from other edges, conditions (2.11,2.12) hold.

$$\rho_{\mathtt{n_s}}(\mathbf{x}) \geq 0\,, \tag{2.11}$$

$$\rho_{\mathtt{n_s}}(\mathbf{x}) - \rho_{\mathtt{n_t}}(\mathbf{x}') \geq 0\,. \tag{2.12}$$

$$\rho_{\mathtt{n_s}}(\mathbf{x}) - \rho_{\mathtt{n_t}}(\mathbf{x}') \geq 1\,. \tag{2.13}$$

Such QLRFs eliminate at least one edge in each iteration of the incremental algorithm, and they can be synthesised in polynomial time using linear programming techniques. Similarly to the case of LRFs, these QLRFs can be lifted to $\max(0, \rho_{\mathtt{n}}(\mathbf{x}) + 1)$ to satisfy the conditions of Definition 2.12.

EXAMPLE 2.13. The LLRF of Example 2.11 is built using the incremental algorithm and QLRFs of Alias et al. [2010] as follows: In the first iteration, it infers $\rho_{1,\mathtt{n}_1}(x, y) = \rho_{1,\mathtt{n}_2}(x, y) = x$ which eliminates the edge of $\mathcal{Q}_1$; and in the second iteration, it infers $\rho_{2,\mathtt{n}_1}(x, y) = \rho_{2,\mathtt{n}_2}(x, y) = y$ which eliminates the edge of $\mathcal{Q}_2$. □

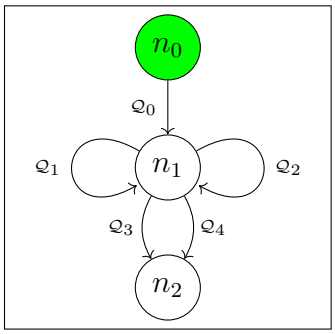
An important aspect of the algorithm of Alias et al. [2010] for inferring QLRFs, is that it guarantees the completeness of the incremental algorithm, i.e., if there is a LLRF (according to their restricted definition) then it will find one, and if there is no LLRF the algorithm will report so. Moreover, the inferred LLRF has optimal depth as well (minimal number of components, a.k.a. minimal length), this is because their algorithm infers a QLRF that eliminates the maximum possible number of edges in each iteration. Besides, the depth of the LLRF is bounded by the number of variables, which induces a bound on the number of iterations of the incremental algorithm.

Bradley et al. [2005a] use a definition of QLRFs that is weaker (in the sense of more general) than Alias et al. [2010]: their condition for the non-increasing transitions requires only (2.12) instead of (2.11,2.12). Such QLRFs eliminate at least one edge in each iteration of the incremental algorithm, and they can be synthesised in polynomial time using linear programming techniques. However, while the incremental algorithm it is still complete for this kind of QLRFs, it is not guaranteed to produces LLRFs of optimal depth since the notion of ranking the maximal number of edges does not apply in this case [Ben-Amram and Genaim 2015]. The depth of the LLRF is bounded by the number of edges in this case.

EXAMPLE 2.14. Consider the TS of Figure 2.5, and let us apply the incremental algorithm to the SCC of node $\mathtt{n}_1$ using the QLRFs of Bradley et al. [2005a]: In the first

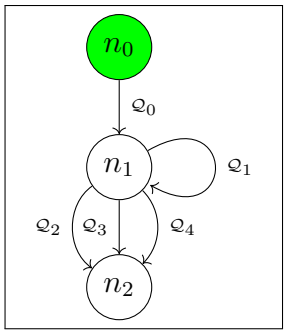
[3]Some authors do not directly use the term quasi-ranking function, however, they are induced by their definition of LLRF.

[4]They actually require (2.11) to hold for all transitions, even if they were eliminated by other QLRFs in previous iterations. This requirements is not important for our purposes.



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & & x' = x, & y' = y & \} \\
\mathcal{Q}_1 \equiv\{ & x \geq 0, & x' = x - 1, & y' = y & \} \\
\mathcal{Q}_2 \equiv\{ & y \geq 0, & y' = y - 1, & x' \leq x & \} \\
\mathcal{Q}_3 \equiv\{ & x \leq -1, & x' = x, & y' = y & \} \\
\mathcal{Q}_4 \equiv\{ & y \leq -1, & x' = x, & y' = y & \}
\end{array}
$$

Figure 2.5: A loop that has a LLRF according to the definition of Bradley et al. [2005a].



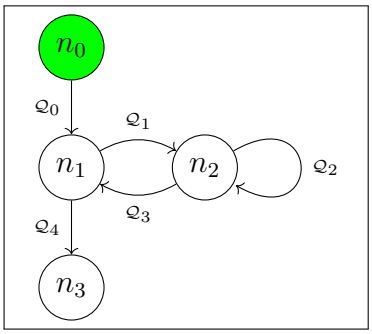
$$
\begin{array}{lll}
\mathcal{Q}_0 \equiv\{ & & x' = x, y' = y, z' = z \quad \} \\
\mathcal{Q}_1 \equiv\{ & x \geq 0, y \geq 0, x + z \geq 0, & x' = x, y' = y - x, z' = z + x - 2 \quad \} \\
\mathcal{Q}_2 \equiv\{ & x \leq -1, & x' = x, y' = y, z' = z \quad \} \\
\mathcal{Q}_3 \equiv\{ & y \leq -1, & x' = x, y' = y, z' = z \quad \} \\
\mathcal{Q}_4 \equiv\{ & x + z \leq -1, & x' = x, y' = y, z' = z \quad \}
\end{array}
$$

Figure 2.6: A loop that has a LLRF according to the definition of Ben-Amram and Genaim [2014].

iteration we use the QLRF $\rho_{1,\mathtt{n}_1}(x, y) = x$ which eliminates the edge of $\mathcal{Q}_1$; and in the second iteration we use the QLRF $\rho_{2,\mathtt{n}_1}(x, y) = y$ which eliminates the edge of $\mathcal{Q}_2$. This induces the LLRF $\tau_{\mathtt{n}_1} = \langle x, y \rangle$. Note that $\rho_{1,\mathtt{n}_1}$ is not a QLRF according to Alias et al. [2010] since it does not satisfy (2.11) for transitions of $\mathcal{Q}_2$, and thus using their definition we would fail to prove termination. □

Ben-Amram and Genaim [2014] suggested another definition for QLRFs that is more general than Alias et al. [2010] and is not comparable to Bradley et al. [2005a]. Technically, they require (2.11,2.13) to hold for at least one $\mathbf{x}'' \in \mathcal{Q}$ of some edge $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$, and require (2.11,2.12) to hold for the rest of transitions, i.e., unlike Alias et al. [2010] and Bradley et al. [2005a], they do not require a complete edge to be ranked. They suggest a polynomial-time algorithm that guarantees the completeness of the incremental algorithm and produces a LLRF of optimal depth since it infers a QLRF that eliminate the maximum possible number of transitions in each iteration. The depth of the LLRF is bounded by the number of variables.

EXAMPLE 2.15. Consider the TS of Figure 2.6, and let us apply the incremental algorithm to the SCC of node $\mathtt{n}_1$ using the QLRFs of Ben-Amram and Genaim [2014]: In the first iteration we use the QLRF $\rho_{1,\mathtt{n}_1}(x, y, z) = y$ which eliminates transitions of $\mathcal{Q}_1$ for which $y - y' > 0$; and in the second iteration we use the QLRF $\rho_{2,\mathtt{n}_1}(x, y, z) = z$ which eliminates the rest of $\mathcal{Q}_1$. This induces the LLRF $\tau_{\mathtt{n}_1} = \langle y, z \rangle$. It is easy to see that this loop does not have a LLRF according to the definitions of Alias et al. [2010] and Bradley et al. [2005a], because for those definitions a SCC with a single edge has a LLRF iff it



$$
\begin{array}{llllll}
\mathcal{Q}_0 \equiv\{ & & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_1 \equiv\{ & y \geq 1, & x' = x - 1, & y' = y, & z' = z & \} \\
\mathcal{Q}_2 \equiv\{ & y \leq z - 1, & x' = x + 1, & y' = y, & z' = z - 1 & \} \\
\mathcal{Q}_3 \equiv\{ & y \geq z, & x' = x, & y' = y + x, & z' = z & \} \\
\mathcal{Q}_4 \equiv\{ & y \leq 0, & x' = x, & y' = y, & z' = z & \}
\end{array}
$$

Figure 2.7: A loop that has a LLRF according to the definition of Larraz et al. [2013].

has a LRF (since QLRFs eliminate complete edges).

Note that while the QLRFs of Ben-Amram and Genaim [2014] are more general than Alias et al. [2010], they are not comparable to Bradley et al. [2005a]. For example, the SCC analysed in Example 2.14 does not have a QLRF according to Ben-Amram and Genaim [2014]. □

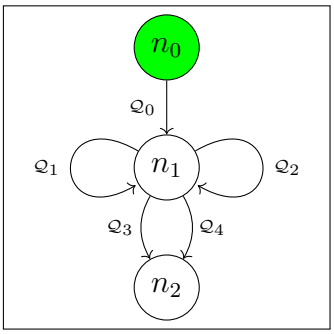
Larraz et al. [2013] provide the most general definition for QLRFs[5]. It is similar to the one of Ben-Amram and Genaim [2014] except that for the non-increasing transitions it requires only (2.12) instead of (2.11,2.12). The question whether there is an algorithm to infer such QLRFs that makes the incremental algorithm complete is still open[6]. The (incomplete) algorithm of Larraz et al. [2013] for inferring such QLRFs is based on the use of Max-SMT.

EXAMPLE 2.16. Consider the TS depicted in Figure 2.7 [Larraz et al. 2013, Fig. 1], and let us apply the incremental algorithm to the SCC of node $\mathtt{n_1}$ and $\mathtt{n_2}$ using QLRFs of Larraz et al. [2013]: In the first iteration, they use the QLRF z for both nodes which eliminates the edge of $\mathcal{Q}_2$ (with the help of an invariant $y \geq 1$ for $\mathtt{n_2}$); in the second iteration, they use the QLRF x for both nodes which eliminates transitions of $\mathcal{Q}_1$ for which $x \geq 0$; and in the third iteration, they use the QLRF y that eliminates $\mathcal{Q}_3$ (since now $x < 0$) and thus breaks the cycle so there is no need to rank the rest of $\mathcal{Q}_1$. This induces the LLRF $\langle z, x, y \rangle$ for both nodes. Note that this TS does not have a LLRF according to other definitions. □

In this thesis, we are particularly interested in a special kind of LLRFs that are called MΦRFs [Ben-Amram and Genaim 2017; Leike and Heizmann 2015]. They have not been defined using QLRFs and the incremental algorithm before. However, one can obtain such a definition by strengthening QLRFs of Larraz et al. [2013] to require non-increasing transitions to satisfy (2.13) instead of (2.12). They are called *multi-phase* since, unlike LLRFs in general, once a component of the MΦRF becomes negative it will continue decreasing and thus cannot be used anymore. This means that the different components define phases through which the execution passes as we have explained in Section 1.1. For example, the TS depicted in Figure 2.2 admits the MΦRF $\langle z, y, x \rangle$ which defines three phases: z always decreases and when it becomes negative y starts to decrease as well,

[5] Technically, they are induced by their definition of LLRF

[6] With current implementations the incremental algorithm might not terminate and thus we need to impose a bound on the number of its iterations.



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & & x' = x, & y' = y & \} \\
\mathcal{Q}_1 \equiv\{ & x \geq 0, & x' = x + y, & y' = y - 1 & \} \\
\mathcal{Q}_2 \equiv\{ & x \geq 0, y \geq 0, & x' = x, & y' = y - 1 & \} \\
\mathcal{Q}_3 \equiv\{ & x \leq -1, & x' = x, & y' = y & \} \\
\mathcal{Q}_4 \equiv\{ & y \leq -1, & x' = x, & y' = y & \}
\end{array}
$$

Figure 2.8: A loop that has a MΦRF.

and when y becomes negative x starts to decrease as well, and when x becomes negative the loop terminates.

EXAMPLE 2.17. Consider the TS depicted in Figure 2.8, and let us use the incremental algorithm with the notion of a QLRF that strengthen that of Larraz et al. [2013] as explained above: In the first iteration we use $\rho_{1,\mathtt{n}_1}(x, y) = y$ to eliminate the edge of $\mathcal{Q}_2$ and the part of $\mathcal{Q}_1$ for which $y \geq 0$; and then we use $\rho_{2,\mathtt{n}_1}(x, y) = x$ to eliminate the rest of $\mathcal{Q}_1$. This induces the MΦRF $\tau_{\mathtt{n}_1}(x, y) = \langle y, x \rangle$. □

All types of QLRFs that we have discussed above use a single linear function for each node. However, one could also use other kinds of functions as far as there are effective techniques for inferring them. In Chapter 5, we define a new kind of QLRF that assigns to each node a tuple of linear functions. It is a kind of *quasi-lexicographic linear ranking function* that generalises the notion of *poly-ranking* [Bradley et al. 2005c] and is based on what is called *nested* ranking functions [Ben-Amram and Genaim 2017; Bradley et al. 2005c; Leike and Heizmann 2015].

2.3.2 Non-termination Analysis Using Recurrent Sets

Proving non-termination of S is often based on the notion of recurrent sets. Roughly, a recurrent set is a set of states such that from any state in the set we can only make a transition to another state in the set using an edge from S.

Definition 2.18 (Recurrent Set)**.** A set of states $\Omega_S \subseteq N \times \mathbb{Q}^n$ is a recurrent set for S if, for each $(\mathtt{n}_1, \mathbf{x}) \in \Omega_S$ there is $\mathtt{n}_1 \xrightarrow{\mathcal{Q}} \mathtt{n}_2 \in E_S$ such $(\mathbf{x}, \mathbf{x}') \in \mathcal{Q}$ and $(\mathtt{n}_2, \mathbf{x}') \in \Omega_S$. When the domain of variables is $\mathbb{Z}$, we require $\Omega_S \subseteq N \times \mathbb{Z}^n$.

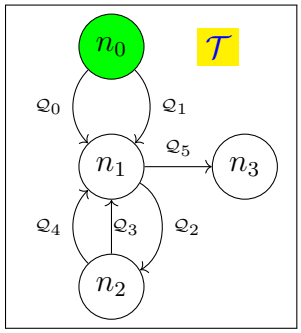
In order to prove that S is non-terminating in the context of a TS $\mathcal{T}$, in addition to finding a recurrent set Ω_S, one should also prove that at least one state $(\mathtt{n}, \mathbf{x}) \in \Omega_S$ is reachable from an initial state $(\mathtt{n}_0, \sigma_0)$.

EXAMPLE 2.19. Consider the TS of Figure 2.9, which is non-terminating for any initial state that implies $x \geq 0$: it continuously execute the *else* branch inside the loop, which corresponds to the edges of $\mathcal{Q}_2$ and $\mathcal{Q}_4$ in the TS. The SCC S formed by nodes $\mathtt{n}_1$ and $\mathtt{n}_2$ has the recurrent set

$$\Omega_S = \{(\mathtt{n}_1, (x, y)^{\mathrm{T}}) \mid x \geq 0, y \leq 0\} \cup \{(\mathtt{n}_2, (x, y)^{\mathrm{T}}) \mid x \geq 0, y \leq 0\}$$

It is reachable from any initial state that implies $x \geq 0$. □

```
1 void multi(int x, int y) {
2   if ( y >= 0 )
3     y = -1;
4   while( x >= 0 ) {
5     if ( y >= 1 )
6       x = x + y;
7     else
8       x = x - y;
9   }
10 }
```



$$
\begin{aligned}
\mathcal{Q}_0 \equiv\{&y \leq -1, \quad x' = x, \quad y' = y \quad\} \\
\mathcal{Q}_1 \equiv\{&y \geq 0, \quad x' = x, \quad y' = -1 \quad\} \\
\mathcal{Q}_2 \equiv\{&x \geq 0, \quad x' = x, \quad y' = y \quad\} \\
\mathcal{Q}_3 \equiv\{&y \geq 1, \quad x' = x + y, \quad y' = y \quad\} \\
\mathcal{Q}_4 \equiv\{&y \leq 0, \quad x' = x - y, \quad y' = y \quad\} \\
\mathcal{Q}_5 \equiv\{&x \leq -1, \quad x' = x, \quad y' = y \quad\}
\end{aligned}
$$

Figure 2.9: A non-terminating TS.

Chapter 3

Control-Flow Refinement of Transition Systems via Partial Evaluation

In Section 1.2, we have discussed the use of CFR as a technique for improving the precision of program analysis in general, and termination and cost analysis in particular. We have seen an example where CFR simplifies the termination proof in a way that allows using LRFs, instead of LLRFs that are required without CFR. This also benefits cost analysis since tools based on bounding loop iterations using LRFs would be able to infer the cost of the transformed program, while they would fail on the original one. We have also mentioned that CFR can help in inferring more precise invariants, and thus benefit any program analysis that makes use of invariants. In this chapter, we will see more examples in this direction, for which proving termination, inferring complexity bounds, or inferring precise invariants is not possible without CFR (wrt. some fixed analysis techniques).

There are CFR techniques that developed for very particular contexts [Albert et al. 2019; Flores-Montoya and Hähnle 2014; Gulwani et al. 2009; Sharma et al. 2011]. However, since CFR is, in principle, a program transformation that specialises a program to distinguish different execution scenarios, a natural question to ask is whether such specialisation can be achieved by partial evaluation, which is a general-purpose specialisation technique. In particular, the partial evaluation technique of Gallagher [2019] that specialises Horn clause programs wrt. a set of predefined properties.

Using partial evaluation for CFR has the clear advantage that soundness comes for free and that it is not tailored for a particular purpose but rather can be tuned depending on the application domain. It can almost be used in any program analysis since it generates semantically equivalent programs, while techniques developed for a specific purpose might not have this property. Moreover, developing a CFR procedure for TSs would allow using it in other tools, that analyse TSs, without any additional effort.

The rest of this chapter is organised as follows.

- In Section 3.1, we describe a CFR procedure for TSs that is based on partial evaluation and suggest heuristics for automatically inferring properties that guide specialisation.
- In Section 3.2, we discuss the use of CFR for termination analysis, by modifying Algorithm 1 to apply CFR at different levels of granularity (as a pre-processing step, only on parts that we could not prove their termination, etc).

$$
\begin{array}{lll}
q_0(x,y) \leftarrow & q_1(x,y). \\
q_1(x,y) \leftarrow & \{x \geq 0\},\ q_2(x,y). \\
q_1(x,y) \leftarrow & \{x \leq -1\}. \\
q_2(x,y) \leftarrow & \{x \geq y, y' = y+1\},\ q_1(x,y'). \\
q_2(x,y) \leftarrow & \{x \leq y-1, x' = x-1\},\ q_1(x',y).
\end{array}
$$

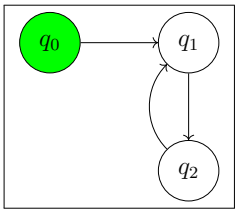


Figure 3.1: A CHC program and a corresponfing call graph.

- In Section 3.3, we discuss the use of CFR for cost analysis, as a pre-processing step, and suggest the use of ranking functions as bases for inferring properties.

Implementation and experimental evaluation are discussed in chapters 5 and 6.

3.1 Partial Evaluation of Transition Systems

In this section, we describe the partial evaluation technique of Gallagher [2019], which is developed for Horn clauses, and discuss its use for CFR of TSs.

3.1.1 Constraint Horn-Clauses

A *Constrained Horn Clause* (CHC) has the form

$$q_0(\vec{x}_0) \leftarrow \varphi, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k)$$

where q_i are predicate names (all of arity n for simplicity), $\vec{x}_i$ are tuples of variables, and φ is a formula over these variables. We call $q_0(\vec{x}_0)$ the *head* of the clause, and "$\varphi, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k)$" the *body* of the clause. In our case, formulas φ will be transition polyhedra coming from corresponding TSs. As in the case of TSs, the domain of variables is left parametric and we mention it explicitly when needed. Our techniques work both for $\mathbb{Q}$ and $\mathbb{Z}$, it is just a matter of requiring the partial evaluation of Gallagher [2019] to use SMT queries over the corresponding domain.

A CHC program is a set of CHCs such that one predicate is marked as the entry. We say that the set of CHCs with head predicate q_i defines q_i. The call graph induced by a CHC program is a directed graph whose nodes are the predicate names, and there is an edge from q_i to q_j if q_i is defined by a CHC including q_j in its body. We say that q_i is a *loop head* if it has a *back edge* in the corresponding call-graph wrt. depth-first traversal from the entry predicate.

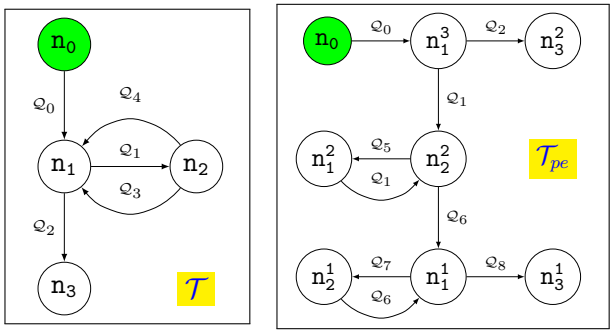
EXAMPLE 3.1. Figure 3.1 includes a CHC program and a corresponding call graph. Node q_0 is the initial node and q_1 is a loop head. □

3.1.2 From Transition Systems to Constraint Horn-Clauses

A TS $\mathcal{T}$ can be translated into a semantically equivalent[1] CHC program $\texttt{HC}(\mathcal{T})$ by translating each edge $\texttt{n}_\texttt{i} \xrightarrow{\mathcal{Q}} \texttt{n}_\texttt{j}$ into a CHC "$q_{\texttt{n}_\texttt{i}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\texttt{n}_\texttt{j}}(\vec{x}')$", and marking $q_{\texttt{n}_0}$ as the entry predicate. The CFG of $\mathcal{T}$ is equivalent to the call-graph of $\texttt{HC}(\mathcal{T})$. Observe that

[1] CHC programs are basically constraint logic programs, so corresponding semantics can be used.

```
1 void phases1(int x, int y, int z) {
2   while( x >= 1 ){
3     if ( y <= z - 1 ) {
4       y = y + 1;
5     } else {
6       x = x - 1;
7     }
8   }
9 }
```



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & & x' = x, & y' = y, & z' = z \ \} \\
\mathcal{Q}_1 \equiv\{x \geq 1, & & x' = x, & y' = y, & z' = z \ \} \\
\mathcal{Q}_2 \equiv\{x \leq 0, & & x' = x, & y' = y, & z' = z \ \} \\
\mathcal{Q}_3 \equiv\{y \leq z - 1, & & x' = x, & y' = y + 1, & z' = z \ \} \\
\mathcal{Q}_4 \equiv\{y \geq z, & & x' = x - 1, & y' = y, & z' = z \ \} \\
\mathcal{Q}_5 \equiv\{x \geq 1, & y \leq z - 1, & x' = x, & y' = y + 1, & z' = z \ \} \\
\mathcal{Q}_6 \equiv\{x \geq 1, & y \geq z, & x' = x - 1, & y' = y, & z' = z \ \} \\
\mathcal{Q}_7 \equiv\{x \geq 1, & y \geq z, & x' = x, & y' = y, & z' = z \ \} \\
\mathcal{Q}_8 \equiv\{x \leq 0, & y \geq z, & x' = x, & y' = y, & z' = z \ \}
\end{array}
$$

Figure 3.2: A loop with 2 phases.

$\mathtt{HC}(\mathcal{T})$ is linear, that is, each clause has only one call in the body. Linearity guarantees that the specialised version is also linear, and thus can be converted back into a TS $\mathcal{T}_{pe}$ in a similar way. Soundness of partial evaluation guarantees equivalence between the traces of $\mathcal{T}$ and $\mathcal{T}_{pe}$, in particular, there is an infinite trace in $\mathcal{T}$ iff there is one in $\mathcal{T}_{pe}$. The complexity is preserved as well when modelled using an auxiliary variable as discussed in Section 2.2.

EXAMPLE 3.2. Consider the method `phases1` of Figure 3.2 and its corresponding TS $\mathcal{T}$. Translating this TS into a CHC program results in:

$$
\begin{array}{lllll}
q_{\mathtt{n}_0}(x,y,z) \leftarrow & q_{\mathtt{n}_1}(x,y,z). & & & \\
q_{\mathtt{n}_1}(x,y,z) \leftarrow & \{x \geq 1\}, & q_{\mathtt{n}_2}(x,y,z). & & \\
q_{\mathtt{n}_1}(x,y,z) \leftarrow & \{x \leq 0\}, & q_{\mathtt{n}_3}(x,y,z). & & \\
q_{\mathtt{n}_2}(x,y,z) \leftarrow & \{y \leq z-1, & y' = y+1\}, & q_{\mathtt{n}_1}(x,y',z). & \\
q_{\mathtt{n}_2}(x,y,z) \leftarrow & \{y \geq z, & x' = x-1\}, & q_{\mathtt{n}_1}(x',y,z). &
\end{array}
$$

The only loop head predicate is $q_{\mathtt{n}_1}$ ($\mathtt{n}_1$ since it has a back edge when traversing $\mathcal{T}$ starting at node $\mathtt{n}_0$. □

3.1.3 Partial Evaluation of Constraint Horn-Clauses

The technique of Gallagher [2019] takes as input a CHC program and a constraint on the entry predicate, and returns a partially evaluated CHC program that preserves the computational behaviour of the original program with the given entry. In particular, it preserves termination and complexity of the original program. In our case, the initial condition of the entry will be *true*. It is an online partial evaluation algorithm, meaning that control decisions are made on the fly, and it yields a *poly-variant* partially evaluated program, meaning that a single predicate q_i from the input program can result in several "versions" of q_i in the output program.

It is not possible to create versions for all reachable possibilities as there may be infinitely many. During partial evaluation, when reaching a call $q(\vec{x})$ with some context φ (i.e., some constraints on variables $\vec{x}$) the algorithm decides whether to create a version of $q(\vec{x})$ for this context, and replace the corresponding call by one to this version, or to *abstract* the context.

Gallagher [2019] suggests the use of property-based abstraction in order to guarantee the generation of a finite number of versions. For a predicate $q(\vec{x})$, it assigns beforehand a finite set of properties $\{\gamma_1, \ldots, \gamma_k\}$, where each γ_i is a formula over $\vec{x}$, and when reaching $q(\vec{x})$ with a context φ, instead of creating a version for φ it creates one for

$$\alpha(\varphi) = \wedge\{\gamma_i \mid \varphi \models \gamma_i\} \tag{3.1}$$

where $\models$ stands for logical implication. This guarantees that there are at most 2^k versions for $q(\vec{x})$.

EXAMPLE 3.3. Suppose a predicate $f(x, y, z)$ is assigned properties $x \geq 1$ and $y \geq z$, it would then have up to 4 versions corresponding to (abstract) contexts $x \geq 1$, $y \geq z$, $x \geq 1 \wedge y \geq z$ and *true*. □

In practice, to guarantee termination of partial evaluation, it is enough to apply property-based abstraction to loop head predicates as they cut all cycles; other predicates can be specialised wrt. to all contexts that are encountered during the evaluation. In addition, partial evaluation might unfold deterministic sequences of calls into a single clause. Next, we describe the partial evaluation algorithm for CHC programs using a worked example, assuming, for simplicity, that the clauses are linear since they are generated from a TS. A more detailed presentation of the algorithm for arbitrary clauses can be found in Gallagher [2019].

Consider the CHC program of Example 3.2, that corresponds to TS $\mathcal{T}$ of Figure 3.2. The execution of this TS has two implicit phases:

- in the first one, `y` is incremented until it reaches the value of `z`; and
- in the second phase `x` is decremented until it reaches `0`.

Our aim is to use partial evaluation to split the loop into two explicit phases as we have done in Section 1.2, in particular to automatically obtain $\mathcal{T}_{pe}$ of Figure 3.2. This would allow proving termination of $\mathcal{T}_{pe}$, and thus $\mathcal{T}$, using LRFs only.

The first step is to assign properties to the different predicates. Let us manually fix the properties of $q_{\mathtt{n}_1}(x, y, z)$ to be

$$\Gamma = \{x \geq 1, y \geq z\} \tag{3.2}$$

which are the properties of the variables at loop entry that determine which loop path or loop exit is taken. Note that this set does not represent a conjunction, its elements are separate constraints that represent properties. In Section 3.1.4 we discuss how these properties are inferred automatically.

The partial evaluation algorithm performs a sequence of iterations. The input to each iteration is a set of "versions" of predicates, where each version is a pair $\langle q(\vec{x}), \varphi \rangle$, where q is a program predicate and φ is a constraint on its arguments. Each iteration performs an *unfolding* for each version, followed by a property-based *abstraction* of the resulting calls using Γ, yielding a new set of versions that is input to the following iteration. The iterations end when no new versions are produced by an unfolding. The unfold operation for a version $\langle q(\vec{x}), \varphi \rangle$ consists of expanding calls in the body of clauses with head $q(\vec{x})$, so long as the call is deterministic and not a loop head, and the constraint φ is satisfied.

Let the initial version be $\langle q_{\mathtt{n}_0}(x, y, z), \mathit{true} \rangle$. Then, the iterations of the algorithm yield the following sequence of new versions:

1. $\{\langle q_{\mathtt{n}_1}(x, y, z), \mathit{true} \rangle\}$.
2. $\{\langle q_{\mathtt{n}_2}(x, y, z), x \geq 1 \rangle, \langle q_{\mathtt{n}_3}(x, y, z), x \leq 0 \rangle\}$.
3. $\{\langle q_{\mathtt{n}_1}(x, y, z), x \geq 1 \rangle, \langle q_{\mathtt{n}_1}(x, y, z), y \geq z \rangle\}$.
4. $\{\langle q_{\mathtt{n}_2}(x, y, z), x \geq 1, y \geq z \rangle, \langle q_{\mathtt{n}_3}(x, y, z), x \leq 0, y \geq z \rangle\}\}$.
5. $\emptyset$.

We look in detail at iteration 3; the versions $\langle q_{\mathtt{n}_2}(x, y, z), x \geq 1 \rangle$ and $\langle q_{\mathtt{n}_3}(x, y, z), x \leq 0 \rangle$ resulting from iteration 2 are unfolded. The latter yields no clauses since there is no clause with head predicate $q_{\mathtt{n}_3}$. The version $\langle q_{\mathtt{n}_2}(x, y, z), x \geq 1 \rangle$ is unfolded giving the clauses:

$$\begin{aligned} q_{\mathtt{n}_2}(x, y, z) \leftarrow \quad & \{x \geq 1, \quad y \leq z - 1, \quad y' = y + 1\}, \quad q_{\mathtt{n}_1}(x, y', z) \\ q_{\mathtt{n}_2}(x, y, z) \leftarrow \quad & \{x \geq 1, \quad y \geq z, \qquad x' = x - 1\}, \quad q_{\mathtt{n}_1}(x', y, z). \end{aligned}$$

The constraints on the body calls are then collected. The constraints on the first call $q_{\mathtt{n}_1}(x, y', z)$, projected onto $\{x, y', z\}$, are $x \geq 1$. The constraints on the second call $q_{\mathtt{n}_1}(x', y, z)$, projected onto $\{x', y, z\}$, are $x' \geq 0 \wedge y \geq z$. Using property-based abstraction with Γ, the set of properties for $q_{\mathtt{n}_1}$, we obtain (after renaming x' to x)

$$\begin{aligned} \alpha(x \geq 1) &= x \geq 1 \\ \alpha(x \geq 0 \wedge y \geq z) &= y \geq z. \end{aligned}$$

Note that the constraint $x \geq 0 \wedge y \geq z$ entails $y \geq z$ but no other member of Γ and hence the constraint $x \geq 0$ is abstracted away. This yields the two versions $\langle q_{\mathtt{n}_1}(x, y, z), x \geq 1 \rangle$ and $\langle q_{\mathtt{n}_1}(x, y, z), y \geq z \rangle$ that are shown as the result of iteration 3. Note that without abstraction, an infinite number of iterations could result. In this example, versions $\langle q_{\mathtt{n}_1}(x, y, z), x \geq 0, y \geq z \rangle, \langle q_{\mathtt{n}_1}(x, y, z), x \geq -1, y \geq z \rangle, \ldots$ would be generated.

For each version, a new predicate is generated. We use predicate names that carry information on the different versions: $q_{\mathtt{n}_\mathtt{i}^\mathtt{j}}$ is the jth version of predicate $q_{\mathtt{n}_\mathtt{i}}$.The entry

predicate $q_{\mathtt{n}_0}$ is not renamed. For our example a suitable renaming is as follows.

$$\begin{array}{lcl}
\langle q_{\mathtt{n}_1}(x,y,z), \mathit{true}\rangle & \Rightarrow & q_{\mathtt{n}_1^3}(x,y,z)\\
\langle q_{\mathtt{n}_2}(x,y,z), x \geq 1\rangle & \Rightarrow & q_{\mathtt{n}_2^2}(x,y,z)\\
\langle q_{\mathtt{n}_1}(x,y,z), x \geq 1\rangle & \Rightarrow & q_{\mathtt{n}_1^2}(x,y,z)\\
\langle q_{\mathtt{n}_2}(x,y,z), x \geq 1 \wedge y \geq z\rangle & \Rightarrow & q_{\mathtt{n}_2^1}(x,y,z)\\
\langle q_{\mathtt{n}_1}(x,y,z), y \geq z\rangle & \Rightarrow & q_{\mathtt{n}_1^1}(x,y,z)\\
\langle q_{\mathtt{n}_3}(x,y,z), x \leq 0\rangle & \Rightarrow & q_{\mathtt{n}_3^2}(x,y,z)\\
\langle q_{\mathtt{n}_3}(x,y,z), x \leq 0, y \geq z\rangle & \Rightarrow & q_{\mathtt{n}_3^1}(x,y,z)
\end{array}$$

The head and body calls of the unfolded clauses for each version are then renamed, yielding the following set of clauses as the result of partial evaluation:

$$\begin{array}{lllll}
q_{\mathtt{n}_0}(x,y,z) \leftarrow & q_{\mathtt{n}_1^3}(x,y,z). & & & \\
q_{\mathtt{n}_1^3}(x,y,z) \leftarrow & \{x \leq 0\}, & q_{\mathtt{n}_3^2}(x,y,z). & & \\
q_{\mathtt{n}_1^3}(x,y,z) \leftarrow & \{x \geq 1\}, & q_{\mathtt{n}_2^2}(x,y,z). & & \\
q_{\mathtt{n}_2^2}(x,y,z) \leftarrow & \{x \geq 1, & y \leq z-1, & y' = y+1\}, & q_{\mathtt{n}_1^2}(x,y',z).\\
q_{\mathtt{n}_2^2}(x,y,z) \leftarrow & \{x \geq 1, & y \geq z, & x' = x-1\}, & q_{\mathtt{n}_1^1}(x',y,z).\\
q_{\mathtt{n}_1^2}(x,y,z) \leftarrow & \{x \geq 1\}, & q_{\mathtt{n}_2^2}(x,y,z). & & \\
q_{\mathtt{n}_1^1}(x,y,z) \leftarrow & \{x \leq 0, & y \geq z\}, & q_{\mathtt{n}_3^1}(x,y,z). & \\
q_{\mathtt{n}_1^1}(x,y,z) \leftarrow & \{x \geq 1, & y \geq z\}, & q_{\mathtt{n}_2^1}(x,y,z). & \\
q_{\mathtt{n}_2^1}(x,y,z) \leftarrow & \{x \geq 1, & y \geq z, & x' = x-1\}, & q_{\mathtt{n}_1^1}(x',y,z).
\end{array}$$

Translating this CHC program back to an TS results in $\mathcal{T}_{pe}$, depicted in Figure 3.2.

3.1.4 Choice of Properties

In the example above, the properties were chosen manually to give the appropriate versions and achieve CFR. While any choice of properties gives a sound partial evaluation, heuristics are typically needed to infer appropriate properties automatically, i.e., properties that lead to the desired versions.

We observe that properties relevant to CFR are closely related to the conditions in the loop head and within the loop body. This leads to the following heuristics that extract properties of $q_{\mathtt{n}_\mathtt{i}}$ from the constraints of incoming and outgoing edges of node $\mathtt{n}_\mathtt{i}$:

$$\begin{array}{rl}
\mathtt{Props}_\mathtt{h}(q_{\mathtt{n}_\mathtt{i}}) = & \{\psi \mid q_{\mathtt{n}_\mathtt{i}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\mathtt{n}_\mathtt{j}}(\vec{x}') \in \mathtt{HC}(\mathcal{T}), \psi = \mathtt{proj}_{\vec{x}}(\mathcal{Q})\}\\
\mathtt{Props}_\mathtt{hv}(q_{\mathtt{n}_\mathtt{i}}) = & \{x_\ell \diamond c \mid q_{\mathtt{n}_\mathtt{i}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\mathtt{n}_\mathtt{j}}(\vec{x}') \in \mathtt{HC}(\mathcal{T}), \ell \in [1..n], \diamond \in \{\leq, \geq\}, \mathcal{Q} \models x_\ell \diamond c\}\\
\mathtt{Props}_\mathtt{c}(q_{\mathtt{n}_\mathtt{i}}) = & \{\psi[\vec{x}'/\vec{x}] \mid q_{\mathtt{n}_\mathtt{j}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\mathtt{n}_\mathtt{i}}(\vec{x}') \in \mathtt{HC}(\mathcal{T}), \psi = \mathtt{proj}_{\vec{x}'}(\mathcal{Q})\}\\
\mathtt{Props}_\mathtt{cv}(q_{\mathtt{n}_\mathtt{i}}) = & \{x_\ell \diamond c \mid q_{\mathtt{n}_\mathtt{j}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\mathtt{n}_\mathtt{i}}(\vec{x}') \in \mathtt{HC}(\mathcal{T}), \ell \in [1..n], \diamond \in \{\leq, \geq\}, \mathcal{Q} \models x'_\ell \diamond c\}
\end{array}$$

Let us explain the different heuristics:

- $\mathtt{Props}_\mathtt{h}$ infers properties by extracting conditions from the CHCs defining $q_{\mathtt{n}_\mathtt{i}}$ (i.e., outgoing edge of $\mathtt{n}_\mathtt{i}$); this is done by projecting the corresponding constraints on $\vec{x}$. This also captures implicit conditions that do not appear syntactically in the constraints defining $\mathcal{Q}$.

- $\mathtt{Props}_\mathtt{hv}$ infers properties by extracting bounds, for each variable x_i, from the CHCs defining $q_{\mathtt{n}_\mathtt{i}}$. Inferring the minimal/maximal value for x_i is done in practice using linear programming.

- $\texttt{Props}_{\texttt{c}}$ is like $\texttt{Props}_{\texttt{h}}$, but it uses calls to $q_{\texttt{n}_\texttt{i}}$, i.e., incoming edge of $\texttt{n}_\texttt{i}$.
- $\texttt{Props}_{\texttt{cv}}$ is like $\texttt{Props}_{\texttt{hv}}$, but it uses calls to $q_{\texttt{n}_\texttt{i}}$, i.e., incoming edge of $\texttt{n}_\texttt{i}$.

Inference of properties can be done at the level of $\mathcal{T}$ or $\texttt{HC}(\mathcal{T})$. Let us see how these heuristics infer the properties that we have chosen before manually.

EXAMPLE 3.4. Let us compute properties for $\texttt{q}_{\texttt{n}_1}$. Node $\texttt{n}_1$ has outgoing edges labelled with $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and incoming edges labelled with $\mathcal{Q}_0$, $\mathcal{Q}_3$ and $\mathcal{Q}_4$. Recall that $\vec{x} = (x, y, z)$ and $\vec{x}' = (x', y', z')$. Then the corresponding properties are:

$$\begin{aligned}
\texttt{Props}_{\texttt{h}}(q_{\texttt{n}_1}) &= \{\texttt{proj}_{\vec{x}}(\mathcal{Q}_1), \texttt{proj}_{\vec{x}}(\mathcal{Q}_2)\} = \{x \geq 1, x \leq 0\} \\
\texttt{Props}_{\texttt{hv}}(q_{\texttt{n}_1}) &= \{x \geq 1, x \leq 0\} \\
\texttt{Props}_{\texttt{c}}(q_{\texttt{n}_1}) &= \{\texttt{proj}_{\vec{x}'}(\mathcal{Q}_0), \texttt{proj}_{\vec{x}'}(\mathcal{Q}_3), \texttt{proj}_{\vec{x}'}(\mathcal{Q}_4)\}[\vec{x}'/\vec{x}] = \{y \leq z, y \geq z\} \\
\texttt{Props}_{\texttt{cv}}(q_{\texttt{n}_1}) &= \emptyset.
\end{aligned}$$

Note that they are the ones used manually in (3.2). □

The above heuristics suffice for many cases in practice, but they may be inadequate when conditions in a loop body are not directly implied by the formulas of incoming/outgoing edges of the loop head. For example, assume method `phase1` of Figure 3.2 includes has an additional statement `w=w+1` at the end of the loop body. In this case, the outgoing edges of $\texttt{n}_2$ will not go directly to $\texttt{n}_1$, but to a new node connected to $\texttt{n}_1$ with the formula $\{w' = w + 1, x' = x, y' = y, z' = z\}$. Thus, the constraints $y \leq z$ and $y \geq z$ are not implied by the constraints of incoming/outgoing edges of $\texttt{n}_1$.

To overcome this limitation, we developed a new heuristic $\texttt{Props}^{\texttt{d}}_{\texttt{h}}$ that propagates all constraints in the loop bodies backwards to loop heads as follows:

1. we remove all back-edges (in $\texttt{HC}(\mathcal{T})$), that is, we replace each

   $$q_{\texttt{n}_\texttt{j}}(\vec{x}) \leftarrow \mathcal{Q}, q_{\texttt{n}_\texttt{i}}(\vec{x}') \in \texttt{HC}(\mathcal{T})$$

   by "$q_{\texttt{n}_\texttt{j}}(\vec{x}) \leftarrow \mathcal{Q}$" if $q_{\texttt{n}_\texttt{i}}$ is a loop head, which results in a recursion-free CHC program.

2. we compute the set of answers (the minimal model) of the resulting CHC program and take them as properties, i.e., if $\gamma_1, \ldots, \gamma_l$ are the answers for $q_{\texttt{n}_\texttt{i}}(\vec{x})$, we define $\texttt{Props}^{\texttt{d}}_{\texttt{h}}(q_{\texttt{n}_\texttt{i}}) = \{\gamma_1, \ldots, \gamma_l\}$.

For the modification of the program of Figure 3.2 just described, we would get

$$\texttt{Props}^{\texttt{d}}_{\texttt{h}} = \{y \leq z - 1, y \geq z, x \geq 1\}.$$

We could propagate conditions in the loop bodies to loop heads forwards as well; however, we do not do it since in practice we add invariants to all transitions (which has the effect of propagating conditions forward) before inferring properties.

From now on, we assume that we have a procedure PE that receives an TS $\mathcal{T}$ and applies CFR via partial evaluation as follows:

1. optionally, it computes invariants for all nodes, and adds them to their outgoing edges, which makes more properties visible as discussed above;

2. generates the CHC program $\texttt{HC}(\mathcal{T})$;
3. computes properties for each loop head;
4. applies partial evaluation as described above;
5. translates the specialised CHCs into a new TS;
6. optionally, computes invariants again in order to eliminate unreachable nodes in the new TS; and
7. returns the new TS.

For simplicity, we assume that the choice of properties at step 3 above is provided via global configuration, and not received as a parameter.

We might also call PE with a list of nodes N as a second argument to indicate that properties should be assigned to a loop head $q_{\texttt{n}_\texttt{i}}$ only if $\texttt{n}_\texttt{i} \in N$, this is used to apply CFR only to a subset of the TS, because predicates without properties are be specialised.

Finally, procedure `PE` can be applied iteratively on its output, which might achieve further refinements, but also risks increasing the size of the resulting TS. In practice, we have few examples that benefit from such iterative application of `PE`.

3.2 Application to Termination Analysis

In this section, we discuss how CFR via partial evaluation, as described in Section 3.1, can improve the precision of termination analysis of TSs. In particular, we present an algorithm for termination analysis incorporating the CFR procedure of Section 3.1, and discuss the different aspects of the algorithm using representative examples. Before presenting our algorithm, we briefly recall some of the details on how a typical termination analyser works using Algorithm 1 (for more details see the discussion in Section 2.3).

A termination analyser receives a TS $\mathcal{T}$, and separately proves termination or non-termination of each of its SCCs using a procedure that we called TerminationSCC. Proving termination of a SCC S is done by inferring a *ranking function*. A practical alternative is to incrementally infer *quasi-ranking functions* and then use them to construct a corresponding *lexicographic-ranking function*. In Section 2.3.1, we have seen several kinds of ranking functions and quasi-ranking functions, all are based on the use of linear functions since there are efficient algorithms for inferring them. Proving non-termination is done by inferring a corresponding recurrent set that witnesses non-termination, and proving that it is actually reachable from the initial node.

An important aspect of TerminationSCC is that when it fails to prove that a SCC S is terminating or non-terminating, it might prove that some of its edges (or individual transitions) cannot be taken infinitely. This implies that any infinite trace in S must have a suffix that uses only the other transitions, which are returned by TerminationSCC as the second component of the returned value $\langle \texttt{MAYBE}, F_S \rangle$. This is important since our CFR approach is based on refining these transitions in order to make inference of termination or non-termination witnesses possible.

As we have mentioned in Section 2.3, procedure TerminationSCC makes use of invariants as well, this increases precision since they propagate constraints between the different nodes. For simplicity, we assume that invariants are added to the input TS, either as annotations or directly to the transition polyhedra, so TerminationSCC can

make use of them, and that procedure PE of Section 3.1 adds such invariants to the refined TS as well. The inference of invariant is treated as a black-box, as it can be done using off-the-shelf tools.

The effectiveness of TerminationSCC is directly affected by the kind of ranking functions and invariants used. Some termination analysers might succeed to prove termination of a given TS while others fail, mainly because they implement different kinds of (quasi-)ranking functions. Apart from considerations like performance, some ranking functions might induce bounds on the length of traces within a SCC, which is fundamental for applications like cost analysis. For example, LRFs induce such bounds while LLRFs cannot always do so (depending on the kind of QLRF used). For invariants, analysers often use abstract domains that are based on conjunctions of linear constraints; such as convex polyhedra [Cousot and Halbwachs 1978], and Octagons [Miné 2006], however, they cannot capture more expressive, but expensive disjunctive invariants (i.e., disjunctions of conjunctions of linear constraints).

CFR splits complex control-flow into phases/cases which can improve termination analysis in several ways:

1. simplify the termination witnesses, e.g., make it possible to use LRFs where without CFR one would need LLRFs;
2. due to case splitting, it might make it possible to infer ranking functions of some kind, while without CFR it is not possible at all; and
3. improve the precision of invariants, e.g., eliminate the need for disjunctive invariants, and thus enable automatic termination and non-termination proofs where without CFR one would need disjunctive invariants.

Later we will see examples for all these scenarios.

3.2.1 Control-Flow Refinement Schemes

CFR is typically used as a pre-processing step. Although simple, this can perform unnecessary refinements and generate large TSs that are more expensive to analyse. A better approach is to apply CFR selectively; however, that requires deeper modifications to the analyser. We suggest the following schemes for adding CFR to a termination analyser, trading off ease of implementation with performance and precision:

- ($\texttt{CFR}_\texttt{B}$): in this scheme CFR is applied directly to the input TS. This is easy to implement but can perform unnecessary refinements that cause overhead in analysis.
- ($\texttt{CFR}_\texttt{S}$): in this scheme, CFR is applied at the level of SCCs, i.e., when TerminationSCC fails to prove termination of a SCC, CFR is applied only to the part of the SCC for which the termination proof has failed.
- ($\texttt{CFR}_\texttt{A}$): this scheme first collects all edges (from all SCCs) on which TerminationSCC has failed, and applies CFR to the input TS taking into account those edges only.

For $\texttt{CFR}_\texttt{S}$ and $\texttt{CFR}_\texttt{A}$ schemes, CFR can be applied iteratively so that refinement is interleaved with termination proof attempts. In this way, each step might introduce further refinements that could not be done in previous steps. The precision and performance of these schemes are compared experimentally in Chapter 6.

Algorithm 2: Pseudo-code of Termination Analysis with Control-Flow Refinement

```
    TERMINATION(T,CFR_B,CFR_A,CFR_S)
1   |  if CFR_B then T := PE(T)
2   |  ⟨Ans, F⟩ := TERMINATIONCFG(T, CFR_S)
3   |  while Ans = MAYBE and CFR_A > 0 do
4   |  |  N := nodes(F)
5   |  |  T := DELNONREACHING(T,N)
6   |  |  T := PE(T,N)
7   |  |  T' := DELTERMINATING(T,N)
8   |  |  CFR_A := CFR_A − 1
9   |  |  ⟨Ans, F⟩ := TERMINATIONCFG(T',CFR_S)
10  |  return ⟨Ans, F⟩

    TERMINATIONCFG(T,CFR_S)
11  |  F := ∅; Q := [⟨T, CFR_S⟩]
12  |  while Q ≠ ∅ do
13  |  |  ⟨T', i⟩ := Q.getFirst()
14  |  |  foreach SCC S of T' do
15  |  |  |  ⟨Ans, F_S⟩ := TERMINATIONSCC(S,T')
16  |  |  |  if Ans is NO then
17  |  |  |  |  return ⟨NO, F_S⟩
18  |  |  |  else if Ans is MAYBE and i > 0 then
19  |  |  |  |  T'' = CONITS(F_S,T')
20  |  |  |  |  T''' = PE(T'')
21  |  |  |  |  Q.add(⟨T''', i − 1⟩)
22  |  |  |  else F := F ∪ F_S
23  |  if F ≠ ∅ then
24  |  |  return ⟨MAYBE, F⟩
25  |  else return ⟨YES, ∅⟩
```

3.2.2 Incorporating CFR into a Termination Algorithm

Algorithm 2 shows the pseudo-code of a termination analysis algorithm that uses the CFR schemes discussed in the previous section. It consists of two procedures TERMINATION and TERMINATIONCFG, and uses TERMINATIONSCC as a black box. It also uses some auxiliary procedures that we explain below.

Procedure TERMINATION receives a TS $\mathcal{T}$, a Boolean $\texttt{CFR}_\texttt{B}$ indicating whether $\texttt{CFR}_\texttt{B}$ should be applied, and integers $\texttt{CFR}_\texttt{A}$ and $\texttt{CFR}_\texttt{S}$ giving the number of times that the respective schemes can be applied. At Line 1 it calls PE($\mathcal{T}$) if $\texttt{CFR}_\texttt{B}$ is *true*, and at Line 2 it calls TERMINATIONCFG to analyses the SCCs of $\mathcal{T}$ for the first time. Afterwards, it executes a while loop (lines 4-9) that alternates CFR and calls to TERMINATIONCFG as long as $\texttt{CFR}_\texttt{A}$ can be applied ($\texttt{CFR}_\texttt{A} > 0$) and (non)termination has not been proven ($Ans =$ MAYBE). Note that if at Line 2 the returned value for Ans is different from MAYBE, then the loop is not executed since there is no need for (further) CFR in such case. The return value of procedure TERMINATION (at Line 10) is basically equal to the return value of the last call to TERMINATIONCFG (either at Line 2 or at Line 9), which indicates whether it has succeeded to prove (non)termination or not.

In each iteration of the refine/analyse loop (lines 4-9) we apply CFR only to the parts corresponding to F (the edges returned by the last call to TerminationCFG, i.e., those it could not prove terminating) as follows:

1. at Line 4 it computes the set of nodes N that appear in F;

2. at Line 5 it removes from $\mathcal{T}$ nodes that do not reach nodes in N. These correspond to parts of $\mathcal{T}$ that have been proven terminating and they do not affect the CFR of any node in N;

3. at Line 6 it applies CFR considering only loop heads in N (nodes that do not correspond to loop heads are assumed to a have an empty set of properties);

4. at Line 7 it removes from $\mathcal{T}$ nodes not in N. These correspond to parts that they have been proven terminating already; and

5. at Line 8 it decreases the $\texttt{CFR}_\texttt{A}$ scheme counter.

Once CFR is applied, it calls TerminationCFG (at Line 9) to analyse $\mathcal{T}'$. Note that the nodes removed at Line 7 are not removed before at Line 5 in order to: (a) guarantee soundness, as CFR must consider all possible ways in which nodes in N are reached; (b) allow CFR to benefit from context information; and (c) allow other parts reachable from nodes in N to benefit from refinements.

Procedure TerminationCFG analyses the SCCs of $\mathcal{T}$, and applies CFR at the level of SCCs if needed. At Line 11 it initialises local variables F and Q, where F is used to accumulate the edges that it fails to prove terminating, and Q is a queue of pending (sub) TSs to be analysed. The elements of Q are pairs $\langle \mathcal{T}', i \rangle$, where $\mathcal{T}'$ is a TS to be analysed and i is the number of times left to apply CFR to its SCCs.

The *while* loop is executed as long as there are pending (sub) TSs in Q: at Line 13 it takes a TS $\mathcal{T}'$ from Q, and in the loop at lines 14-22 it analyses each SCC S. This is done by first calling TerminationSCC on S at Line 15 and depending on the result it proceeds as follows:

1. if it proves that S is non-terminating ($Ans = \texttt{NO}$), it returns a corresponding answer at Line 17;

2. if it fails ($Ans = \texttt{MAYBE}$) and $\texttt{CFR}_\texttt{S}$ can be applied ($i > 0$), then at Line 19 it builds a new TS $\mathcal{T}''$ from the problematic part F_S, by calling ConITS(F_S), and then at Line 20 it is passed to PE to apply CFR and obtain a new TS $\mathcal{T}'''$. At Line 21 $\mathcal{T}'''$ is added to Q with a corresponding $\texttt{CFR}_\texttt{S}$ counter;

3. otherwise, it adds F_S to F at Line 22 (F_S is empty if Ans is YES and non-empty if Ans is MAYBE).

Building a TS from F_S at Line 19 is done as follows: ConITS builds a TS that consists of the nodes and edges of F_S, together with a new entry node that has edges to all nodes of F_S that are reachable from nodes not in F_S. This is because CFR must consider all possible ways in which nodes of F_S are reached (we assume that ConITS has access to the original TS $\mathcal{T}'$).

3.2.3 How CFR Benefits Termination Analysis

In this section, we present case studies of how Algorithm 2 can benefit termination analysis using the different CFR schemes. For all examples, we give programs and TSs, but we omit CHCs as they are similar to TSs. TSs are sometimes simplified (e.g. by joining chains of nodes) for simplifying the presentation; however, they are very similar to what we actually get in practice.

We start with an example that demonstrates how CFR can simplify the termination witness from LLRFs to LRFs, which is useful if the underlying analyser does not use LLRFs, and can help in making cost analysis feasible as well.

EXAMPLE 3.5. Consider method `phases1` of Figure 3.2 and its corresponding TS $\mathcal{T}$. It is easy to prove that $\mathcal{T}$ terminates using the LLRF $\langle z-y, x\rangle$ for nodes $\mathtt{n}_1$ and $\mathtt{n}_2$, but it is not possible if we restrict ourselves to the use of LRFs. Using CFR (e.g., with $\mathtt{Props}_\mathtt{h}$) refines $\mathcal{T}$ in a way that makes proving termination using LRFs possible. Calling procedure Termination($\mathcal{T}$, *true*, 0, 0) applies CFR at Line 1 before trying to prove termination, yielding the TS $\mathcal{T}_{pe}$ of Figure 3.2. The two loop phases are now explicit: nodes $\mathtt{n}_1^2$ and $\mathtt{n}_2^2$ correspond to the first phase, and nodes $\mathtt{n}_1^1$ and $\mathtt{n}_2^1$ to the second phase. Using this refined TS, TerminationCFG finds LRFs $z-y$ and x for the corresponding SCCs. □

The next example shows a special case of the notion of phases, where the different phases are actually cases that are never used in the same execution.

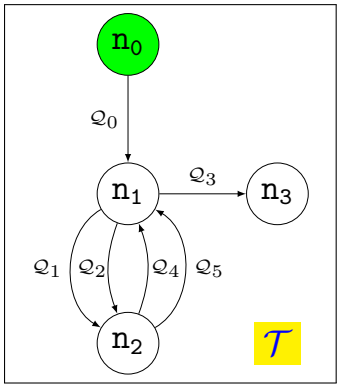
EXAMPLE 3.6. Consider the program depicted in Figure 3.3. Note that the loop guard is translated into two case: $x \leq y-1$ in $\mathcal{Q}_1$ and $x \geq y+1$ in $\mathcal{Q}_2$. The loop terminates when $x = y$ holds. It is easy to see that only one branch will be used during an execution. iRankFinder cannot handle this loop, not with LRFs nor LLRFs, and thus we resort to CFR. Applying CFR (using $\mathtt{Props}_\mathtt{c}$) both branches are separate as shown in the TS $\mathcal{T}_{pe}$ in Figure 3.3. Now iRankFinder can find LRFs: $y-x$ for the SCC of nodes $\mathtt{n}_1^2$ and $\mathtt{n}_2^2$; and $x-y$ for the SCC of nodes $\mathtt{n}_2^1$ and $\mathtt{n}_1^1$. □

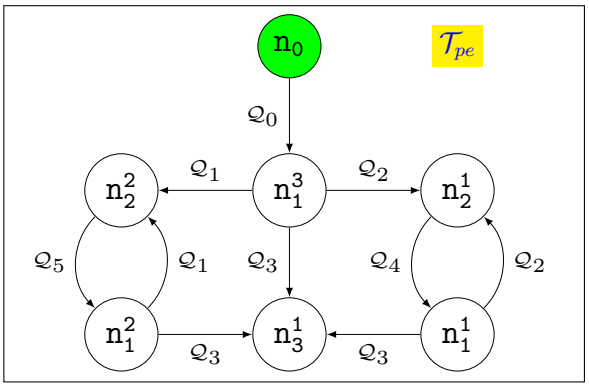
The next example discusses cases for which CFR is essential for proving termination, not only for simplifying the form of the termination witness.

EXAMPLE 3.7. Consider method `search` of Figure 3.4. It receives an array q representing a circular queue, the size of the array n, the indexes h and t of the head and tail, and a value v to search for. The `assert` instruction guarantees the validity of the input. Note that when $h \leq t$, the elements of the queue lie in the interval $[h..t]$, and when $h \geq t+1$ the elements lie in intervals $[h..n-1]$ and $[0..t]$. The loop first (if $h \geq t+1$) searches for v in the interval $[h..n-1]$ and then in $[0..t]$. This defines two phases where moving from the first to the second is done, when h is equal to $n-1$, by setting h to 0. For $h \leq t$ the first phase is not executed.

It is easy to see that (a) in the first phase, $n-h$ is non-negative and decreasing in all iterations except the last, i.e., when setting h to 0 , and thus, in principle, can be used to argue that the first phase is terminating; and (b) in the second phase, $n-h$ is decreasing and non-negative, and thus can be used to argue that this phase is terminating. The last iteration of the first phase makes automatic proofs suitable, because it breaks the conditions that a LRF or a LLRF has to satisfy as function $n-h$ increases when setting h to 0. On the other hand, if we succeed in splitting these phases into separate loops, such that the last iteration of the first phase is a transition that connects them, then proving termination should be possible with LRFs only. Unlike method `phases1` of Figure 3.2,

```
1 void PastaA10(int x, int y) {
2   while (x != y) {
3     if (x >= y + 1)
4       y++;
5     else
6       x++;
7   }
8 }
```





$$
\begin{aligned}
\mathcal{Q}_0 \equiv\{ & & x' = x, & & y' = y & \} \\
\mathcal{Q}_1 \equiv\{ & x \leq y - 1, & x' = x, & & y' = y & \} \\
\mathcal{Q}_2 \equiv\{ & x \geq y + 1, & x' = x, & & y' = y & \} \\
\mathcal{Q}_3 \equiv\{ & x = y, & x' = x, & & y' = y & \} \\
\mathcal{Q}_4 \equiv\{ & x \geq y + 1, & x' = x, & & y' = y + 1 & \} \\
\mathcal{Q}_5 \equiv\{ & x \leq y, & x' = x + 1, & & y' = y & \}
\end{aligned}
$$

Figure 3.3: A loop with 2 phases that are never executed together.

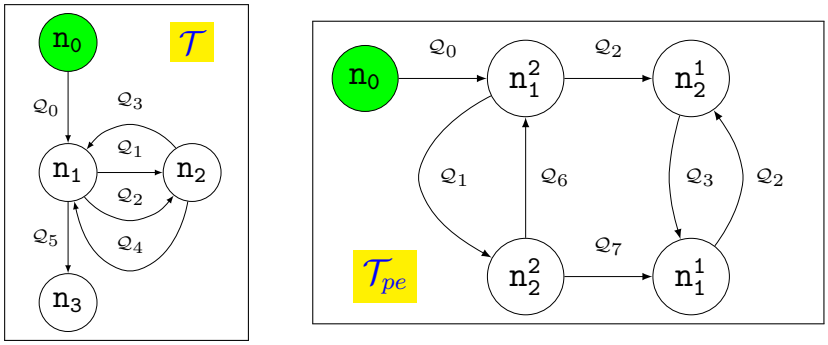
in this one the two phases execute the same code, i.e., the *then* branch at Line 4,which make things more challenging.

The TS $\mathcal{T}$ of this program is shown in Figure 3.4. Node $\mathtt{n}_1$ corresponds to the loop head, and $\mathtt{n}_2$ to the *if* statement. Assume that this loop is the only loop in a larger program that we cannot prove terminating, so as to take advantage of applying CFR at the level of SCCs. Calling TERMINATION($\mathcal{T}$, *false*, 0, 1) we eventually reach Line 15 with SCC S containing nodes $\mathtt{n}_1$ and $\mathtt{n}_2$. TERMINATIONSCC fails to prove termination of any edge of S and returns the set of all edges of S (via F_S). The new TS $\mathcal{T}''$ built at Line 19 is like the TS $\mathcal{T}$ of Figure 3.4, but without the exit node $\mathtt{n}_3$. Applying CFR at Line 20 using properties $\{h = 0, h \leq t\}$ (or $\mathtt{Props}_{\mathtt{c}}$ and $\mathtt{Props}_{\mathtt{cv}}$) we obtain $\mathcal{T}'''$ as $\mathcal{T}_{pe}$ of Figure 3.4, which is added at Line 21 to Q in order to analyse it again. Node $\mathtt{n}_1$ of $\mathcal{T}$ now has 2 versions in $\mathcal{T}_{pe}$:

- $\mathtt{n}_1^2$ is for the first phase which excludes the last step that sets h to 0, which is now handled by the edge $\mathtt{n}_2^2 \xrightarrow{\mathcal{Q}_7} \mathtt{n}_1^1$; and
- node $\mathtt{n}_1^1$ is for the second phase.

Note that node $\mathtt{n}_2$ has 2 versions as well even if it is not a loop head, this is due to the split of $\mathtt{n}_1$. When $\mathcal{T}_{pe}$ is analysed, TERMINATIONSCC finds LRFs for both SCCs of $\mathcal{T}_{pe}$ since the phases have been split as discussed above. □

```
1 int search(int q[], int n, int h, int t, int v) {
2   assert(n>=1 && h<=n-1 && t<=n-1 && h>=0 && t>=0);
3   while (h != t && q[h] != v) {
4      if ( h <= n-2 ) h++;
5      else h = 0;
6   }
7   return q[h] == v;
8 }
```



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv \{h \geq 0, h \leq n-1, t \geq 0, t \leq n-1, n \geq 1, & h' = h, & t' = t, & n' = n & \} \\
\mathcal{Q}_1 \equiv \{h \geq t+1, & h' = h, & t' = t, & n' = n & \} \\
\mathcal{Q}_2 \equiv \{h \leq t-1, & h' = h, & t' = t, & n' = n & \} \\
\mathcal{Q}_3 \equiv \{h \leq n-2, & h' = h+1, & t' = t, & n' = n & \} \\
\mathcal{Q}_4 \equiv \{h \geq n-1, & h' = 0, & t' = t, & n' = n & \} \\
\mathcal{Q}_5 \equiv \{h = t, & h' = h, & t' = t, & n' = n & \} \\
\mathcal{Q}_6 \equiv \{h \leq n-2, h \geq t+1, & h' = h+1, & t' = t, & n' = n & \} \\
\mathcal{Q}_7 \equiv \{h = n-1, & h' = 0, & t' = t, & n' = n & \}
\end{array}
$$

Figure 3.4: A loop that searches in a circular queue.

The next example shows (i) how CFR is useful for inferring precise invariants, without using disjunctive abstract domains; and (ii) the importance of the scheme $\texttt{CFR}_\texttt{A}$, i.e., the scheme that refines all parts of the TS that TerminationSCC failed to prove terminating together.

EXAMPLE 3.8. Let us modify method `search` (and its TS $\mathcal{T}$) to include another variable w, that is initialised to $t-h+1$ before the loop and is set to 1 in the *else* branch at Line 5. The initial value of w is at least 1 if $h \leq t$, and it is at most 0 if $h \geq t+1$. However, in the later case the execution eventually passes through the *else* branch and sets w to 1. This means that $w \geq 1$ holds after the loop. Using the TS $\mathcal{T}$ of this program, invariant generators would fail to infer this information without relying on disjunctive invariants, e.g., $(h \leq t \wedge w \geq 1) \vee (h \geq t+1 \wedge w \leq 0)$ for node $\texttt{n}_1$. However, when using $\mathcal{T}_{pe}$ they succeed using only conjunctions of linear constraints, because in $\mathcal{T}_{pe}$ it is explicit that the second phase is reached, in either way, with $w \geq 1$.

Precise invariants are essential for the precision of termination analysis. Assume that the loop of method `search` is followed by a second loop "`while (x>=1) x=x-w;`", then termination analysis of this loop would fail without the invariant $w \geq 1$ since the loop is non-terminating for $w \leq 0$. Note that in order to propagate $w \geq 1$ from

the first loop to the second, we cannot use the scheme $\texttt{CFR}_\texttt{S}$ since it analyses the SCCs independently, and thus after applying CFR to the SCC of the first loop the new invariants are not propagated to the SCC of the second loop. Instead, we can use $\texttt{CFR}_\texttt{A}$ by calling Termination$(\mathcal{T}, \mathit{false}, 1, 0)$. In this case, the first call to TerminationCFG at Line 2 fails to prove any of the two loops terminating, and then CFR at Line 1 is applied on both loops together and now the constraint $w \geq 1$ is propagated to the second loop. One could also use $\texttt{CFR}_\texttt{B}$, but it might be less efficient if the program is part of a larger one that does not need CFR to handle other SCCs. □

Next we discuss one of the examples that no tool could handle in the termination competition 2019[2] neither 2020[3], except iRankFinder when using the CFR techniques of this thesis.

EXAMPLE 3.9. Consider method `randomwalk` of Figure 3.5. It simulates a random walk process where w is repeatedly increased or decreased at Line 9 depending on the random choice for c at Line 4. The code at Line 5-8 ensures termination, let us explain how. Assuming that $n \geq 1$, the loop passes through the following phases:

1. $z \geq 1$, so z is decremented at Line 5 until it reaches 0;
2. since now $z = 0$ and $i = 1$, it either executes Line 8 and exits the loop, or executes Line 7 which decrements i to 0 and in the next iteration executes Line 5 and sets i to 2 and z back to n;
3. since now $z \geq 1$, it is decremented at Line 5 until it reaches 0;
4. since now $z = 0$ and $i = 2$, it either executes Line 7 twice, which means that $c = 0$ and thus at Line 9 w is decremented twice to 0 and exits the loop, or it executes Line 8 at least once and exits the loop.

The case of $n \leq 0$ passes in steps 2 and 4 only. Applying CFR to $\mathcal{T}$ of Figure 3.5, e.g., using properties $\texttt{Props}_{\texttt{cv}}$ and $\texttt{Props}^{\texttt{d}}_{\texttt{h}}$, we obtain the TS $\mathcal{T}_{pe}$ sketched in Figure 3.5 (each box is a SCC with several nodes and a single cycle that decrements z). Calling Termination on $\mathcal{T}$ with any of the schemes refines the graph to something like $\mathcal{T}_{pe}$, and proves termination with LRF z for both SCCs. □

The next example demonstrates how CFR can simplify non-termination proofs, in particular the part that proves that a recurrent set is actually reachable.

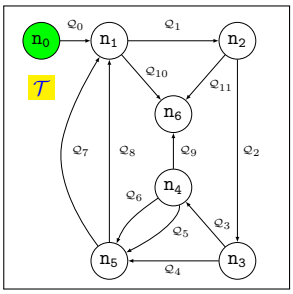
EXAMPLE 3.10. Consider the TS $\mathcal{T}$ depicted in Figure 3.6. The *while* loop has two phases: the first increments y by 2 until condition $x \geq y + 1$ is violated, and the second increments y by 1. Note that the second phase is non-terminating since $y \geq x$ will always hold. The non-termination analysis technique implemented iRankFinder (see Chapter 5) finds a recurrent set

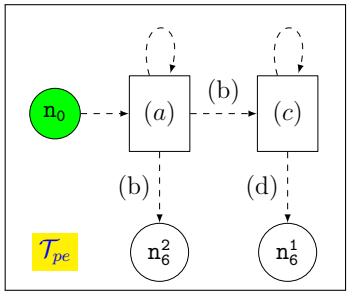
$$\Omega = \{(\mathtt{n}_1, (x, y, x', y')^{\mathrm{T}}) \mid y \geq x, y' = y + 1, x' = x\}$$

which corresponds to taking the edge of $\mathcal{Q}_2$ repeatedly. However, it fails to prove reachability because it uses a very simply reachability analysis that does not unroll loops. Applying CFR using properties $\texttt{Props}_\texttt{c}$ we obtain to TS $\mathcal{T}_{pe}$ depicted in Figure 3.6, and now iRankFinder can prove the reachability of Ω. □

[2] http://termination-portal.org/wiki/Termination_Competition_2019
[3] http://termination-portal.org/wiki/Termination_Competition_2020

```
1 void randomwalk(int n) {
2   int w=1, i=1, z=n, c=0;
3   while ( 1 <= w && w <= 2 ) {
4     c=nondet(); assert(0<=c && c<=1);
5     if ( z >= 1 ) z--;
6     else if ( i <= 0 ) { i=2; z=n; }
7     else if ( i >= 1 && c <= 0 ) i--;
8     else break;
9     if ( c <= 0 ) w--; else w++;
10  }
11 }
```





$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & c'=0, & i'=1, & n'=n, w'=1, & z'=n \} \\
\mathcal{Q}_1 \equiv\{w \leq 2, & c'=c, & i'=i, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_2 \equiv\{w \geq 1, & c' \geq 0, c' \leq 1, & i'=i, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_3 \equiv\{z \leq 0, & c'=c, & i'=i, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_4 \equiv\{z \geq 1, & c'=c, & i'=i, & n'=n, w'=w, & z'=z-1 \} \\
\mathcal{Q}_5 \equiv\{i \geq 1, c \leq 0, & c'=c, & i'=i-1, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_6 \equiv\{i \leq 0, & c'=c, & i'=2, & n'=n, w'=w, & z'=n \} \\
\mathcal{Q}_7 \equiv\{c \leq 0, & c'=c, & i'=i, & n'=n, w'=w-1, & z'=z \} \\
\mathcal{Q}_8 \equiv\{c \geq 1, & c'=c, & i'=i, & n'=n, w'=w+1, & z'=z \} \\
\mathcal{Q}_9 \equiv\{i \geq 1, c \geq 1, & c'=c, & i'=i, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_{10} \equiv\{w \geq 3, & c'=c, & i'=i, & n'=n, w'=w, & z'=z \} \\
\mathcal{Q}_{11} \equiv\{w \leq 0, & c'=c, & i'=i, & n'=n, w'=w, & z'=z \}
\end{array}
$$

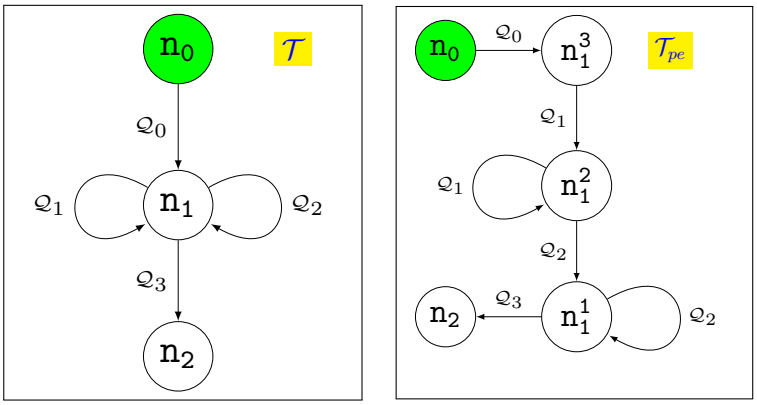
Figure 3.5: A loop that implements a random walk process.

3.3 Application to Cost Analysis

In this section, we discuss the use of CFR for cost analysis, namely the inference of upper-bound functions (in terms of the input variables) on the length of traces. There are several cost analysis tools for (variants of) TSs, and they are based on similar techniques to those used in termination analysis. In particular, they use ranking functions to infer *visit-bounds*, which are upper-bounds on the number of visits to edges in the TS. These tools are typically used as a back-end for cost analysis of different programming languages. Next, we briefly describe some of these tools.

`KoAT` [Brockschmidt et al. 2016b] is a tool that works directly on TSs as defined in Section 2.2.1 and uses the QLRFs of Alias et al. [2010] to infer visit-bounds. `KoAT` uses visit-bounds to also bound the values that a variable can take, for example, if a variable x is incremented by 2 in a SCC, and y is a visit-bound for the edge that increments x, then

```
1 void nonter(int x, int y) {
2   assert(x >= y + 1);
3   while(x != y) {
4     if( x >= y + 1)
5       y = y + 2;
6     else
7       y = y + 1
8   }
9 }
```



$$\begin{aligned}
\mathcal{Q}_0 \equiv\{&x \geq y+1, \quad x' = x, \quad y' = y \quad\} \\
\mathcal{Q}_1 \equiv\{&x \geq y+1, \quad x' = x, \quad y' = y+2 \quad\} \\
\mathcal{Q}_2 \equiv\{&x \leq y-1, \quad x' = x, \quad y' = y+1 \quad\} \\
\mathcal{Q}_3 \equiv\{&x = y, \quad\quad x' = x, \quad y' = y \quad\}
\end{aligned}$$

Figure 3.6: A non-terminating loop.

the value of x would be at most $x_0 + 2y$ when leaving the SCC where x_0 is the value of x when entering the SCC. `CoFloCo` [Flores-Montoya 2017] is a tool that works on a form of TSs that is called *Cost Relations* (CRS), which are similar to recurrence relations. It mainly uses LRFs to infer visit-bounds, and it applies CFR (directly to CRSs) to increase precision. The CFR techniques of `CoFloCo` are very similar to those of Gulwani et al. [2009]. `PUBs` [Albert et al. 2011] is the first tool to infer upper-bounds for CRSs, actually it is the one that introduced the notion of CRSs. It uses LRFs to infer visit-bounds and it does not apply any kind of CFR. In terms of precision, `CoFloCo` can handle anything that `PUBs` can, and it is not directly comparable to `KoAT` as there are cases where one succeed and the other fail to infer upper-bounds.

3.3.1 How CFR Benefits Cost Analysis

In the context of cost analysis, CFR can improve the precision of inferring visit-bounds in a way that is similar to simplifying the witness of a termination proof. The next example shows this for all programs that we have discussed so far. For CFR in this context we simply apply procedure `PE` of Section 3.1 to the input TS.

EXAMPLE 3.11. For Example 3.5, `KoAT` infers a linear upper-bound without CFR thanks to the use of QLRFs, which allows the inference of x and $z - y$ as visit-bounds for $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_4} \mathtt{n}_1$ and $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_3} \mathtt{n}_1$, respectively; `CoFloCo` infers a linear upper-bound since it applies

	Without CFR			With CFR		
	KoAT	CoFloCo	PUBs	KoAT	CoFloCo	PUBs
Example 3.5: phases1	O(n)	O(n)	Fail	O(n)	O(n)	O(n)
Example 3.6: PastaA10	$O(n^2)$	O(n)	Fail	O(n)	O(n)	O(n)
Example 3.7: search	Fail	O(n)	Fail	O(n)	O(n)	O(n)
Example 3.8: search (w)	Fail	Fail	Fail	O(n)	O(n)	O(n)
Example 3.9: randomwalk	Fail	Fail	Fail	O(n)	O(n)	O(n)

Table 3.1: Summary of applying cost analysis on the examples of Section 3.2.3 with and without CFR.

CFR that splits the loop into phases; PUBs fails to infer any upper-bound, however, it infers a linear upper-bound when applied to the TS after CFR. For Example 3.6, without CFR KoAT infer a quadratic bound, CoFloCo a linear bound and PUBs fails. Recall that CoFloCo uses its own CFR. After applying CFR also KoAT and PUBs infer a linear bound. For Example 3.7, KoAT and PUBs fail to infer any upper-bound, but they infer a linear upper-bound after applying CFR; CoFloCo infers a linear upper-bound since it applies its own CFR that splits the loop into two phases. For Example 3.8, all tools fail to infer an upper-bound, and all infer a linear upper-bound after applying CFR (note that CoFloCo's own CFR is insufficient here). For Example 3.9, all tools fail to infer any upper-bound, and they infer a linear upper-bound for the refined TS. These results are summarized in Table 3.1. □

Let us now consider another example for which CFR can improve the precision, and where the phases of the loop share instructions.

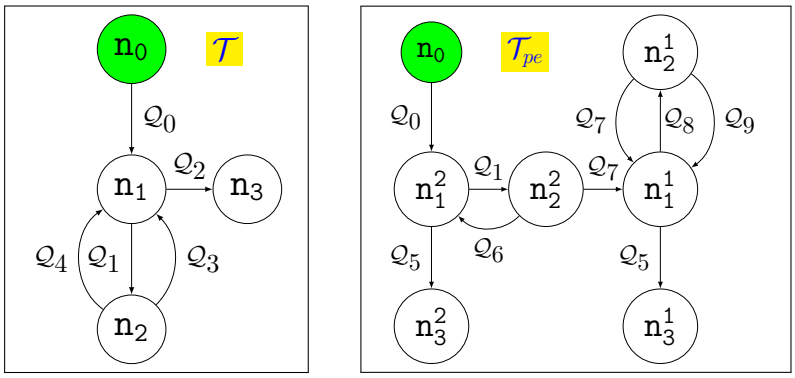
EXAMPLE 3.12. Consider the program of Figure 3.7, which is a variation of the classical iterative McCarthy-91 in which the value of c is not restricted to 1. The loop has two phases:

1. for $x \geq 101$, the *then* branch is executed repetitively until $x \leq 100$ (or $c \leq -1$); and
2. for $x \leq 100$ the *else* branch is executed repetitively until the value of x reaches the interval $x \in [101..111]$, and then the execution alternates between the *then* and *else* branches until c reaches 0.

The complexity of this program is linear. All cost analysis tools that we have considered above fail to infer any upper-bound for the corresponding TS $\mathcal{T}$ that is depicted in Figure 3.7. We noticed that the problem is actually in that the value of x is unbounded, and that when bounding x by a maximum value the SCC of $\mathcal{T}$ has a LRF and all tools succeed to infer a linear upper-bound then.

If we consider the two phases discussed above, we can see that in the second phase the value of x is actually bounded by $x \leq 111$, therefore, if we succeed to split the phases then inferring a linear upper-bound should be possible since the first phase clearly has a LRF. Applying CFR to $\mathcal{T}$ using the property $\{x \leq 111\}$ results in $\mathcal{T}_{pe}$ of Figure 3.7 (we get similar results when using $\texttt{Props}_{\texttt{cv}}$ for example). We can see that the loop has been split into two loops: the SCC of nodes $\texttt{n}_1^2$ and $\texttt{n}_2^2$ corresponds to the first phase, and the SCC of nodes $\texttt{n}_1^1$ and $\texttt{n}_2^1$ to the second one. All tools that we dicussed above succeed to infer a linear bound for $\mathcal{T}_{pe}$. □

```
1 int McCarthy(int x, int c){
2   while (c >= 1) {
3     if (x >= 101) {
4       x -= 10; c--;
5     } else {
6       x += 11; c++;
7     }
8   }
9   return x;
10 }
```



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & & c' = c, & x' = x & \} \\
\mathcal{Q}_1 \equiv\{c \geq 1, & & c' = c, & x' = x & \} \\
\mathcal{Q}_2 \equiv\{c \leq 0, & & c' = c, & x' = x & \} \\
\mathcal{Q}_3 \equiv\{x \geq 101, & & c' = c-1, & x' = x-10 & \} \\
\mathcal{Q}_4 \equiv\{x \leq 100, & & c' = c+1, & x' = x+11 & \} \\
\mathcal{Q}_5 \equiv\{c = 0, & & c' = c, & x' = x & \} \\
\mathcal{Q}_6 \equiv\{c \geq 1, x \geq 101, & & c' = c-1, & x' = x-10 & \} \\
\mathcal{Q}_7 \equiv\{c \geq 1, x \leq 100, & & c' = c+1, & x' = x+11 & \} \\
\mathcal{Q}_8 \equiv\{c \geq 1, x \leq 111, & & c' = c, & x' = x & \} \\
\mathcal{Q}_9 \equiv\{c \geq 1, x \leq 111, x \geq 101, & & c' = c-1, & x' = x-10 & \}
\end{array}
$$

Figure 3.7: Iterative McCarthy.

The following example demonstrates that CFR can improve precision of invariant generation, which in turn improves the precision of cost analysis.

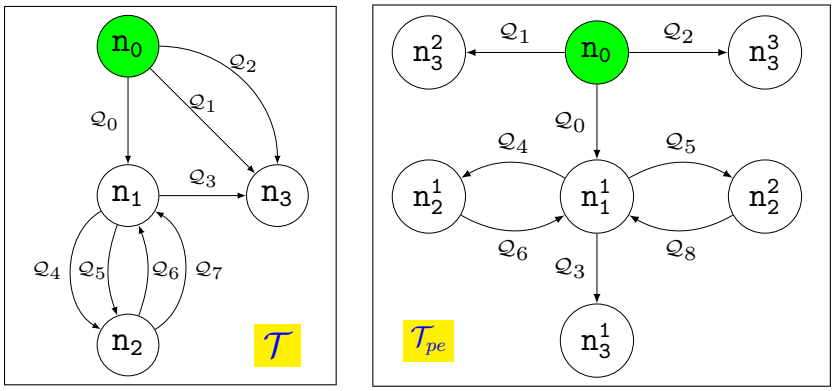
EXAMPLE 3.13. Consider the GCD program depicted in Figure 3.8. It calculates the GCD of two positive integers using iterative subtracting. This program terminates and has a linear runtime complexity.

The TS $\mathcal{T}$ depicted in Figure 3.8 (middle) is automatically obtained using `LLVM2KTTEL`[4]. Note that edge $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_7} \mathtt{n}_1$, which corresponds to the *else* branch, includes the constraint $x \geq y$ while at runtime it will always be the case that $x \geq y + 1$. This happens because `LLVM2KTTEL` translates the if-statement independently from the context (the while-condition), and thus for the *else* branch it takes the negation of the if-condition.

iRankFinder fails to prove termination of this TS. The reason is that proving termination requires the invariant $\{x \geq 1, y \geq 1\}$ for node $\mathtt{n}_2$ in order to rank transitions

[4]A tool for converting C programs into TSs https://github.com/s-falke/llvm2kittel

```
1 int gcd(int x, int y){
2   assert(x>=1 && y>=1);
3   while(x != y){
4     if(x<=y-1)
5       y = y-x;
6     else
7       x = x-y;
8   }
9   return x;
10 }
```



$$
\begin{aligned}
\mathcal{Q}_0 \equiv\{&x \geq 1, y \geq 1, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_1 \equiv\{&x \leq 0, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_2 \equiv\{&y \leq 0, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_3 \equiv\{&x = y, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_4 \equiv\{&x \leq y - 1, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_5 \equiv\{&x \geq y + 1, & x' = x, \quad & y' = y & \}\\
\mathcal{Q}_6 \equiv\{&x \leq y - 1, & x' = x, \quad & y' = y - x & \}\\
\mathcal{Q}_7 \equiv\{&x \geq y, & x' = x - y, \quad & y' = y & \}\\
\mathcal{Q}_8 \equiv\{&x \geq y + 1, & x' = x - y, \quad & y' = y & \}
\end{aligned}
$$

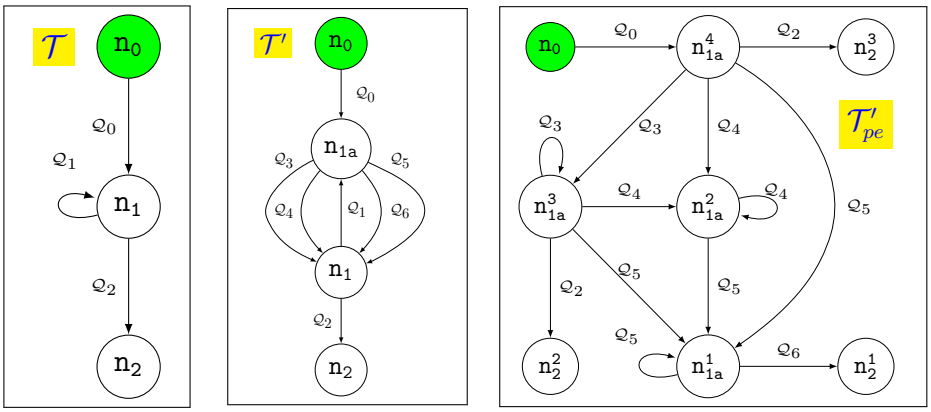
Figure 3.8: A loop that calculates the *greatest common divisor* by iterative subtracting.

$\mathtt{n}_2 \xrightarrow{\mathcal{Q}_6} \mathtt{n}_1$ and $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_7} \mathtt{n}_1$. iRankFinder fails to infer this invariant mainly due to the constraint $x \geq y$, which at some point in the fixpoint computation introduces $x \geq 0$ at node $\mathtt{n}_1$, and then the widening operation loses the lower bound of x (a more clever widening would have solved the problem). In order to solve this problem, it is enough to unfold transitions in the corresponding SCC, and thus collapse the two nodes into one ($\mathtt{n}_1 \xrightarrow{\mathcal{Q}_4} \mathtt{n}_2$ followed by $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_6} \mathtt{n}_1$, and $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_5} \mathtt{n}_2$ followed by $\mathtt{n}_2 \xrightarrow{\mathcal{Q}_7} \mathtt{n}_1$) which then will eliminate $x \geq y$. Unfolding is the basic operation in partial evaluation. `KoAT` and `PUBs` fail to infer any bound, while `CoFloCo` infers the expected linear bound.

Applying CFR to this TS results in $\mathcal{T}_{pe}$ depicted in Figure 3.8 on the right. We can see that CFR actually did not collapse the nodes of the recursive SCC into one, which would have solved the problem. It actually splits the two phases of the loop into separated ones – nodes $\mathtt{n}_1^1$ and $\mathtt{n}_2^2$ correspond to the *else* branch and nodes $\mathtt{n}_1^1$ and $\mathtt{n}_2^1$ correspond to the *then* branch. In addition, it has merged transitions so that $x \geq y$ disappeared.

Applying iRankFinder of this TS infers a LRF for all components. This is possible

```
1 void phases_xyz(int x, int y, int z) {
2   while( x+z >= 0 ){
3     x = x + y;
4     y = y + z;
5     z = z - 1;
6   }
7 }
```



$$
\begin{array}{lllll}
\mathcal{Q}_0 \equiv\{ & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_1 \equiv\{x + z \geq 0, & x' = x + y, & y' = y + z, & z' = z - 1 & \} \\
\mathcal{Q}_2 \equiv\{x + z \leq -1, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_3 \equiv\{z \geq 0, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_4 \equiv\{z \leq -1, y \geq 0, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_5 \equiv\{z \leq -1, y \leq -1, x \geq 0, & x' = x, & y' = y, & z' = z & \} \\
\mathcal{Q}_6 \equiv\{z \leq -1, y \leq -1, x \leq -1, & x' = x, & y' = y, & z' = z & \}
\end{array}
$$

Figure 3.9: A SLC loop with 3 phases.

since now it infers the invariant $x \geq 1 \wedge y \geq 1$ where needed. Applying `KoAT` or `PUBs` on this TS we get the desired bound $O(|x| + |y|)$. □

3.3.2 Using Ranking Functions as Properties

Sometimes, we can prove termination of a TS but cannot infer an upper-bound on its cost. The reason is that in termination analysis we can use ranking functions that do not necessarily imply an upper-bound on the length of traces, e.g., LLRFs, while in cost analysis we typically need ranking functions that imply such upper-bounds.

For some kind of LLRFs, e.g., MΦRFs, even if they do not imply an upper-bound, they provide useful information about the control-flow. Our goal is to use these ranking functions to help CFR to make this control flow explicit, which in turn helps cost analysis to infer upper-bounds. We first explain the idea using an example, and then explain the general case.

EXAMPLE 3.14. Consider the program `phases_xyz` in Figure 3.9. Termination analysis succeeds in proving termination, assigning to node $\mathtt{n}_1$ the MΦRF $\langle z, y, x\rangle$ which, as we have seen already, has the following behaviour (considering consecutive visits to $\mathtt{n}_1$): z decreases always, and when it becomes negative y starts to decrease, and when

y becomes negative x starts to decrease. This means that whenever $\mathtt{n_1}$ is reached one of the following must hold: $z \geq 0$, $z \leq -1 \wedge y \geq 0$, $z \leq -1 \wedge y \leq -1 \wedge x \geq 0$ or $z \leq -1 \wedge y \leq -1 \wedge x \leq -1$. The TS $\mathcal{T}'$ of Figure 3.9 is a modification of $\mathcal{T}$ making this information explicit: we added a new node $\mathtt{n_{1a}}$, changed all incoming edges of $\mathtt{n_1}$ to go to $\mathtt{n_{1a}}$, and added 4 edges from $\mathtt{n_{1a}}$ to $\mathtt{n_1}$, each annotated with one of the above constraints. Applying CFR to $\mathcal{T}'$ results in the TS $\mathcal{T}'_{pe}$ in which the phases are explicit (nodes $\mathtt{n^1_{1a}}$, $\mathtt{n^2_{1a}}$, and $\mathtt{n^3_{1a}}$). Cost analysis tools can infer a linear upper-bound for $\mathcal{T}'_{pe}$. The addition of node $\mathtt{n_{1a}}$ is essential, applying CFR directly to $\mathcal{T}$ would not do any refinement. □

The above example can be generalized as follows. If a loop-head node $\mathtt{n_i}$ is assigned a MΦRF $\langle \rho_1, \ldots, \rho_k \rangle$ during termination analysis: we add a new node $\mathtt{n_{ia}}$ and change all incoming edges of $\mathtt{n_i}$ to $\mathtt{n_{ia}}$; and add $k+1$ edges from $\mathtt{n_{ia}}$ to $\mathtt{n_i}$ where the ith edge has the constraints

$$\rho_1(\vec{x}) < 0 \wedge \cdots \wedge \rho_{i-1}(\vec{x}) < 0 \wedge \rho_i(\vec{x}) \geq 0 \wedge \mathtt{ID}(\mathbf{x}).$$

Applying CFR to the new TS takes advantage of the new edges, and splits the loop into corresponding phases. Finally, note that this technique, to some extension, can be see as a special case of the techniques of Albert et al. [2019] for cost analysis.

Chapter 4

Multi-Phase Ranking Functions and Their Relation to Recurrent Sets

In Chapter 1, we have discussed ranking functions as a fundamental tool for proving termination, in particular, in Section 1.1, we have seen examples for which termination can be proven using LRFs, and examples for which LRFs do not suffice and one has to use a more sophisticated class of ranking functions such as LLRFs. In Section 2.3.1, we have seen several definitions for LLRFs, discussed their properties, and compared their relative power. Section 2.3.1, ended by a discussion on a subclass of LLRFs called MΦRFs, that have not been explored enough in the literature. In this chapter, we study this class of ranking functions for the case of SLC loops, where some of the results are then generalised to TSs as well.

The formal definition of MΦRFs has been given in Section 2.3.1 and will be specialised for SLC loops in the next section to simplify the presentation. But recall that a MΦRF is a tuple $\langle \rho_1, \ldots, \rho_d \rangle$ of linear functions that define phases of the loop (or SCC in TS) that are linearly ranked, where d is the *depth* of the MΦRF, intuitively the number of phases. The decision problem *Existence of a MΦRF* asks to determine whether a SLC loop has a MΦRF. The *bounded* decision problem restricts the search to MΦRFs of depth d, where d is part of the input. The complexity and algorithmic aspects of the bounded version of the MΦRF problem were completely settled by [Ben-Amram and Genaim 2017]. The decision problem is `PTIME` for SLC loops with rational-valued variables, and `coNP`-complete for SLC loops with integer-valued variables; synthesising MΦRFs, when they exist, can be performed in polynomial and exponential time, respectively. Besides, Ben-Amram and Genaim [2017] show that for SLC loops MΦRFs have the same power as general lexicographic-linear ranking functions, and that, surprisingly, MΦRFs induce linear iteration bounds. The problem of deciding if a given SLC loop admits a MΦRF, without a given bound on the depth, is still open.

In practice, termination analysis tools search for MΦRFs starting by depth 1 and increase the depth until they find one, or reach a predefined limit, after which the returned answer is *don't know*. Finding a theoretical upper-bound on the depth of a MΦRF, given the loop, would also settle this problem. As shown by Ben-Amram and Genaim [2017], such bound must depend not only on the number of constraints or variables as in other classes of LLRFs [Alias et al. 2010; Ben-Amram and Genaim 2014; Bradley et al. 2005a] but also on the coefficients used in the constraints.

In this chapter, we make progress towards solving the problem of *existence of a MΦRF*, i.e., seeking a MΦRF without a given bound on the depth. In particular, we present an

algorithm for seeking MΦRFs that reveals new insights on the structure of these ranking functions. In a nutshell, the algorithm starts from the set of transitions of the given SLC loop, which is a polyhedron, and iteratively removes transitions $\binom{\mathbf{x}}{\mathbf{x}'}$ such that $\rho(\mathbf{x}) - \rho(\mathbf{x}') > 0$ for some function $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$ that is *non-negative on all enabled states*. The process continues iteratively, since after removing some transitions, more functions ρ may satisfy the non-negativity condition, and they may eliminate additional transitions in the next iteration. When all transitions are eliminated in a finite number of iterations, we can construct a MΦRF using the ρ functions; and when reaching a situation in which no transition can be eliminated, we prove that we have reached a recurrent set that witnesses non-termination.

The algorithm always finds a MΦRF if one exists, and in many cases, it finds a recurrent set when the loop is non-terminating, however, it is not a decision procedure as it diverges in some cases. Nonetheless, our algorithm provides important insights into the structure of MΦRFs. Apart from revealing a relation between seeking MΦRFs and seeking recurrent sets, these insights are useful for finding classes of SLC loops for which, when terminating, there is always a MΦRF and thus have linear run-time bound.

Our research has, in addition, led to a new representation for SLC loops, that we refer to as the *displacement* representation, that provides us with new tools for studying termination of SLC loops in general, and the existence of a MΦRF in particular. In this representation, a transition $\binom{\mathbf{x}}{\mathbf{x}'}$ is represented as $\binom{\mathbf{x}}{\mathbf{y}}$ where $\mathbf{y} = \mathbf{x}' - \mathbf{x}$. Using this representation our algorithm can be formalised in a simple way that avoids computing the ρ functions mentioned above (which might be expensive), and reduces the existence of a MΦRF of depth d to unsatisfiability of a certain linear constraint system. Moreover, any satisfying assignment is a witness that explains why the loop has no MΦRF of depth d. As evidence on the usefulness of this representation in general, we also show that some non-trivial observations on termination of bounded SLC loops are made straightforward in this representation, while they are not easy to see in the normal representation.

The rest of this chapter is organised as follows.

- In Section 4.1, we describe our algorithm and its possible outcomes, in particular: Section 4.2.1 describes the algorithm; Section 4.2.2 describes how it can be used to infer recurrent sets for SLC loop; Section 4.2.3 describes how it can also be used for inferring recurrent sets for TSs; and Section 4.2.4 discusses cases for which the algorithm does not terminate and raises some interesting questions.

- In Section 4.3, we describe the displacement representation for SLC loops and its benefits, in particular: Section 4.3.1 describes how it can be used to generate witnesses against the existence of MΦRFs; Section 4.3.2 describes how it can be used for conditional termination; Section 4.3.3 uses it to make some non-trivial observations on termination in general; and Section 4.3.4 discusses some new directions, that are a direct consequence of this representation, for addressing the MΦRF problem .

- In Section 4.4, we discuss the usefulness of our algorithm for characterising classes of SLC loops for which when terminating, there is always a MΦRF. The content of this section is not published and appears for the first time in this thesis.

Implementation and experimental evaluation, in particular for the use of our algorithm in the context of non-termination, are discussed in chapters 5 and 6.

4.1 Multi-Phase Ranking Functions for SLC loops

In the rest of this chapter, we assume that our TS is given as a SLC loop specified by a transition polyhedra $\mathcal{Q}$. Variables are assumed to range over the rationals, and we always discuss the case of integers after considering the rationals case.

In Section 2.3.1, we have already provided a definition for MΦRF, which are a special case of LLRFs, using the notion of QLRFs. However, since now our interest is in SLC loops only, we start by giving an explicit definition for the case of SLC loops. For brevity, we use $\Delta\rho(\mathbf{x}'')$ for $\rho(\mathbf{x}) - \rho(\mathbf{x}')$.

Definition 4.1. Given a SLC loop $\mathcal{Q} \subseteq \mathbb{Q}^{2n}$, we say that $\tau = \langle \rho_1, \ldots, \rho_d \rangle$ is a MΦRF (of depth d) for $\mathcal{Q}$ if for every $\mathbf{x}'' \in \mathcal{Q}$ there is an index i such that:

$$\rho_i(\mathbf{x}) \geq 0\,, \tag{4.1}$$
$$\forall j \leq i \;.\; \Delta\rho_j(\mathbf{x}'') \geq 1\,, \tag{4.2}$$
$$\forall j < i \;.\; \quad \rho_j(\mathbf{x}) \leq 0\,. \tag{4.3}$$

We say that $\mathbf{x}''$ is *ranked by* ρ_i (for the minimal such i).

It is not hard to see that a MΦRF $\langle \rho_1 \rangle$ of depth $d = 1$ is a LRF. If the MΦRF is of depth $d > 1$, it implies that if $\rho_1(\mathbf{x}) \geq 0$, transition $\mathbf{x}''$ is ranked by ρ_1, while if $\rho_1(\mathbf{x}) < 0$, $\langle \rho_2, \ldots, \rho_d \rangle$ becomes a MΦRF. This agrees with the intuitive notion of "phases." that we have mentioned before. Moreover, the definition above coincides with the explanation used in Section 2.3.1: ρ_1 is a QLRF for $\mathcal{Q}$, ρ_2 is a QLRF for $\mathcal{Q} \wedge \rho(\mathbf{x}) < 0$, and so on. We say that τ is *irredundant* if removing any component invalidates the MΦRF. Finally, it is convenient to allow an empty tuple as a MΦRF, of depth 0, for the empty set.

4.2 Inferring Multi-Phase Ranking Functions

In this section, we describe an algorithm for deciding the existence of MΦRFs and how to construct them, this algorithm is also able to find recurrent sets for certain non-terminating SLC loops without any extra computation.

Let us start with an intuitive description of the algorithm and its possible outcomes. Our algorithm is based on the following crucial observation: given linear functions $\rho_1, \ldots, \rho_l$ such that

- $\rho_1, \ldots, \rho_l$ are non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$, i.e., over all enabled states;
- for some ρ_i, we have $\Delta\rho_i(\mathbf{x}'') > 0$ for at least one transition $\mathbf{x}'' \in \mathcal{Q}$; and
- $\mathcal{Q}' = \mathcal{Q} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_l(\mathbf{x}'') \leq 0$ has a MΦRF of depth d

then $\mathcal{Q}$ has a MΦRF of depth at most $d+1$. The proof of this observation is constructive, i.e., given a MΦRF τ' for $\mathcal{Q}'$, we can construct a MΦRF τ for $\mathcal{Q}$ using conic combinations of the components of τ' and $\rho_1, \ldots, \rho_l$.

Let us assume that we have a procedure $F(\mathcal{Q})$ that picks some candidate functions $\rho_1, \ldots, \rho_l$, i.e., non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$, and computes

$$F(\mathcal{Q}) = \mathcal{Q} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_l(\mathbf{x}'') \leq 0.$$

Clearly, if $F^d(\mathcal{Q}) = \emptyset$, for some $d > 0$, then using the above observation we can conclude that $\mathcal{Q}$ has a MΦRF of depth at most d. Obviously, the difficult part in defining F is how to pick functions $\rho_1, \ldots, \rho_l$, and, moreover, how to ensure that if $\mathcal{Q}$ has a MΦRF of optimal depth d then $F^d(\mathcal{Q}) = \emptyset$, i.e., to find the optimal depth. For this, we observe that the set of all non-negative functions over $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a polyhedral cone, and thus it has generators $\rho_1, \ldots, \rho_l$ that can be effectively computed. These generators $\rho_1, \ldots, \rho_l$ turn out to be the right candidates to use. In addition, when using these candidates, we prove that if we cannot make progress, i.e., we get $F^{i-1}(\mathcal{Q}) = F^i(\mathcal{Q})$, then we have actually reached a recurrent set that witnesses non-termination.

The rest of this section is organised as follows: in Section 4.2.1 we present the algorithm and discuss how it is used to decide existence of MΦRFs; in Section 4.2.2 we discuss how the algorithm can infer recurrent sets; in Section 4.2.3 we discuss how to infer recurrent sets for general CFGs and not only SLC loops; and in Section 4.2.4 we discuss cases where the algorithm does not terminate and raise some questions on what happens in the limit.

4.2.1 Deciding Existence of MΦRFs

We start by formally defining the set of all non-negative functions over a given polyhedron $\mathcal{S} \subseteq \mathbb{Q}^n$, which is crucial for picking up the candidate functions $\rho_1, \ldots, \rho_l$ that we have discussed above.

Definition 4.2. The set of all non-negative functions, over a polyhedron $\mathcal{S} \subseteq \mathbb{Q}^n$, is defined as $\mathcal{S}^{\#} = \{(\vec{a}, b) \in \mathbb{Q}^{n+1} \mid \forall \mathbf{x} \in \mathcal{S}.\ \vec{a} \cdot \mathbf{x} + b \geq 0\}$.

It is known that $\mathcal{S}^{\#}$ is a polyhedral cone [Schrijver 1986, p. 112]. Equivalently, it is generated by a finite set of rays $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$. The cone generated by $\vec{a}_1, \ldots, \vec{a}_l$ is known as the dual of the cone $\texttt{rec.cone}(\mathcal{S})$ – we make use of this in Section 4.3. These rays are actually the ones that are important for the algorithm, as can be seen in the definition below, however, in the definition of $\mathcal{S}^{\#}$ we included the b_i's as they makes some statements smoother. Since $\mathcal{S}$ is a closed convex set, it is known that it is equal to the intersection of all half-spaces defined by the elements of $\mathcal{S}^{\#}$, i.e., $\mathcal{S} = \wedge\{\vec{a} \cdot \mathbf{x} + b \geq 0 \mid (\vec{a}, b) \in \mathcal{S}^{\#}\}$. In what follows we are interested in the set of non-negative functions over the enabled stated $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$, namely in the set $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$.

Definition 4.3. Let $\mathcal{Q}$ be a SLC loop, and define

$$F(\mathcal{Q}) = \mathcal{Q} \ \wedge\ \vec{a}_1 \cdot \mathbf{x} - \vec{a}_1 \cdot \mathbf{x}' \leq 0 \ \wedge\ \cdots \ \wedge\ \vec{a}_l \cdot \mathbf{x} - \vec{a}_l \cdot \mathbf{x}' \leq 0$$

where $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$ are the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$.

It is easy to see that each $\vec{a}_i \cdot \mathbf{x} - \vec{a}_i \cdot \mathbf{x}' \leq 0$ above is actually $\Delta\rho_i(\mathbf{x}'') \leq 0$ where $\rho_i = \vec{a}_i \cdot \mathbf{x} + b_i$. Intuitively, $F(\mathcal{Q})$ removes from $\mathcal{Q}$ all transitions $\mathbf{x}''$ for which there is $(\vec{a}, b) \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ such that $\vec{a} \cdot \mathbf{x} - \vec{a} \cdot \mathbf{x}' > 0$. This is because any $(\vec{a}, b) \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ is a conic combination of $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$, and thus for some i we must have $\vec{a}_i \cdot \mathbf{x} - \vec{a}_i \cdot \mathbf{x}' > 0$, otherwise we would have $\vec{a} \cdot \mathbf{x} - \vec{a} \cdot \mathbf{x}' = 0$.

EXAMPLE 4.4. Consider the SLC loop of Figure 2.2 whose transition polyhedron is defined by

$$\mathcal{Q} = \{x + z \geq 0, x' = x + y,\ y' = y + z,\ z' = z - 1\}.$$

The generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ are $\{(1,0,1,0),(0,0,0,1)\}$. That is, the corresponding non-negative functions are $\rho_1(x,y,z) = x + z$ and $\rho_2(x,y,z) = 1$ (the last component of each generator corresponds to the free constant b, and the rest to $\vec{a}$). Computing $F(\mathcal{Q})$, following Definition 4.3, results in:

$$\mathcal{Q}' = \mathcal{Q} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \Delta\rho_2(\mathbf{x}'') \leq 0 = \mathcal{Q} \wedge (x+z) - (x'+z') \leq 0 \tag{4.4}$$

This eliminates any transition for which the quantity $x + z$ decreases. □

The reader might have noticed that function ρ_i above are similar, but not quite the same, to the notion of QLRFs described in Section 2.3.1. In fact, they have a similar role, but they will be used in another way to construct the MΦRF.

In what follows, we aim at showing that $\mathcal{Q}$ has a MΦRF of optimal depth d iff we have $F^d(\mathcal{Q}) = \emptyset$. We first state some auxiliary lemmas.

The following lemma is fundamental, it basically states that if we have a finite set of linear functions where each is positive on some parts of a given polyhedron $\mathcal{P}$, and that any point of $\mathcal{P}$ is covered by one of these functions, then it is possible to construct a single function that is positive over all $\mathcal{P}$.

LEMMA 4.5. *Given a polyhedron $\mathcal{P} \neq \emptyset$, and linear functions $\rho_1, \ldots, \rho_k$ such that*

(i) $\mathbf{x} \in \mathcal{P} \rightarrow \rho_1(\mathbf{x}) > 0 \vee \cdots \vee \rho_{k-1}(\mathbf{x}) > 0 \vee \rho_k(\mathbf{x}) \geq 0$

(ii) $\mathbf{x} \in \mathcal{P} \not\rightarrow \rho_1(\mathbf{x}) > 0 \vee \cdots \vee \rho_{k-1}(\mathbf{x}) > 0$

There exist non-negative constants $\mu_1, \ldots, \mu_{k-1}$ such that

$$\mathbf{x} \in \mathcal{P} \rightarrow \mu_1\rho_1(\mathbf{x}) + \cdots + \mu_{k-1}\rho_{k-1}(\mathbf{x}) + \rho_k(\mathbf{x}) \geq 0.$$

Proof. Let $\mathcal{P}$ be $B\mathbf{x} \leq \mathbf{c}$, $\rho_i = \vec{a}_i \cdot \mathbf{x} - b_i$, then (i) is equivalent to infeasibility of

$$B\mathbf{x} \leq \mathbf{c} \wedge A\mathbf{x} \leq \mathbf{b} \wedge \vec{a}_k \cdot \mathbf{x} < b_k \tag{4.5}$$

where A consists of the $k-1$ rows $\vec{a}_i$, and $\mathbf{b}$ of corresponding b_i. However, $B\mathbf{x} \leq \mathbf{c} \wedge A\mathbf{x} \leq \mathbf{b}$ is assumed to be feasible.

According to Motzkin's transposition theorem [Schrijver 1986, Corollary 7.1k, p. 94], this implies that there are row vectors $\vec{\lambda}, \vec{\lambda}' \geq 0$ and a constant $\mu \geq 0$ such that the following is true:

$$\vec{\lambda}B + \vec{\lambda}'A + \mu a_k = 0 \wedge \vec{\lambda}\mathbf{c} + \vec{\lambda}'\mathbf{b} + \mu b_k \leq 0 \wedge (\mu \neq 0 \vee \vec{\lambda}\mathbf{c} + \vec{\lambda}'\mathbf{b} + \mu b_k < 0) \tag{4.6}$$

Now, if (4.6) is true, then for all $\mathbf{x} \in \mathcal{P}$,

$$\begin{aligned}(\sum_i \lambda'_i\rho_i(\mathbf{x})) + \mu\rho_k(\mathbf{x}) &= \vec{\lambda}'A\mathbf{x} - \vec{\lambda}'\mathbf{b} + \mu a_k\mathbf{x} - \mu b_k \\ &= -\vec{\lambda}B\mathbf{x} - \vec{\lambda}'\mathbf{b} - \mu b_k \geq -\vec{\lambda}\mathbf{c} - \vec{\lambda}'\mathbf{b} - \mu b_k \geq 0\end{aligned}$$

where if $\mu = 0$, the last inequality must be strict. However, if $\mu = 0$, then $\vec{\lambda}B + \vec{\lambda}'A = 0$, so by feasibility of $B\mathbf{x} \leq \mathbf{c}$ and $A\mathbf{x} \leq \mathbf{b}$, this implies $\vec{\lambda}\mathbf{c} + \vec{\lambda}'\mathbf{b} \geq 0$, a contradiction. Thus, $(\sum_i \lambda'_i\rho_i) + \mu\rho_k \geq 0$ on $\mathcal{P}$ and $\mu > 0$. Dividing by μ we get the conclusion of the lemma. □

The following lemma shows how to construct a MΦRF for $\mathcal{Q}$, from a MΦRF for $\mathcal{Q}' = F(\mathcal{Q})$. It also clarifies how the approach of this chapter is different from QLRFs and the incremental algorithm of Section 2.3.1.

LEMMA 4.6. *If $\mathcal{Q}' = F(\mathcal{Q})$ has a MΦRF of depth at most d, then $\mathcal{Q}$ has a MΦRF of depth at most $d+1$.*

Proof. Consider the generators $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$ used in Definition 4.3, and let $\rho_i(\mathbf{x}) = \vec{a}_i \cdot \mathbf{x} + b_i$. We have $\mathcal{Q}' = \mathcal{Q} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_l(\mathbf{x}'') \leq 0$. We assume that no ρ_i is redundant, otherwise we take an irredundant subset.

Let $\tau = \langle g_1, \ldots, g_d \rangle$ be a MΦRF for $\mathcal{Q}'$, and w.l.o.g. assume that it is of optimal depth, we show how to construct a MΦRF $\tau' = \langle g'_1 + 1, \ldots, g'_d + 1, g_{d+1} \rangle$ for $\mathcal{Q}$. Note that simply appending $\rho_1, \ldots, \rho_l$ to a MΦRF τ of $\mathcal{Q}'$ does not always produce a MΦRF, since the components of τ are not guaranteed to decrease over $\mathcal{Q} \setminus \mathcal{Q}'$. Instead, we construct the components of τ' one by one.

If g_1 is decreasing over $\mathcal{Q}$, we define $g'_1(\mathbf{x}) = g_1(\mathbf{x})$, otherwise we have

$$\mathbf{x}'' \in \mathcal{Q} \rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_l(\mathbf{x}'') > 0 \vee \Delta g_1(\mathbf{x}'') - 1 \geq 0 \tag{4.7}$$

$$\mathbf{x}'' \in \mathcal{Q} \not\rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_l(\mathbf{x}'') > 0 \tag{4.8}$$

and by Lemma 4.5 there are non-negative constants $\mu_1, \ldots, \mu_l$ such that

$$\mathbf{x}'' \in \mathcal{Q} \rightarrow \Delta g_1(\mathbf{x}'') - 1 + \sum_{i=1}^{l} \mu_i \Delta\rho_i(\mathbf{x}'') \geq 0. \tag{4.9}$$

Define $g'_1(\mathbf{x}) = g_1(\mathbf{x}) + \sum_{i=1}^{l} \mu_i \rho_i(\mathbf{x})$. Clearly, $\mathbf{x}'' \in \mathcal{Q} \rightarrow \Delta g'_1(\mathbf{x}'') \geq 1$. Moreover, since $\rho_1, \ldots, \rho_l$ are non-negative on all enabled states, g'_1 is non-negative on the states on which g_1 is non-negative. If $d > 1$, we proceed with

$$\mathcal{Q}^{(1)} = \mathcal{Q} \cap \{\mathbf{x}'' \mid g'_1(\mathbf{x}) \leq (-1)\}. \tag{4.10}$$

If g_2 is decreasing over $\mathcal{Q}^{(1)}$, let $g'_2 = g_2$, otherwise, since transitions in $\mathcal{Q}' \cap \mathcal{Q}^{(1)}$ are ranked by $\langle g_2, \ldots, g_d \rangle$ we have

$$\mathbf{x}'' \in \mathcal{Q}^{(1)} \rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_l(\mathbf{x}'') > 0 \vee \Delta g_2(\mathbf{x}'') - 1 \geq 0 \tag{4.11}$$

$$\mathbf{x}'' \in \mathcal{Q}^{(1)} \not\rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_l(\mathbf{x}'') > 0 \tag{4.12}$$

and again by Lemma 4.5 we can construct the desired g'_2 as we did for g'_1. In general, for any $j \leq d$ we construct g'_{j+1} such that $\Delta g'_{j+1}(\mathbf{x}'') \geq 1$ over

$$\mathcal{Q}^{(j)} = \mathcal{Q} \cap \{\mathbf{x}'' \in \mathbb{Q}^{2n} \mid g'_1(\mathbf{x}) \leq (-1) \wedge \cdots \wedge g'_j(\mathbf{x}) \leq (-1)\} \tag{4.13}$$

and $\mathbf{x}'' \in \mathcal{Q} \wedge g_j(\mathbf{x}) \geq 0 \rightarrow g'_j(\mathbf{x}) \geq 0$. Finally we define

$$\mathcal{Q}^{(d)} = \mathcal{Q} \cap \{\mathbf{x}'' \in \mathbb{Q}^{2n} \mid g'_1(\mathbf{x}) \leq (-1) \wedge \cdots \wedge g'_d(\mathbf{x}) \leq (-1)\} \tag{4.14}$$

and note that

$$\mathbf{x}'' \in \mathcal{Q}^{(d)} \rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_l(\mathbf{x}'') > 0 \tag{4.15}$$

We have assumed that no ρ_i is redundant in (4.15), otherwise we take an irredundant subset. Now from (4.15) we get

$$\mathbf{x}'' \in (\mathcal{Q}^{(d)} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_{l-1}(\mathbf{x}'') \leq 0) \rightarrow \Delta\rho_l(\mathbf{x}'') > 0 \tag{4.16}$$

Algorithm 3: Deciding existence of MΦRFs and inferring recurrent sets.

```
DecideMLRF(𝒬)
1  if (𝒬 is empty) then return ∅
2  else
3      Compute the generators (a⃗_1, b_1), ..., (a⃗_l, b_l) of proj_x(𝒬)^#
4      𝒬' = 𝒬 ∧ a⃗_1 · x − a⃗_1 · x' ≤ 0 ∧ ··· ∧ a⃗_l · x − a⃗_l · x' ≤ 0
5      if (𝒬' == 𝒬) then return 𝒬
6      else return DecideMLRF (𝒬')
```

and since the left-hand side is a polyhedron, there is a constant $c > 0$ such that

$$\mathbf{x}'' \in (\mathcal{Q}^{(d)} \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_{l-1}(\mathbf{x}'') \leq 0) \rightarrow \Delta\rho_l(\mathbf{x}'') \geq c. \tag{4.17}$$

W.l.o.g. we may assume that $c \geq 1$, otherwise we divide ρ_l by c. Then we have

$$\mathbf{x}'' \in \mathcal{Q}^{(d)} \rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_{l-1}(\mathbf{x}'') > 0 \vee \Delta\rho_l(\mathbf{x}'') - 1 \geq 0 \tag{4.18}$$

$$\mathbf{x}'' \in \mathcal{Q}^{(d)} \not\rightarrow \Delta\rho_1(\mathbf{x}'') > 0 \vee \cdots \vee \Delta\rho_{l-1}(\mathbf{x}'') > 0 \tag{4.19}$$

By Lemma 4.5, we can construct $g_{d+1} = \rho_l + \sum_{i=1}^{l-1} \mu_i \rho_i$ such that

$$\mathbf{x}'' \in \mathcal{Q}^{(d)} \rightarrow \Delta g_{d+1}(\mathbf{x}'') \geq 1. \tag{4.20}$$

Moreover, g_{d+1} is non-negative over $\mathcal{Q}^{(d)}$ and thus it ranks all $\mathcal{Q}^{(d)}$. Now, by construction, $\tau' = \langle g'_1 + 1, \ldots, g'_d + 1, g_{d+1} \rangle$ is a MΦRF for $\mathcal{Q}$. □

LEMMA 4.7. *If $\mathcal{Q}$ has a MΦRF of depth d then $\mathcal{Q}' = F(\mathcal{Q})$ has a MΦRF of depth at most $d - 1$.*

Proof. Let $\tau = \langle \rho_1, \ldots, \rho_k \rangle$ be an MΦRF for $\mathcal{Q}$, of optimal depth $k \leq d$. Without lose of generality we may assume that ρ_k is non-negative on all $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$, this follows immediately from the definition of nested MΦRF [Leike and Heizmann 2015], which is a special case of MΦRF in which the last component is non-negative, and the fact that for the case of SLC loops existence of a MΦRF implies the existence of a nested MΦRF [Ben-Amram and Genaim 2017] of the same optimal depth. Clearly $\tau' = \langle \rho_1, \ldots, \rho_{k-1} \rangle$ is a MΦRF for $\mathcal{Q} \wedge \Delta\rho_k(\mathbf{x}'') \leq 0$ since transitions that are ranked by ρ_k are eliminated. Now since ρ_k is a conic combination of the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ we have

$$\mathcal{Q}' = F(\mathcal{Q}) \subseteq \mathcal{Q} \wedge \Delta\rho_k(\mathbf{x}'') \leq 0$$

and thus τ' is a MΦRF for $\mathcal{Q}'$ as well. □

LEMMA 4.8. *$\mathcal{Q}$ has a MΦRF of depth d iff $F^d(\mathcal{Q}) = \emptyset$.*

Proof. For the first direction, suppose that $\mathcal{Q}$ has a MΦRF of depth at most d, then applying Lemma 4.7 iteratively we must reach $F^k(\mathcal{Q}) = \emptyset$ for some $k \leq d$, thus $F^d(\mathcal{Q}) = \emptyset$. For the other direction, suppose $F^d(\mathcal{Q}) = \emptyset$, then using Lemma 4.6 we can construct a MΦRF of depth d. It is easy to see that if $F^d(\mathcal{Q}) = \emptyset$ and $F^{d-1}(\mathcal{Q}) \neq \emptyset$, then d is the optimal depth. □

Procedure DecideMLRF($\mathcal{Q}$) of Algorithm 3 implements the above idea, it basically applies F (Lines 3-4) iteratively until it either reaches an empty set (Line 1) or stabilises (Line 5). If it returns $\emptyset$ then $\mathcal{Q}$ has a MΦRF and we can construct one simply by invoking the polynomial-time procedure for synthesising nested MΦRFs as described by Ben-Amram and Genaim [2017], or construct one as in the proof of Lemma 4.6. Note that, by Lemma 4.8, if we bound the recursion depth by a parameter d, then the algorithm is actually a decision procedure for the existence of MΦRFs of depth at most d. The case in which it returns a non-empty set is discussed in Section 4.2.2.

The complexity of Algorithm 3 is exponential since computing the generators at Line 3 might take exponential time. In Section 4.3, we provide a polynomial-time implementation that does not require computing the generators. For implementation purposes, it is worth mentioning as well that computing the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ at Line 3 can be done without computing $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$ as follows. Assume that $\mathcal{Q} \equiv [A''\mathbf{x} \leq \mathbf{c}'']$, using Farkas' lemma [Schrijver 1986, p. 93] we get that $(\vec{a}, b) \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ iff it satisfies

$$\mathcal{C} \equiv [\vec{\lambda}A'' = (-\vec{a}, \vec{0}) \wedge \vec{\lambda}\mathbf{c} \leq b \wedge \vec{\lambda}I \geq \vec{0}]$$

for some $\vec{\lambda}$ where I is the identity matrix of appropriate dimension. $\mathcal{C}$ defines a cone over the coordinates $\vec{\lambda}, \vec{a}$ and b. We can compute its generators using a standard algorithm, and then take the coordinates that correspond to $(\vec{a}, b)$.

EXAMPLE 4.9. Let us apply Algorithm 3 to the loop of Figure 2.2. We start by calling DecideMLRF with the transition polyhedron $\mathcal{Q} = \{x + z \geq 0, x' = x + y, y' = y + z, z' = z - 1\}$ and proceed as follows ($\mathcal{Q}_i$ represents the polyhedron passed in the ith call to DecideMLRF):

$\mathcal{Q}_i$	Generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_i)^{\#}$
$\mathcal{Q}_0$=$\mathcal{Q}$	$\{(\mathbf{1},\mathbf{0},\mathbf{1},\mathbf{0}), (\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1})\}$
$\mathcal{Q}_1$=$\mathcal{Q}_0 \wedge (x + z) - (x' + z') \leq 0$	$\{(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{-1}), (1,0,1,0), (0,0,0,1)\}$
$\mathcal{Q}_2$=$\mathcal{Q}_1 \wedge y - y' \leq 0$	$\{(\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{0}), (0,1,0,-1), (1,0,1,0), (0,0,0,1)\}$
$\mathcal{Q}_3$=$\mathcal{Q}_2 \wedge z - z' \leq 0 = \emptyset$	

Let us explain the above steps:

- $\mathcal{Q}_0$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_0)^{\#}$, which define the non-negative functions $\rho_1(x, y, z) = x + z$ and $\rho_2(x, y, z) = 1$, and then compute $\mathcal{Q}_1 = \mathcal{Q}_0 \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \Delta\rho_2(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_0$ we recursively call DecideMLRF($\mathcal{Q}_1$).

- $\mathcal{Q}_1$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_1)^{\#}$, which define the non-negative function $\rho_3(x, y, z) = y - 1$, and then compute $\mathcal{Q}_2 = \mathcal{Q}_1 \wedge \Delta\rho_3(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_1$ we recursively call DecideMLRF($\mathcal{Q}_2$). Note that the only new generator wrt. the previous iteration is the one in bold font, the others can be ignored since they have been used already when computing $\mathcal{Q}_1$.

- $\mathcal{Q}_2$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_2)^{\#}$, which define the non-negative function $\rho_4(x, y, z) = z$, and then compute $\mathcal{Q}_3 = \mathcal{Q}_2 \wedge \Delta\rho_4(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_2$ we recursively call DecideMLRF($\mathcal{Q}_3$).

- $\mathcal{Q}_3$ is empty, so we return $\emptyset$.

Since we have reached an empty set in 3 iterations, we conclude that the loop of Figure 2.2 has a MΦRF of optimal depth 3, e.g., $\langle z+1, y+1, x+z+1\rangle$. □

Let us discuss now the case in which the variables range over the integers, i.e., the set of integer transitions $I(\mathcal{Q})$. It is know that $I(\mathcal{Q})$ has a MΦRF iff the integer hull $\mathcal{Q}_I$ of $\mathcal{Q}$ has a MΦRF (over the rationals) [Ben-Amram and Genaim 2017, Sect. 5]. This leads to the following lemma.

LEMMA 4.10. *$I(\mathcal{Q})$ has a MΦRF of depth d iff $F^d(\mathcal{Q}_I) = \emptyset$.*

EXAMPLE 4.11. Consider a SLC loop defined [Ben-Amram and Genaim 2017] by the transition polyhedron:

$$\mathcal{Q} = \{x \geq y, x + y \geq 1, z \geq 0, y' = y - 2x + 1, z' = z + 10y + 9\}$$

When interpreted over the rationals, the loop does not terminate, e.g. for $(\frac{1}{2}, \frac{1}{2}, 0)$. When interpreted over the integers, computing the integer hull of the transition polyhedron results in $\mathcal{Q}_I = \mathcal{Q} \wedge x \geq 1$. Calling DecideMLRF($\mathcal{Q}_I$) proceeds as follows

$\mathcal{Q}_i$	Generators of $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}_i)^{\#}$
$\mathcal{Q}_0$=$\mathcal{Q}_I$=$\mathcal{Q} \wedge x \geq 1$	$\{(\mathbf{1},\mathbf{0},\mathbf{0},\mathbf{-1}), (\mathbf{1},\mathbf{1},\mathbf{0},\mathbf{-1}),$ $(\mathbf{1},\mathbf{-1},\mathbf{0},\mathbf{0}), (\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{0}), (\mathbf{0},\mathbf{0},\mathbf{0},\mathbf{1})\}$
$\mathcal{Q}_1 = \mathcal{Q}_0 \wedge x - x' \leq 0 \wedge$ $(x+y) - (x'+y') \leq 0 \wedge$ $(x-y) - (x'-y') \leq 0 \wedge$ $z - z' \leq 0$	$\{(\mathbf{0},\mathbf{10},\mathbf{0},\mathbf{9}), (1,0,0,-1), (1,1,0,-1),$ $(1,-1,0,0), (0,0,1,0), (0,0,0,1)\}$
$\mathcal{Q}_2$=$\mathcal{Q}_1 \wedge 10y - 10y' \leq 0 = \emptyset$	

Let us explain the above steps:

- $\mathcal{Q}_0$ is not empty, so we compute the generators of $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}_0)^{\#}$, which define the non-negative functions $\rho_1(x,y,z) = x-1$, $\rho_2(x,y,z) = x+y-1$, $\rho_3(x,y,z) = x-y$, $\rho_4(x,y,z) = z$ and $\rho_5(x,y,z) = 1$, and then compute

  $$\mathcal{Q}_1 = \mathcal{Q}_0 \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \cdots \wedge \Delta\rho_5(\mathbf{x}'') \leq 0$$

  and since it differs from $\mathcal{Q}_0$ we recursively call DecideMLRF($\mathcal{Q}_1$).

- $\mathcal{Q}_1$ is not empty, so we compute the generators of $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}_1)^{\#}$, which define the non-negative function $\rho_6(x,y,z) = 10y + 9$, and then compute $\mathcal{Q}_2 = \mathcal{Q}_1 \wedge \Delta\rho_6(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_1$ we recursively call DecideMLRF($\mathcal{Q}_2$).

- $\mathcal{Q}_2$ is empty, so we return $\emptyset$.

Since we have reached an empty set in 2 iterations, we conclude that the loop defined by $\mathcal{Q}$, when interpreted over the integers $I(\mathcal{Q})$, has a MΦRF of optimal depth 2. Indeed $\mathcal{Q}_I$ has the MΦRF $\langle 10y, z\rangle$. □

4.2.2 Inference of Recurrent Sets

Next, we discuss the case in which DecideMLRF($\mathcal{Q}$) returns a non-empty set of transition $\mathcal{S} \subseteq \mathcal{Q}$ (Line 5), and show that $\mathcal{S}$ is actually a set of transitions that witnesses non-termination of $\mathcal{Q}$. In Chapter 6, we discuss practical aspects of the use of Algorithm 3 for proving non-termination by means of experimental evaluation.

Let us start by specialising Definition 2.18 of a recurrent set to the case of SLC loops, which will simplify the presentation below. Since a SLC loop represents a TS with a single non-trivial SCC formed by the edge $\mathtt{n} \xrightarrow{\mathcal{Q}} \mathtt{n}$, all tuples of a recurrent set Ω as in Definition 2.18 will include the same value $\mathtt{n}$ in the first component, and thus, we can identify Ω with the set of states $\mathcal{S} = \{\mathbf{x} \mid (\mathtt{n}, \mathbf{x}) \in \Omega\}$. The next definition goes one step further, and treat $\mathcal{S}$ as a set of transitions instead of a set of states.

Definition 4.12. We say $\mathcal{S} \subseteq \mathcal{Q}$ is a recurrent set of *transitions*, if $\mathtt{proj}_{\mathbf{x}'}(\mathcal{S}) \subseteq \mathtt{proj}_{\mathbf{x}}(\mathcal{S})$.

Clearly this new notion of recurrent sets is equivalent to that based on states, which can be obtained from this one by projecting on $\mathbf{x}$, i.e., $\mathtt{proj}_{\mathbf{x}}(\mathcal{S})$ is a recurrent set of states. We prefer to use this new notion since it makes some claims smother.

LEMMA 4.13. *Let $\mathcal{S} \subseteq \mathbb{Q}^{2n}$ be a polyhedron, if $\mathcal{S} = F(\mathcal{S})$ then $\mathcal{S}$ is a recurrent set.*

Proof. According Definition 4.12, we need to show that $\mathtt{proj}_{\mathbf{x}'}(\mathcal{S}) \subseteq \mathtt{proj}_{\mathbf{x}}(\mathcal{S})$. Since $\mathtt{proj}_{\mathbf{x}}(\mathcal{S})$ and $\mathtt{proj}_{\mathbf{x}'}(\mathcal{S})$ are closed convex sets in this particular case, each is an intersection of half-spaces that are defined by $\mathtt{proj}_{\mathbf{x}}(\mathcal{S})^{\#}$ and $\mathtt{proj}_{\mathbf{x}'}(\mathcal{S})^{\#}$, namely

$$\begin{aligned} \mathtt{proj}_{\mathbf{x}}(\mathcal{S}) = &\ \wedge\{\vec{a} \cdot \mathbf{x} + b \geq 0 \mid (\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}}(\mathcal{S})^{\#}\} \\ \mathtt{proj}_{\mathbf{x}'}(\mathcal{S}) = &\ \wedge\{\vec{a} \cdot \mathbf{x} + b \geq 0 \mid (\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}'}(\mathcal{S})^{\#}\} \end{aligned}$$

Thus, to show that $\mathtt{proj}_{\mathbf{x}'}(\mathcal{S}) \subseteq \mathtt{proj}_{\mathbf{x}}(\mathcal{S})$ it is enough to show that $\mathtt{proj}_{\mathbf{x}}(\mathcal{S})^{\#} \subseteq \mathtt{proj}_{\mathbf{x}'}(\mathcal{S})^{\#}$.

Let $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}}(\mathcal{S})^{\#}$, we show that $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}'}(\mathcal{S})^{\#}$ as well. Define the function $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$. Since $\mathcal{S} = F(\mathcal{S})$, by definition of F we have

$$\mathbf{x}'' = (\mathbf{x}, \mathbf{x}') \in \mathcal{S} \Rightarrow \rho(\mathbf{x}) - \rho(\mathbf{x}') \leq 0 \tag{4.21}$$

which together with the fact that ρ is non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{S})$ implies that $\rho(\mathbf{x}') \geq 0$ holds for any $\mathbf{x}' \in \mathtt{proj}_{\mathbf{x}'}(\mathcal{S})$, and thus $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}'}(\mathcal{S})^{\#}$. □

Corollary 4.14. If DecideMLRF($\mathcal{Q}$) returns $\mathcal{S} \neq \emptyset$ then $\mathcal{S}$ is a recurrent set, and thus $\mathcal{Q}$ is non-terminating.

Proof. This follows from Lemma 4.13, since the algorithm returns a non-empty set $\mathcal{S} \subseteq \mathcal{Q}$ iff it finds one such that $\mathcal{S} = F(\mathcal{S})$ (Line 5 of DecideMLRF). □

EXAMPLE 4.15. Let us apply Algorithm 3 to the following loop [Tiwari 2004]:

$$\texttt{while } (x - y \geq 1) \texttt{ do } x' = -x + y,\ y' = y \tag{4.22}$$

This loop does not terminate, e.g., for $x = -1, y = -2$. We call DecideMLRF with $\mathcal{Q} = \{x - y \geq 1, x' = -x + y, y' = y\}$, and proceed as in Example 4.9:

$\mathcal{Q}_i$	Generators of $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}_i)^{\#}$
$\mathcal{Q}_0 = \mathcal{Q}$	$\{(\mathbf{1}, -\mathbf{1}, -\mathbf{1}), (\mathbf{0}, \mathbf{0}, \mathbf{1})\}$
$\mathcal{Q}_1 = \mathcal{Q}_0 \wedge (x - y) - (x' - y') \leq 0$	$\{(-\mathbf{2}, \mathbf{1}, \mathbf{0}), (1, -1, -1), (0, 0, 1)\}$
$\mathcal{Q}_2 = \mathcal{Q}_1 \wedge (-2x + y) - (-2x' + y') \leq 0$	$\{(\mathbf{2}, -\mathbf{1}, \mathbf{0}), (-\mathbf{1}, \mathbf{0}, -\mathbf{1}), (-2, 1, 0), (0, 0, 1)\}$
$\mathcal{Q}_3 = \mathcal{Q}_2 \wedge (2x - y) - (2x' - y') \leq 0 \wedge$ $(-x) - (-x') \leq 0$	

Let us explain the above steps:

- $\mathcal{Q}_0$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_0)^{\#}$, which define the non-negative functions $\rho_1(x,y) = x - y - 1$ and $\rho_2(x,y) = 1$, and then compute $\mathcal{Q}_1 = \mathcal{Q}_0 \wedge \Delta\rho_1(\mathbf{x}'') \leq 0 \wedge \Delta\rho_2(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_0$ we recursively call DecideMLRF($\mathcal{Q}_1$).

- $\mathcal{Q}_1$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_1)^{\#}$, which define the non-negative function $\rho_3(x,y) = -2x + y$, and then compute $\mathcal{Q}_2 = \mathcal{Q}_1 \wedge \Delta\rho_3(\mathbf{x}'') \leq 0$; and since it differs from $\mathcal{Q}_1$ we invoke DecideMLRF($\mathcal{Q}_2$).

- $\mathcal{Q}_2$ is not empty, so we compute the generators of $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_2)^{\#}$, which define the non-negative functions $\rho_4(x,y) = 2x - y$ and $\rho_5(x,y) = -x - 1$, and then compute $\mathcal{Q}_3 = \mathcal{Q}_2 \wedge \Delta\rho_4(\mathbf{x}'') \leq 0 \wedge \Delta\rho_5(\mathbf{x}'') \leq 0$; and since it is equal to $\mathcal{Q}_2$ (because $\Delta\rho_4(\mathbf{x}'') \leq 0$ and $\Delta\rho_5(\mathbf{x}'') \leq 0$ are implied by $\mathcal{Q}_2$) we return $\mathcal{Q}_2$.

Thus, $\mathcal{Q}_2$ is a recurrent set of transitions and we conclude that Loop (4.22) is not terminating. Projecting $\mathcal{Q}_2$ on x and y we get $\{x \leq -1, 2x - y = 0\}$, which is the corresponding recurrent set of states.

We remark that Loop (4.22) has a fixed point $(-1,-2)$, i.e., from state $x = -1, y = -2$ we have a transition to $x = -1, y = -2$. The algorithm also detects non-termination of loops that do not have fixed points. For example, if we change $y' = y$ in Loop (4.22) by $y' = y - 1$, we obtain a recurrent set of transitions $\mathcal{S}$ such that $\texttt{proj}_{\mathbf{x}}(\mathcal{S}) = \{-2y \geq 3, 4x - 2y = 1\}$. □

Now that we have seen the possible outcomes of the algorithm (in case it terminates), we see that this approach reveals an interesting relation between seeking MΦRFs and seeking recurrent sets. A possible view is that the algorithm seeks a recurrent set (of a particular form) and when it concludes that no such set exists, i.e., reaching $\emptyset$, we can construct a MΦRF.

The recurrent sets inferred by Algorithm 3 belong to a narrower class than that of Definition 4.12. For a polyhedral set $\mathcal{S}$ to be a recurrent set, Definition 4.12 requires that $\texttt{proj}_{\mathbf{x}'}(\mathcal{S}) \subseteq \texttt{proj}_{\mathbf{x}}(\mathcal{S})$, i.e., any $\rho(\mathbf{x}) \geq 0$ that is satisfied by $\texttt{proj}_{\mathbf{x}}(\mathcal{S})$ is also satisfied by $\texttt{proj}_{\mathbf{x}'}(\mathcal{S})$. On the other hand, in our recurrent sets, ρ is required to be monotonic as well, i.e., $\rho(\mathbf{x}') \geq \rho(\mathbf{x})$ for any $(\mathbf{x}, \mathbf{x}') \in \mathcal{S}$.

EXAMPLE 4.16. Consider the following SLC loop:

$$\texttt{while } (x \geq 0) \texttt{ do } x' = 1 - x \tag{4.23}$$

The largest recurrent set of transitions for this loop is $\{x \geq 0, x \leq 1, x' = 1 - x\}$, and it is not monotonic. Algorithm 3 infers the largest *monotonic* recurrent set $\{x = \frac{1}{2}, x' = \frac{1}{2}\}$, where in the first iteration it eliminates all transitions for which $x - x' > 0$, i.e., those for which $x \in (\frac{1}{2}, \infty)$, and in the second those for which $(-x) - (-x') > 0$, i.e., those for which $x \in [0, \frac{1}{2})$. □

At this point, it is natural to explore the difference between the two kinds of recurrent sets. The most intriguing question is if non-terminating SLC loops always have monotonic recurrent sets (either polyhedral or closed convex in general). This is clearly true for loops that have a fixed point, i.e., there is $\mathbf{x}$ such that $(\mathbf{x}, \mathbf{x}) \in \mathcal{Q}$, however, this question is left open for the general case. We note that the *geometric non-termination argument*

introduced by Leike and Heizmann [2018] is also related to monotonic recurrent sets. Specifically, it is easy to show that in some cases (when the non-negative coefficients μ_i and λ_i of Definition 5 of Leike and Heizmann [2018], are either 0 or at least 1), we can construct a monotonic recurrent set. Moreover, for the case in which μ_i and λ_i are between 0 and 1, we believe that the loop must have a fixpoint which in turn defines a monotonic recurrent set.

Let us discuss now the case of integer loops. First, we note that the difference between the two kinds of recurrent sets is clear in the integer case: Example 4.16 shows that the Loop (4.23) has a recurrent set $\{(0,1),(1,0)\}$, but does not have a monotonic recurrent set. Apart from this difference, a natural question is whether the recurrent set $\mathcal{S}$ returned by DecideMLRF($\mathcal{Q}_I$), or more precisely $I(\mathcal{S})$, witnesses non-termination of $I(\mathcal{Q})$. This is not true in general (see Example 4.18 below), however, there are practical cases for which it is true.

LEMMA 4.17. *Let $\mathcal{Q}$ be a SLC loop with affine update $\mathbf{x}' = U\mathbf{x} + \mathbf{c}$, and assume the coefficients U and $\mathbf{c}$ are integer. If $\mathcal{S}$ is a recurrent set of $\mathcal{Q}$, and $I(\mathcal{S})$ is not empty, then $I(\mathcal{S})$ is recurrent for $I(\mathcal{Q})$.*

Proof. Since the update is affine with integer coefficients, it follows that any state in $\mathtt{proj}_{\mathbf{x}}(I(\mathcal{S}))$ has a successor in $\mathtt{proj}_{\mathbf{x}'}(I(\mathcal{S})) \subseteq \mathtt{proj}_{\mathbf{x}}(I(\mathcal{S}))$, which is the definition of a recurrent set. □

In the context of the above lemma, assuming that $\mathcal{S} = \text{DecideMLRF}(\mathcal{Q}_I)$, if $\mathcal{S} \neq \emptyset$ and $I(\mathcal{S}) = \emptyset$ all we can conclude (when the algorithm is applied to $\mathcal{Q}_I$) is that $I(\mathcal{Q})$ does not have a MΦRF, we cannot conclude anything about non-termination as in the rational case. For example, for the loop $\mathcal{Q}_I = \mathcal{Q} = \{x \geq 0, x' = 10 - 2x\}$ we have $\mathcal{S} = \{(3\frac{1}{3}, 3\frac{1}{3})\}$ and $I(\mathcal{S}) = \emptyset$ and the loop is terminating over the integers, and for the loop $\mathcal{Q}_I = \mathcal{Q} = \{x \geq 0, x' = 1 - x\}$ we have $\mathcal{S} = \{(\frac{1}{2}, \frac{1}{2})\}$ and $I(\mathcal{S}) = \emptyset$ and the loop is non-terminating over the integers.

We note that one can allow some degree of non-determinism in Lemma 4.17, in particular, non-deterministically setting a variable to an arbitrary value. Note that tools for proving non-termination that are based on the use of Farkas' lemma [Gupta et al. 2008; Larraz et al. 2014], impose similar restrictions to guarantee that the recurrent set is valid over the integers. The next example demonstrates that Lemma 4.17 does not extend to SLC loops in general, even when the algorithm is applied to the integer hull $\mathcal{Q}_I$. This is because it is not guaranteed that any integer state $\mathbf{x} \in I(\mathtt{proj}_{\mathbf{x}}(\mathcal{S}))$ has an integer successor $\mathbf{x}' \in I(\mathtt{proj}_{\mathbf{x}'}(\mathcal{S}))$.

EXAMPLE 4.18. Consider the following loop

$$\texttt{while } (x \geq 2) \texttt{ do } x' = \frac{3}{2}x \tag{4.24}$$

which is specified by $\mathcal{Q} = \{x \geq 2, x' = \frac{3}{2}x\}$. It is clearly non-terminating over the rationals, for any $x \geq 2$, and is terminating over the integers because

(*i*) starting from x odd, the next state $\frac{3}{2}x$ is not integer, and;

(*ii*) starting from x even, then for some $i > 0$ the value $\frac{3^i}{2^i}x$ is odd (because $\frac{x}{2^i}$ must be odd some $i > 0$ and 3^i is odd), and then the next state is not integer.

The algorithm returns $\mathcal{Q}$ as a recurrent set, but $I(\mathcal{Q})$, which is not empty, is not a recurrent set as the loop is terminating over the integers. Note that the transition polyhedron is integral, i.e., $\mathcal{Q} = \mathcal{Q}_I$. □

4.2.3 Recurrent Sets for Transition Systems

In the previous section, we discussed how Algorithm 3 can be used to infer recurrent sets for SLC loops, however, to make it useful in practice, next we describe how to generalise it to prove non-termination of TSs, i.e., CFGs with more than one node and transition. This generalisation includes two different components:

(*i*) the first deals with finding a recurrent set for a given set of transitions (a set of edges in the CFGs); and

(*ii*) the second deals with proving that this recurrent set is actually reachable from the initial node.

Next, we define the notion of a *closed walk* that will be used to refer to some executions of interest.

Definition 4.19 (Closed Walk)**.** Let $\mathcal{T}$ be a TS. A *path* between two nodes $\mathtt{n}_1, \mathtt{n}_k \in N$ is a sequence of edges $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_1} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_2} \cdots \xrightarrow{\mathcal{Q}_k} \mathtt{n}_k$, such that, $\mathtt{n}_i \xrightarrow{\mathcal{Q}_i} \mathtt{n}_{i+1} \in E$. A *closed walk* is a path that starts and ends in the same node.

Note that the a closed walk might visit the same node or edge several times.

EXAMPLE 4.20. The following are closed walks for the CFG of Figure 2.9:

(*i*) $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_3} \mathtt{n}_1$

(*ii*) $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_4} \mathtt{n}_1$

(*iii*) $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_3} \mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2$

(*iv*) $\cdots$

It is clear that there are an infinite different closed walks. □

For finding recurrent sets we rely on the notion of *closed walks* as follows. Suppose we are given a closed walk

$$\mathtt{n}_1 \xrightarrow{\mathcal{Q}_1} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_2} \cdots \xrightarrow{\mathcal{Q}_{k-1}} \mathtt{n}_k \xrightarrow{\mathcal{Q}_k} \mathtt{n}_1 \tag{4.25}$$

and assume that variables in $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ are renamed such that the variables of $\mathcal{Q}_i$ are $(\mathbf{x}_i, \mathbf{x}_{i+1})$, i.e., the target variable of $\mathcal{Q}_i$ are the same as the source variable of $\mathcal{Q}_{i+1}$. Now define

$$\mathcal{Q} = \mathcal{Q}_1 \wedge \cdots \wedge \mathcal{Q}_k \tag{4.26}$$

This $\mathcal{Q}$ can be seen as a SLC loop, where $\mathbf{x}$ are $\mathbf{x}_1$ and $\mathbf{x}'$ are $\mathbf{x}_{k+1}$. The only difference from the definition of SLC loops is that here we have extra (intermediate) variables $\mathbf{x}_2, \ldots, \mathbf{x}_k$. In principle, we could convert it to a SLC loop simply by projecting on $\mathbf{x}_1$ and $\mathbf{x}_{k+1}$ in order to eliminate all extra variables, however, it is easy to see that Algorithm 3 is still valid even if we do not eliminate these extra variables. This is because the set of positive functions computed at Line 3 is the same for $\mathcal{Q}$ of (4.26) and for $\mathtt{proj}_{\mathbf{x}_1, \mathbf{x}_{k+1}}(\mathcal{Q})$. This shows that Algorithm 3 can be used to check non-termination of a given closed walk.

LEMMA 4.21. *Let* (4.25) *be closed walk in some SCC,* $\mathcal{Q}$ *defined as in* (4.26)*, and* $\mathcal{S} = \textsc{DecideMLRF}(\mathcal{Q})$ *such that* $\mathcal{S} \neq \emptyset$*. Then*

$$\{(\mathtt{n}_1, \mathbf{x}) \mid \mathbf{x} \in \mathtt{proj}_{\mathbf{x}_1}(\mathcal{S})\} \cup \cdots \cup \{(\mathtt{n}_k, \mathbf{x}) \mid \mathbf{x} \in \mathtt{proj}_{\mathbf{x}_k}(\mathcal{S})\}$$

is a recurrent set for that SCC.

Proof. It follows from the fact that $\textsc{DecideMLRF}(\mathtt{proj}_{\mathbf{x}_1,\mathbf{x}_{k+1}}(\mathcal{Q}))$ returns the recurrent set $\mathtt{proj}_{\mathbf{x}_1,\mathbf{x}_{k+1}}(\mathcal{S})$. □

Our procedure for proving non-termination is based on enumerating closed walks in a given set of transitions (a connected subgraph that we failed to prove terminating, typically a subset of a SCC of the transition system), and then using Algorithm 3 to find recurrent sets of each closed walk as it is generated. In addition, once a recurrent set $\mathcal{S}$ is found we check that it is actually reachable as follows: we collect all constraints on a path from the initial node to node $\mathtt{n}_1$ of the closed walk, and we ask an SMT solver to find a solution for these constraints together with $\mathcal{S}$. If it finds one, then the TS is non-terminating for the corresponding initial input. We could also use any off-the-shelf tool for reachability analysis.

EXAMPLE 4.22. Consider the TS of Figure 2.9. We fail to prove termination of the SCC formed by nodes $\mathtt{n}_1$ and $\mathtt{n}_2$. Suppose that we start to enumerate closed walks in the order in which they are generated in Example 4.20:

1. For the first one $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_3} \mathtt{n}_1$, we compute the recurrent set $\{x \geq 0, y \geq 1\}$, but node $\mathtt{n}_1$ is not reachable with $y \geq 1$.

2. For the second one $\mathtt{n}_1 \xrightarrow{\mathcal{Q}_2} \mathtt{n}_2 \xrightarrow{\mathcal{Q}_4} \mathtt{n}_1$, we compute the recurrent set $\{x \geq 0, y \leq 0\}$, which is clearly reachable using the path $\mathtt{n}_0 \xrightarrow{\mathcal{Q}_0} \mathtt{n}_1$.

To compute the recurrent set for the second case we proceed as follows: we first rename the variables of $\mathcal{Q}_2$ to ($\mathbf{x}_1$,$\mathbf{x}_2$) and those of $\mathcal{Q}_4$ to ($\mathbf{x}_2$,$\mathbf{x}_3$) – where we treat $\mathbf{x}_1$ as $\mathbf{x}$, and $\mathbf{x}_3$ as $\mathbf{x}'$ – and then call DecideMLRF with $\mathcal{Q}_2 \wedge \mathcal{Q}_4$ □

As a final remark, for the case of integer variables, Lemma 4.17 can be easily extended to the case of SLC loop with intermediate variables as in (4.26), and, moreover, in this case we require the SMT query that checks reachability to find an integer solution.

4.2.4 Cases in which Algorithm 3 does not Terminate

When Algorithm 3 terminates, it either finds a MΦRF or proves non-termination of the given loop. This means that if applied to a terminating loop that has no MΦRF, then Algorithm 3 will not terminate. This non-terminating behaviour shows that our algorithm is not complete, however, it can be used to show that a SLC loop does not have a MΦRF.

EXAMPLE 4.23. Consider the following loop

$$\texttt{while } (x \geq y, y \geq 1) \texttt{ do } x' = 2x,\ y' = 3y \tag{4.27}$$

which is specified by $\mathcal{Q} = \{x \geq y, y \geq 1, x' = 2x, y' = 3y\}$, and is terminating [Leike and Heizmann 2015]. Let $\mathcal{Q}_i$ be the SLC loop passed to DecideMLRF at the ith iteration, with $\mathcal{Q}_0 = \mathcal{Q}$. The cone $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}_i)^{\#}$ is generated by the rays $(0, 1)$ and $(1, -2^i)$. It is easy to show (by induction) that $\mathcal{Q}_i = \{x \geq 2^i y,\ y \geq 1,\ x' = 2x,\ y' = 3y\}$, which satisfies $\mathcal{Q}_i \neq \emptyset$ and $\mathcal{Q}_{i+1} \subset \mathcal{Q}_i$. This implies that the algorithm does not terminate, and thus the loop does not have a MΦRF. □

The following example shows that Algorithm 3 might not terminate also when applied to non-terminating loops.

EXAMPLE 4.24. Consider the following loop

$$\texttt{while } (x + y \geq 3) \texttt{ do } x' = 3x - 2,\ y' = 2y \tag{4.28}$$

which is specified by $\mathcal{Q} = \{x + y \geq 3, x' = 3x - 2, y' = 2y\}$, and is non-terminating [Leike and Heizmann 2018] and has a monotonic recurrent set, e.g., $\mathcal{S} = \{x \geq 1,\ y' = 2y,\ x' = 3x - 2\}$. From considerations similar to those of Example 4.23, it is easy to show that Algorithm 3 does not terminate on this loop. □

When the algorithm does not terminate, the iterates $F^i(\mathcal{Q})$ converge to

$$\mathcal{Q}_\omega = \cap_{i \geq 0} F^i(\mathcal{Q}). \tag{4.29}$$

For example, for Loop (4.27), which is terminating, we have $\mathcal{Q}_\omega = \emptyset$, and for Loop (4.28), which is non-terminating, we have $\mathcal{Q}_\omega = \{x \geq 1, y' = 2y, x' = 3x - 2\}$ which is a monotonic recurrent set. Some natural questions arise at this point:

(*i*) is it true that $\mathcal{Q}_\omega = \emptyset$ iff $\mathcal{Q}$ is terminating?

(*ii*) is it true that if $\mathcal{Q}_\omega \neq \emptyset$ then it is a (monotonic) recurrent set?

For *deterministic* loops, it is easy to show that termination implies $\mathcal{Q}_\omega = \emptyset$, and that if $\mathcal{Q}_\omega \neq \emptyset$ then $\mathcal{Q}_\omega$ is a monotonic recurrent set. The general questions, however, are left open.

Once we start to explore properties of $\mathcal{Q}_\omega$, there is a difference between real and rational loops as we demonstrate in the next example, this difference does not exists when the algorithm terminate.

EXAMPLE 4.25. Consider the following loop

$$\texttt{while } (4x + y \geq 1) \texttt{ do } x' = -2x + 4y,\ y' = 4x \tag{4.30}$$

which is specified by $\mathcal{Q} = \{4x + y \geq 1, x' = -2x + 4y, y' = 4x\}$. It is terminating over the rationals and non-terminating over the reals [Braverman 2006]. The algorithm does not terminate when applied to this loop. If each $\mathcal{Q}_i$ is considered as a set of rational transitions, then $\mathcal{Q}_w = \emptyset$, however, if we include also the irrational transitions then $\mathcal{Q}_w$ would be the closed convex set

$$\{\mu(\sqrt{17} - 1, 4, (\sqrt{17} - 1)^2, 4(\sqrt{17} - 1)) \mid \mu \geq \frac{1}{\sqrt{17} + 3}\}$$

which is a monotonic recurrent set. □

4.3 The Displacement Polyhedron

In this section, we introduce an alternative representation for SLC loops, that we refer to as the *displacement polyhedron*, and show that Algorithm 3, or more precisely the check $F^k(\mathcal{Q}) = \emptyset$, has a simple encoding in this representation that can be performed in polynomial time, specifically, we show that it is equivalent to checking for unsatisfiability

of a particular linear constraint system. Note that we already know that deciding the existence of a MΦRF of depth d can be done in polynomial time [Ben-Amram and Genaim 2017], so in this sense we do not provide any new knowledge. However, apart from the efficient encoding of the check $F^k(\mathcal{Q}) = \emptyset$, the new formulation has some important advantages:

- Unlike existing algorithms for inferring MΦRFs [Ben-Amram and Genaim 2017; Leike and Heizmann 2018], it allows synthesising witnesses for the non-existence of a MΦRF of a given depth (any solution of the corresponding linear constraint system mentioned above) which explains why a loop does not have a MΦRF of the given depth – see Section 4.3.1.
- It shows that MΦRFs have an interesting application for conditional termination – see Section 4.3.2.
- Some non-trivial observations about termination and non-termination of SLC loops are made straightforward through this representation – see Section 4.3.3.
- It provides a new tool for addressing the general MΦRF problem, i.e., without a depth bound, that is still open – see Section 4.3.4.

In what follows, we first define the notion of the *displacement polyhedron*, show how the check $F^d(\mathcal{Q}) = \emptyset$ can be encoded in this representation, and then discuss each of the above points.

Definition 4.26. Given a SLC loop $\mathcal{Q} \subseteq \mathbb{Q}^{2n}$, we define its corresponding *displacement polyhedron* as $\mathcal{R} = \mathtt{proj}_{\mathbf{x},\mathbf{y}}(\mathcal{Q} \wedge \mathbf{x}' = \mathbf{x} + \mathbf{y}) \subseteq \mathbb{Q}^{2n}$.

Note that the projection drops $\mathbf{x}'$. Intuitively, an execution step using $\mathcal{Q}$ starts from a state $\mathbf{x}$, and chooses a state $\mathbf{x}'$ such that $\binom{\mathbf{x}}{\mathbf{x}'} \in \mathcal{Q}$. To perform the step using $\mathcal{R}$, select $\mathbf{y}$ such that $\binom{\mathbf{x}}{\mathbf{y}} \in \mathcal{R}$ and let the new state be $\mathbf{x} + \mathbf{y}$. By definition, we obtain the same transitions. The constraint representation of $\mathcal{R}$ can be derived from that of $\mathcal{Q}$ as follows. Let $\mathcal{Q} \equiv [A''\binom{\mathbf{x}}{\mathbf{x}'} \leq \mathbf{c}'']$ where A'' is the matrix below on the left (see Section 2.2.2), then $\mathcal{R} \equiv [R\binom{\mathbf{x}}{\mathbf{y}} \leq \mathbf{c}'']$ where R is the matrix below on the right:

$$A'' = \begin{pmatrix} B & 0 \\ A & A' \end{pmatrix} \qquad R = \begin{pmatrix} B & 0 \\ A + A' & A' \end{pmatrix} \tag{4.31}$$

EXAMPLE 4.27. As notation, let $\mathbf{x} = (x, y, z)^{\mathrm{T}}$ and $\mathbf{y} = (y_1, y_2, y_3)^{\mathrm{T}}$. Consider the loop in Figure 2.2 defined by $\mathcal{Q} = \{x + z \geq 0, x' = x + y, y' = y + z, z' = z - 1\}$. The corresponding displacement polyhedron is $\mathcal{R} = \{x + z \geq 0, y_1 = y, y_2 = z, y_3 = -1\}$. □

We will show that the displacement polyhedron $\mathcal{R}_k$ of $\mathcal{Q}_k = F^k(\mathcal{Q})$ is equivalent to the following polyhedron projected onto $\mathbf{x}$ and $\mathbf{y}_0$

$$\widehat{\mathcal{R}}_k \equiv \; R\tbinom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}'' \;\wedge\; R\tbinom{\mathbf{y}_0}{\mathbf{y}_1} \leq \mathbf{0} \wedge\; R\tbinom{\mathbf{y}_1}{\mathbf{y}_2} \leq \mathbf{0} \;\wedge\; \ldots \;\wedge\; R\tbinom{\mathbf{y}_{k-1}}{\mathbf{y}_k} \leq \mathbf{0} \tag{4.32}$$

Now since, by Definition 4.26, $\mathcal{Q}_k$ is empty iff $\mathcal{R}_k$ is empty, the check $F^k(\mathcal{Q}) = \emptyset$ is reduced to checking that (4.32) is empty, which can be done in polynomial time in the bit-size of the constraint representation of $\mathcal{Q}$ and the parameter k. It is important to observe that the first conjunct $R\binom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}''$ of (4.32) is actually $\mathcal{R}$, and that each $R\binom{\mathbf{y}_i}{\mathbf{y}_{i+1}} \leq \mathbf{0}$ is actually $\mathtt{rec.cone}(\mathcal{R})$. Observe also how the conjuncts of (4.32) are connected, i.e., that the lower part of the variables vector of each conjunct is equal to the upper part of the next one. We first show how $\mathcal{R}_{k+1}$ can be obtained from $\mathcal{R}_k$ similarly to $\mathcal{Q}_{k+1} = F(\mathcal{Q}_k)$.

LEMMA 4.28. *Let* $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$ *generate the cone* $\texttt{proj}_{\mathbf{x}}(\mathcal{R})^{\#}$. *Then* $\mathcal{R}_{k+1} = \mathcal{R}_k \wedge -\vec{a}_1 \cdot \mathbf{y} \leq 0 \wedge \cdots \wedge -\vec{a}_l \cdot \mathbf{y} \leq 0$.

Proof. Follows from the fact that $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_k) = \texttt{proj}_{\mathbf{x}}(\mathcal{R}_k)$, and thus $\texttt{proj}_{\mathbf{x}}(\mathcal{Q}_k)^{\#}$ and $\texttt{proj}_{\mathbf{x}}(\mathcal{R}_k)^{\#}$ are the same, and that for $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$ we have $\Delta\rho(\mathbf{x}'') = \rho(\mathbf{x}) - \rho(\mathbf{x}') = -\vec{a} \cdot \mathbf{y}$, by definition of the displacement polyhedron. □

LEMMA 4.29. *Let* $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$ *generate the cone* $\texttt{proj}_{\mathbf{x}}(\mathcal{R})^{\#}$. *Then the condition* $-\vec{a}_1 \cdot \mathbf{y} \leq 0 \wedge \cdots \wedge -\vec{a}_l \cdot \mathbf{y} \leq 0$ *of Lemma 4.28 is equivalent to* $M\mathbf{y} \leq 0$, *where* M *is such that* $\texttt{proj}_{\mathbf{x}}(\mathcal{R}) \equiv [M\mathbf{x} \leq \mathbf{b}]$.

Proof. Consider $(\vec{a}, b) \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#} = \texttt{proj}_{\mathbf{x}}(\mathcal{R})^{\#}$. By Farkas' lemma, a function $f(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$ is non-negative over $\texttt{proj}_{\mathbf{x}}(\mathcal{R})$ iff there are non-negative $\vec{\lambda} = (\lambda_1, \ldots, \lambda_m)$ such that $\vec{\lambda} \cdot M = -\vec{a} \wedge \vec{\lambda} \cdot \mathbf{b} \leq b$. Note that any (non-negative) values for $\vec{\lambda}$ define corresponding values for $\vec{a}$ and b. Thus the valid values for $\vec{a}$ are all conic combinations of the rows of $-M$, i.e., this cone is generated by the rows of $-M$. Hence $-\vec{a}_1 \cdot \mathbf{y} \leq 0 \wedge \cdots \wedge -\vec{a}_l \cdot \mathbf{y} \leq 0$ is equivalent to $M\mathbf{y} \leq \mathbf{0}$. □

We use the above lemma to show that $\mathcal{R}_k$ can be represented as in (4.32), without the need to compute M explicitly. We first note that using Lemma 4.28 and Lemma 4.29 we get that $\mathcal{R}_{k+1} = \mathcal{R}_k \cap \mathcal{D}_k$, where

$$\begin{aligned}\mathcal{D}_k &= \{\tbinom{\mathbf{x}}{\mathbf{y}} \in \mathbb{Q}^{2n} \mid M\mathbf{y} \leq \mathbf{0}\} && (M \text{ as in Lemma 4.29})\\ &= \{\tbinom{\mathbf{x}}{\mathbf{y}} \in \mathbb{Q}^{2n} \mid \mathbf{y} \in \texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\mathcal{R}_k))\}\,.\end{aligned}$$

Before we proceed, let us state a lemma with a useful property of the projection operation for polyhedral sets.

LEMMA 4.30. *For a polyhedron* $\mathcal{P} \subseteq \mathbb{Q}^n$, $\texttt{proj}_{\mathbf{x}}(\texttt{rec.cone}(\mathcal{P})) = \texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\mathcal{P}))$.

Proof. A polyhedron $\mathcal{P}$ whose variables are split into two sets, $\mathbf{x}$ and $\mathbf{y}$, can be represented in the form $A\mathbf{x} + G\mathbf{y} \leq \mathbf{b}$ for matrices A, G and a vector $\mathbf{b}$ of matching dimensions. Then Theorem 11.11 of Conforti et al. [2010] states that $\texttt{proj}_{\mathbf{x}}(\mathcal{P})$ is specified by the constraints $V(\mathbf{b} - A\mathbf{x}) \geq \mathbf{0}$ for a certain matrix V determined by G only. From this it follows that $\texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\mathcal{P})) = \{\mathbf{x} \ : \ VA\mathbf{x} \leq \mathbf{0}\}$. But we can also apply the theorem to $\texttt{rec.cone}(\mathcal{P})$, which is specified by $A\mathbf{x} + G\mathbf{y} \leq \mathbf{0}$, and we get the same result $\texttt{proj}_{\mathbf{x}}(\texttt{rec.cone}(\mathcal{P})) = \{\mathbf{x} \ : \ VA\mathbf{x} \leq \mathbf{0}\}$. □

Now we are ready to state the main lemma, that relates $\mathcal{R}_k$ to (4.32).

LEMMA 4.31. $\mathcal{R}_k = \texttt{proj}_{\mathbf{x},\mathbf{y}_0}(\widehat{\mathcal{R}}_k)$ *where* $\widehat{\mathcal{R}}_k$ *is defined by* (4.32).

Proof. We use induction on k. For $k = 0$ the lemma states that $\mathcal{R}_0$ is specified by $R\tbinom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}''$, which is correct since by definition $\mathcal{R}_0 = \mathcal{R}$. Assume the lemma holds for $\mathcal{R}_k$, we prove it for $\mathcal{R}_{k+1} = \mathcal{R}_k \cap \mathcal{D}_k$. By the induction hypothesis,

$$\mathcal{R}_k = \{\ \tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid R\tbinom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}'' \ \wedge \ R\tbinom{\mathbf{y}_0}{\mathbf{y}_1} \leq \mathbf{0} \ \wedge \ \ldots \ \wedge \ R\tbinom{\mathbf{y}_{k-1}}{\mathbf{y}_k} \leq \mathbf{0} \ \} \tag{4.33}$$

and

$$\begin{aligned}\mathcal{D}_k =& \{\tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid \mathbf{y}_0 \in \texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\mathcal{R}_k))\} && \text{by definition}\\ =& \{\tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid \mathbf{y}_0 \in \texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\texttt{proj}_{\mathbf{x},\mathbf{y}_0}(\widehat{\mathcal{R}}_k)))\} && \text{by IH}\\ =& \{\tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid \mathbf{y}_0 \in \texttt{rec.cone}(\texttt{proj}_{\mathbf{x}}(\widehat{\mathcal{R}}_k))\}\\ =& \{\tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid \mathbf{y}_0 \in \texttt{proj}_{\mathbf{x}}(\texttt{rec.cone}(\widehat{\mathcal{R}}_k))\} && \text{by Lemma 4.30}\\ =& \{\tbinom{\mathbf{x}}{\mathbf{y}_0} \in \mathbb{Q}^{2n} \mid R\tbinom{\mathbf{y}_0}{\mathbf{y}_1} \leq \mathbf{0} \wedge R\tbinom{\mathbf{y}_1}{\mathbf{y}_2} \leq \mathbf{0} \wedge \cdots \wedge R\tbinom{\mathbf{y}_k}{\mathbf{y}_{k+1}} \leq \mathbf{0}\}\end{aligned}$$

Note that in the last step, we incorporated the recession cone of $\widehat{\mathcal{R}}_k$ as in (4.32), after renaming $\mathbf{y}_i$ to $\mathbf{y}_{i+1}$, and $\mathbf{x}$ to $\mathbf{y}_0$ just to make it easier to read in the next step. Now, let us compute $\mathcal{R}_{k+1} = \mathcal{R}_k \cap \mathcal{D}_k$. Note that any $\binom{\mathbf{x}}{\mathbf{y}_0} \in \mathcal{R}_{k+1}$ must satisfy the constraint $R\binom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}''$ that comes form $\mathcal{R}_k$. Adding this constraint to $\mathcal{D}_k$ above we clearly obtain a subset of $\mathcal{R}_k$, and thus

$$\mathcal{R}_{k+1} = \{\binom{\mathbf{x}}{\mathbf{y}_0} \mid R\binom{\mathbf{x}}{\mathbf{y}_0} \leq \mathbf{c}'' \wedge R\binom{\mathbf{y}_0}{\mathbf{y}_1} \leq \mathbf{0} \wedge \cdots \wedge R\binom{\mathbf{y}_k}{\mathbf{y}_{k+1}} \leq \mathbf{0}\}$$

which is exactly $\mathtt{proj}_{\mathbf{x},\mathbf{y}_0}(\widehat{\mathcal{R}}_{k+1})$, justifying the lemma's statement for $k+1$. □

LEMMA 4.32. *$\mathcal{Q}$ has a MΦRF of depth d iff $\widehat{\mathcal{R}}_d$ is empty.*

Proof. By Lemma 4.8 $\mathcal{Q}$ has a MΦRF of depth d iff $\mathcal{Q}_d = F^d(\mathcal{Q})$ is empty, and by Definition 4.26 $\mathcal{Q}_d$ is empty iff $\mathcal{R}_d$ is empty, and since $\mathcal{R}_d$ is empty iff $\widehat{\mathcal{R}}_d$ is empty the lemma follows. □

EXAMPLE 4.33. Consider again Example 4.27, in particular the SLC loop $\mathcal{Q}$ and the corresponding displacement polyhedron $\mathcal{R}$. As notation, let $\mathbf{x}_0 = (x, y, z)^{\mathrm{T}}$, $\mathbf{y}_0 = (y_1, y_2, y_3)^{\mathrm{T}}$, $\mathbf{y}_1 = (w_1, w_2, w_3)^{\mathrm{T}}$, $\mathbf{y}_2 = (z_1, z_2, z_3)^{\mathrm{T}}$, and $\mathbf{y}_3 = (v_1, v_2, v_3)^{\mathrm{T}}$. Then

$$\begin{aligned}\mathcal{R}_2 \equiv\ & \{x + z \geq 0, y_1 = y, y_2 = z, y_3 = -1\} \wedge \\ & \{y_1 + y_3 \geq 0, w_1 = y_2, w_2 = y_3, w_3 = 0\} \wedge \\ & \{w_1 + w_3 \geq 0, z_1 = w_2, z_2 = w_3, z_3 = 0\}\end{aligned}$$

is satisfiable, e.g., for $\mathbf{x}_0 = (0, 1, 0)$, $\mathbf{y}_0 = (1, 0, -1)$, $\mathbf{y}_1 = (0, -1, 0)$ and $\mathbf{y}_2 = (-1, 0, 0)$, and thus, as expected, the loop does not have a MΦRF of depth 2. On the other hand

$$\mathcal{R}_3 = \mathcal{R}_2 \wedge \{z_1 + z_3 \geq 0, v_1 = z_2, v_2 = z_3, v_3 = 0\}$$

is not satisfiable, and thus the loop has a MΦRF of depth 3. □

Apart from providing a polynomial-time implementation for the check $F^d(\mathcal{Q}) = \emptyset$, the use of the displacement polyhedra, and Lemma 4.32 in particular, has several important consequences that we discuss in the next sections.

4.3.1 Witnesses for Non-existence of MΦRFs of a Given Depth

Existing algorithms for deciding whether a given SLC loop has a MΦRF of depth d [Ben-Amram and Genaim 2017; Leike and Heizmann 2018] synthesise a MΦRF in the case of success, but in the case of failure they do not provide any further knowledge on why the loop does not have such a MΦRF. In this section we show that any satisfying assignment for $\widehat{\mathcal{R}}_k$ (as defined in (4.32)) witnesses the non-existence of a MΦRF of depth k, i.e., it can be used to explains the reason due to which the loop does not have such MΦRF.

Let us start by explaining the intuition for $\widehat{\mathcal{R}}_1$, i.e., the case of LRFs. If $\mathbf{x}_0, \mathbf{y}_0, \mathbf{y}_1$ is a satisfying assignment for $\widehat{\mathcal{R}}_1$, then by construction we have

$$\binom{\mathbf{x}_0}{\mathbf{y}_0} \in \mathcal{R} \qquad \binom{\mathbf{y}_0}{\mathbf{y}_1} \in \mathtt{rec.cone}(\mathcal{R}) \tag{4.34}$$

Note that for any $c \geq 0$, $\binom{\mathbf{x}_0}{\mathbf{y}_0} + c \cdot \binom{\mathbf{y}_0}{\mathbf{y}_1} \in \mathcal{R}$ is a transition. Assume there is a LRF $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$ for $\mathcal{R}$, then it must rank $\binom{\mathbf{x}_0}{\mathbf{y}_0}$ and $\binom{\mathbf{x}_0}{\mathbf{y}_0} + c \cdot \binom{\mathbf{y}_0}{\mathbf{y}_1} \in \mathcal{R}$ for any $c > 0$.

This means that the following must hold:

$$\vec{a} \cdot \mathbf{x}_0 + b \geq 0 \tag{4.35}$$
$$\vec{a} \cdot \mathbf{y}_0 \leq -1 \tag{4.36}$$
$$\rho(\mathbf{x}_0 + c \cdot \mathbf{y_0}) = \vec{a} \cdot \mathbf{x}_0 + c \cdot \vec{a} \cdot \mathbf{y_0} + b \geq 0 \tag{4.37}$$
$$\rho(\mathbf{y}_0 + c \cdot \mathbf{y_1}) = \vec{a} \cdot \mathbf{y}_0 + c \cdot \vec{a} \cdot \mathbf{y_1} \leq -1 \tag{4.38}$$

Clearly for c large enough (4.36) and (4.37) contradict. Thus the point $\binom{\mathbf{x}_0}{\mathbf{y}_0}$ and the ray $\binom{\mathbf{y}_0}{\mathbf{y}_1}$ form a witness that explains why the loop does not have a LRF. More precisely, the subset of the loop generated by the point and the ray of (4.34), in particular

$$\texttt{conv.hull}\{\tbinom{\mathbf{x}_0}{\mathbf{y}_0}\} + \texttt{cone}\{\tbinom{\mathbf{y}_0}{\mathbf{y}_1}\} \subseteq \mathcal{R}$$

cannot have a LRF.

Let us generalise this intuition for MΦRFs. Assume the loop has a MΦRF $\langle \rho_1, \ldots, \rho_k \rangle$, and let $\mathbf{x_0}, \mathbf{y_0}, \ldots, \mathbf{y}_k$ be an assignment satisfying $\widehat{\mathcal{R}}_k$. By construction, we have

$$\tbinom{\mathbf{x}_0}{\mathbf{y}_0} \in \mathcal{R} \qquad \tbinom{\mathbf{y}_0}{\mathbf{y}_1} \in \texttt{rec.cone}(\mathcal{R}) \quad \cdots \quad \tbinom{\mathbf{y}_{k-1}}{\mathbf{y}_k} \in \texttt{rec.cone}(\mathcal{R}) \tag{4.39}$$

We may assume that $\binom{\mathbf{x}_0}{\mathbf{y}_0}$ is ranked by ρ_1 (we can lift ρ_1 up to make $\rho_1(\mathbf{x}_0)$ non-negative).

Now let us eliminate all transitions that are ranked by $\rho_1(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + b$, i.e., compute a new displacement polyhedron $\mathcal{R}' = \mathcal{R} \wedge \rho_1(\mathbf{x}) \leq -1$. Since ρ_1 is decreasing on all transitions of $\mathcal{R}$, we must have $\vec{a} \cdot \mathbf{y}_0 \leq -1$ and $\vec{a} \cdot \mathbf{y}_i \leq 0$ for $1 \leq i \leq k$. This means that the rays $\binom{\mathbf{y}_0}{\mathbf{y}_1} \cdots \binom{\mathbf{y}_{k-1}}{\mathbf{y}_k}$ are in the $\texttt{rec.cone}(\widehat{\mathcal{R}'})$ too (because $\mathcal{R}'$ is obtained from $\mathcal{R}$ by adding $\vec{a} \cdot \mathbf{x} + b \leq -1$). Moreover, for $c > 0$ large enough, the point $\binom{\mathbf{x}_0 + c \cdot \mathbf{y}_0}{\mathbf{y}_0 + c \cdot \mathbf{y}_1}$ is in $\mathcal{R}'$ since it cannot be ranked by ρ_1 (from the same considerations of the LRF case). Now we have the following

$$\tbinom{\mathbf{x}_0 + c \cdot \mathbf{y}_0}{\mathbf{y}_0 + c \cdot \mathbf{y}_1} \in \mathcal{R}' \quad \tbinom{\mathbf{y}_0 + c \cdot \mathbf{y}_1}{\mathbf{y}_1 + c \cdot \mathbf{y}_2} \in \texttt{rec.cone}(\mathcal{R}') \quad \cdots \quad \tbinom{\mathbf{y}_{k-2} + c \cdot \mathbf{y}_{k-1}}{\mathbf{y}_{k-1} + c \cdot \mathbf{y}_k} \in \texttt{rec.cone}(\mathcal{R}')$$

It has the same form as in (4.39), i.e., the lower part of each point/ray is equal to the upper part of the next one, but the number of rays is reduced by 1, and since $\langle \rho_2, \ldots, \rho_k \rangle$ is a MΦRF for $\mathcal{R}'$ we can apply the same reasoning again and reduce the number of rays to $k - 2$. Repeating this, we arrive to a point and ray as in (4.34) that are supposed to be ranked by the last component ρ_k, but we know that they cannot have a LRF so we need at least one more component in the MΦRF. Thus, we conclude that the point and rays of (4.39) form a witness that explains why the loop cannot have a MΦRF of depth k. In fact, the subset of the loop generated by this witness, in particular

$$\texttt{conv.hull}\{\tbinom{\mathbf{x}_0}{\mathbf{y}_0}\} + \texttt{cone}\{\tbinom{\mathbf{y}_0}{\mathbf{y}_1}, \ldots, \tbinom{\mathbf{y}_{k-1}}{\mathbf{y}_k}\} \subseteq \mathcal{R}$$

cannot have a MΦRF of depth k.

EXAMPLE 4.34. The satisfying assignment for $\widehat{\mathcal{R}}_2$ in Example 4.33, is a witness for the non-existence of MΦRF of depth 2 for the SLC loop induced by the TS of Figure 2.2. The transition polyhedron corresponding to this witness is

$$\{\underline{x + z = 0, y \leq 1, z \leq 0}, x' = x + y, y' = y + z, z' = z - 1\}.$$

Note how the guard is strengthened wrt. $x + z \geq 0$ of program depicted Figure 2.2. □

4.3.2 Conditional Termination

In this section, we show how the displacement polyhedron can be used in the context of conditional termination, i.e., inferring a set of initial states for which termination of a given SLC loop is guaranteed.

Suppose that a SLC loop $\mathcal{Q}$ does not have a MΦRF of depth d, which means that $\widehat{\mathcal{R}}_d$ is satisfiable. Now let us consider the polyhedron $\mathcal{P} = \mathtt{proj}_{\mathbf{x}}(\widehat{\mathcal{R}}_d)$, and let the inequality $\vec{a}_i \cdot \mathbf{x} \leq b_i$ be part of the constraints representation of $\mathcal{P}$ (or more generally, we can use any $(\vec{a}_i, b_i) \in \mathcal{P}^{\#}$). Observe that adding the constraint $\vec{a}_i \cdot \mathbf{x} \geq b_i + \epsilon$ to $\mathcal{Q}$, for any $\epsilon > 0$, we get a SLC loop that has a MΦRF of depth at most d, since the corresponding $\widehat{\mathcal{R}}_d$ is empty then. We can use this fact to infer initial states for which $\mathcal{Q}$ is guaranteed to terminate as follows.

Let $I_i \subseteq \mathbb{Q}^n$ be an over-approximation of the set of all initial state that might reach $\vec{a}_i \cdot \mathbf{x} \leq b_i$, which can be computed using standard abstract interpretation techniques. Clearly, starting the execution from $\neg I_i$, i.e., the set of states not in I_i, the loop terminates. This is because such executions will use transitions from $\mathcal{Q} \wedge \vec{a}_i \cdot \mathbf{x} > b_i$ only, and as we have seen above that such executions are terminating since they have a MΦRF. We can do this for each inequality of the constrains representation of $\mathcal{P}^{\#}$ and then take $\cup_i \neg I_i$ as a set of initial states that are guaranteed to terminate.

EXAMPLE 4.35. Consider the following loop

$$\texttt{while } (x + z \geq 0) \texttt{ do } x' = x + y,\ y' = y + z,\ z' = z \tag{4.40}$$

which is a non-terminating variation of the one in Figure 2.2. The transition polyhedron of this loop is $\mathcal{Q} = \{x + z \geq 0, x' = x + y, y' = y + z, z' = z\}$, and the corresponding displacement polyhedron is $\mathcal{R} = \{x + z \geq 0, y_1 = y, y_2 = z, y_3 = 0\}$.

Let us consider $\widehat{\mathcal{R}}_2$ which is defined by

$$\begin{aligned}\widehat{\mathcal{R}}_2 = \{&x + z \geq 0, y_1 \geq 0, w_1 \geq 0, y_1 = y, y_2 = z, y_2 = w_1,\\ &y_3 = 0, w_2 = 0, w_3 = 0, z_1 = 0, z_2 = 0, z_3 = 0\}\end{aligned}$$

Projecting $\widehat{\mathcal{R}}_2$ on the variables $\mathbf{x} = (x, y, z)^{\mathrm{T}}$ results in

$$\mathcal{P} = \mathtt{proj}_{\mathbf{x}}(\widehat{\mathcal{R}}_2) = \{x + z \geq 0, y \geq 0, z \geq 0\}.$$

Next we compute the preconditions induced by each constraint of $\mathcal{P}$:

- For $x + z \geq 0$, the set of initial states $I_0 = \{x + z \geq 0\}$ is an over-approximation of those that might reach $x + z \geq 0$. Thus, executions that start from states in $\neg I_0 = \{x + z < 0\}$ are terminating.
- For $y \geq 0$, the set of initial states $I_1 = \{y \geq 0\} \vee \{z \geq 0\}$ is an over-approximation of those that might reach $y \geq 0$. Thus, executions that start from states in $\neg I_1 = \{y < 0, z < 0\}$ are terminating.
- For $z \geq 0$, the set of initial states $I_2 = \{z \geq 0\}$ is an over-approximation of those that might reach $z \geq 0$. Thus, executions that start from states in $\neg I_2 = \{z < 0\}$ are terminating.

We conclude that executions that starts from initial states in

$$I_0 \vee I_1 \vee I_2 = x + z < 0 \vee (y < 0 \wedge z < 0) \vee z < 0$$

are guaranteed to terminate. □

4.3.3 Termination and Non-termination of Bounded SLC Loops

To further demonstrate the usefulness of the displacement polyhedra, in this section we provide some observations, regarding SLC loops whose set of enabled states are defined by bounded polyhedra, that are easy to see using the displacement polyhedron and are much less obvious using the transition polyhedron. Recall that a polyhedron is bounded if its recession cone consists of a single point $\mathbf{0}$.

LEMMA 4.36. *Let $\mathcal{Q}$ be a SLC loop such that the set of enabled states $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a bounded polyhedron, then $\mathcal{Q}$ is non-terminating iff it has a fixpoint $\binom{\mathbf{x}}{\mathbf{x}} \in \mathcal{Q}$, and it is terminating iff it has a LRF.*

Proof. Let $\mathcal{R}$ be the displacement polyhedron of $\mathcal{Q}$. Since $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ is bounded, $\mathtt{proj}_{\mathbf{x}}(\mathcal{R})$ is bounded. This means that its recession cone $R\binom{\mathbf{x}}{\mathbf{y}} \leq \mathbf{0}$ consists of points of the form $\binom{\mathbf{0}}{\mathbf{y}}$. From the form of $\widehat{\mathcal{R}}_k$, which is a conjunction of instances of $R\binom{\mathbf{y}_i}{\mathbf{y}_{i+1}} \leq \mathbf{0}$, it is easy to see that $\mathcal{R}_2 = \mathcal{R}_1$. This means that the algorithm will terminate in at most two iterations with one of the following outcomes: (*i*) $\mathcal{R}_0 = \mathcal{R}_1$; (*ii*) $\mathcal{R}_2 = \mathcal{R}_1$; or (*iii*) $\mathcal{R}_1$ is empty. In the first two cases all transitions of $\mathcal{R}_1$ or $\mathcal{R}_2$ are of the form $\binom{\mathbf{x}}{\mathbf{0}}$, and thus $\binom{\mathbf{x}}{\mathbf{x}} \in \mathcal{Q}$ by definition; and in the third case we have found a MΦRF of depth 1, i.e., LRF. Note that the part that relates non-termination to the existence of a fixpoint follows from the work of Leike and Heizmann [2018] as well. □

4.3.4 New Directions for the General MΦRF Problem

We believe that the displacement polyhedra representation, in particular the check induced by Lemma 4.31, provides us with new tools that can be used for addressing the problem of deciding whether a given SLC loop $\mathcal{Q} \equiv [A''\mathbf{x}'' \leq \mathbf{c}'']$ has a MΦRF of any depth, which is still an open problem. Next we discuss some directions.

One direction is to come up with conditions on the matrices A'' (or equivalently R of the corresponding displacement representation) and $\mathbf{c}''$ under which it is guaranteed that if $\widehat{\mathcal{R}}_k$ is empty then k must be smaller than some d, i.e., bounding the depth of MΦRFs for classes of loops that satisfy these conditions.

We can also view the problem as looking for some N, such that $\mathcal{C}^N = \mathcal{C}^{N+1}$ where $\mathcal{C} \equiv [R\binom{\mathbf{y}}{\mathbf{y}'} \leq \mathbf{0}]$, which is a sufficient condition for Algorithm 3 to terminate in at most N iterations, since then $\mathcal{R}_N = \mathcal{R}_{N+1}$, either with a recurrent set or with a MΦRF. This is particularly interesting if the loop is deterministic with an affine update $\mathbf{x}' = U\mathbf{x} + \mathbf{c}$. In such case $\mathcal{C} = [B\mathbf{y} \leq 0 \wedge \mathbf{y}' = (U - I)\mathbf{y}]$, where $I \in \mathbb{Q}^{n \times n}$ is the identity matrix, and thus if the matrix $(U - I)$ is nilpotent, for example, then there is N such that $\mathcal{C}^N = \mathcal{C}^{N+1}$. This, for example, also holds when matrix $(U - I)$ satisfies the finite-monoid property that is used by Iosif et al. [2014].

Another tantalising observation reduces the existence of d such that $\widehat{\mathcal{R}}_d$ is empty to the question whether a related SLC loop terminates, for a given polyhedron of initial states, in a bounded number of steps. Specifically, the loop:

$$\mathtt{while}\ (B\mathbf{y} \leq \mathbf{0})\ \mathtt{do}\ (A + A')\mathbf{y} + A'\mathbf{y}' \leq \mathbf{0}.$$

where B, A and A' are those used in the definition of R in (4.31), and the question whether it terminates in at most d steps for all $\mathbf{y} \in \{\mathbf{y} \in \mathbb{Q}^n \mid R\binom{\mathbf{x}}{\mathbf{y}} \leq \mathbf{c}''\}$. This is because $\widehat{\mathcal{R}}_d$ as in (4.32) is equivalent to unrolling the above loop d times. If the update

is affine, i.e., $\mathbf{x}' = U\mathbf{x} + \mathbf{c}$, then the above loop is equivalent to

$$\texttt{while } (B\mathbf{y} \leq \mathbf{0}) \texttt{ do } \mathbf{y}' = (U - I)\mathbf{y}$$

where $I \in \mathbb{Q}^{n \times n}$ is the identity matrix.

4.4 Loops for Which MΦRFs are Sufficient

The purpose of this section is to demonstrate the usefulness of Algorithm 3 for studying properties of SLC loops. In particular, we use it to characterise kinds of SLC loops for which there is always a MΦRF, if the loop is terminating. We shall prove this result for two kinds of loops, both considered in previous work, namely *octagonal relations* and *affine relations with the finite-monoid property* – for both classes, termination has been proven decidable by Iosif et al. [2014]. We only consider the rational case. Another question, which we do not answer, is whether we can ensure that Algorithm 3 recognises the non-terminating members of the class.

Let us set some notation first. The composition of transition relations $S, T \subseteq \mathbb{Q}^{2n}$ is defined as $S \circ T = \{\binom{\mathbf{x}}{\mathbf{z}} \mid \exists \mathbf{y} \,.\, \binom{\mathbf{x}}{\mathbf{y}} \in S \wedge \binom{\mathbf{y}}{\mathbf{z}} \in T\}$. We let $T^n = T^{n-1} \circ T$ where T^0 is the identity relation. We use $\texttt{pre}^n(T)$ for the projection of T^n over $\mathbf{x}$, i.e., $\texttt{pre}^n(T) = \texttt{proj}_{\mathbf{x}}(T^n)$, which is the set of states from which we can make traces of length at least n (for non-deterministic loops some might be less than n as well). When T is polyhedral, i.e., a SLC loop, then T^n and $\texttt{pre}^n(T)$ are polyhedral as well.

4.4.1 Finite Loops

First, we consider loops which always terminate and, moreover, their number of iterations is bounded by a constant, i.e., there is $N > 0$ such that $\mathcal{Q}^N = \emptyset$. Note that such loop terminates in at most $N - 1$ iterations, or equivalently $N - 1$ is an upper-bound on the length of the corresponding traces.

LEMMA 4.37. *If $\mathcal{Q}^N = \emptyset$, then it has a MΦRF of depth less than N.*

Proof. The proof is by induction on N. For $N = 1$, $\mathcal{Q} = \emptyset$, and it has a MΦRF of zero depth, by definition. Let $N > 1$, and assume that $\mathcal{Q}^{N-1} \neq \emptyset$, otherwise it trivially follows for N. Consider a transition $\mathbf{x}'' = \binom{\mathbf{x}}{\mathbf{x}'}$ that is the last in a terminating trace. We have $\mathbf{x} \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})$ and $\mathbf{x}' \notin \texttt{proj}_{\mathbf{x}}(\mathcal{Q})$. Since $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a closed polyhedral set, this means that there is a function ρ, defined by some $(\vec{a}, b) \in \texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$, that is non-negative over $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})$ but negative on $\mathbf{x}'$, and thus $\Delta\rho(\mathbf{x}'') = \rho(\mathbf{x}) - \rho(\mathbf{x}') > 0$. It follows that $\mathbf{x}''$ is eliminated by Algorithm 3 when computing $\mathcal{Q}'$ at Line 4. This means that any transition of $\mathcal{Q}'$ cannot be the last transition of any terminating run of $\mathcal{Q}$, and thus $(\mathcal{Q}')^{N-1} = \emptyset$. Therefore, by induction, it has a MΦRF of depth at most $N - 2$, and by Lemma 4.7 $\mathcal{Q}$ has a MΦRF of depth at most $N - 1$. □

4.4.2 The Class $\texttt{RF}(b)$

This class contains loops which can be described as having the following behaviour: Transitions are linearly ranked, as long as we are in states from which we can make runs of length at least b. In other words, once we reach a state from which we cannot make more than $b - 1$ transitions we do not require the rest of the trace to be linearly ranked.

Definition 4.38. We say that a SLC loop $\mathcal{Q}$ belongs to the class $\mathtt{RF}(b)$ if the loop $\mathcal{Q} \cap \{(\mathbf{x}, \mathbf{x}') \in \mathbb{Q}^{2n} \mid \mathbf{x} \in \mathtt{pre}^b(\mathcal{Q})\}$ has a LRF.

We note that that $\mathtt{RF}(1)$ is the class of loops which have a LRF.

LEMMA 4.39. *Loops in* $\mathtt{RF}(b)$ *have MΦRFs of depth at most* b.

Proof. This lemma actually generalises Lemma 4.37, since the loops concerned there are $\mathtt{RF}(N-1)$. The proof is done similarly by induction on b. For $b = 1$, $\mathcal{Q}$ has a LRF by definition. Let $b > 1$, and suppose that $\mathbf{x}'' = \binom{\mathbf{x}}{\mathbf{x}'} \in \mathcal{Q}$ is a last transition of a terminating run, then $\mathbf{x} \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ and $\mathbf{x}' \notin \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$. Since $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a closed polyhedral set, this means that there is a function ρ, defined by some $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$, that is non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ but negative over $\mathbf{x}'$, and thus $\Delta\rho(\mathbf{x}'') = \rho(\mathbf{x}) - \rho(\mathbf{x}') > 0$. It follows that $\mathbf{x}''$ is eliminated by Algorithm 3 when computing $\mathcal{Q}'$ at Line 4. This means that any transition of $\mathcal{Q}'$ cannot be the last transition of any terminating run of $\mathcal{Q}$, and thus $\mathcal{Q}'$ is $\mathtt{RF}(b-1)$. Therefore, by induction, it has a MΦRF of depth at most $b-1$, and by Lemma 4.7 $\mathcal{Q}$ has a MΦRF of depth at most b. □

EXAMPLE 4.40. Consider the loop [Iosif et al. 2014] defined by

$$\mathcal{Q} = \{x_2 - x_1' \leq -1, x_3 - x_2' \leq 0, x_1 - x_3' \leq 0, x_4' - x_4 \leq 0, x_3' - x_4 \leq 0\}.$$

This loop is $\mathtt{RF}(3)$, since adding $\mathtt{pre}^3(Q) = \{x_2 + x_4 \geq 1, x_3 + x_4 \geq 1, x_1 + x_4 \geq 0\}$ to the loop we find a LRF, e.g., $\rho(\mathbf{x}) = -x_1 - x_2 - x_3 + 3x_4 + 1$. Indeed, $\mathcal{Q}$ has a MΦRF of optimal depth 3, e.g., $\langle -x_1 - x_2 - x_3 + 3x_4 + 1, -\frac{2}{3}x_1 - \frac{1}{3}x_2 + x_4 + 1, -\frac{1}{4}x_1 + \frac{1}{4}x_4 + 1\rangle$. Note that the first component is the LRF that we have just found for $\mathcal{Q} \cap \mathtt{pre}^3(Q)$. □

If we know that a given class of loop belongs to $\mathtt{RF}(b)$, then bounding the recursion depth of Algorithm 3 by b gives us a decision procedure for the existence of MΦRF for this class. Iosif et al. [2014] proved that *octagonal relations* are $\mathtt{RF}(5^{2n})$, where n is the number of variables[1]. Thus for octagonal relations, we can decide termination and for terminating loops obtain MΦRFs. For the depth of the MΦRF, namely the parameter b above, Iosif et al. [2014] gives a tighter (polynomial) result for those octagonal relations which allow arbitrarily long executions (called $*$-consistent).

4.4.3 Loops with Affine-linear Updates

In certain cases, we can handle loops with affine-linear updates – which are, in general, not octagonal. Recall that a loop with affine-linear update has a transition relation of the form:

$$\mathcal{Q} \equiv [B\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x}' = U\mathbf{x} + \mathbf{c}]\,. \tag{4.41}$$

We keep the meaning of the symbols $U, B, \mathbf{b}, \mathbf{c}$ fixed for the sequel. Moreover, we express the loop using the transformation $\mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{c}$ and the guard $\mathcal{G} \equiv [B\mathbf{x} \leq \mathbf{b}]$. We use U_{ij} to denote the entry of matrix U in row i and column j, and for a vector $\mathbf{v}$ we let $\mathbf{v}[i..j]$ be the vector obtained from components i to j of the vector $\mathbf{v}$.

Our goal is to show that if U^p, for some $p > 0$, is diagonalizable and all its eigenvalues are in $\{0, 1\}$, then $\mathcal{Q}$ is $\mathtt{RF}(3p)$, and thus, by Lemma 4.39, if terminating, it has a MΦRF. Affine loops with the *finite monoid property* that has been addressed by Iosif et al. [2014], satisfy this condition (interestingly, in Section 4.3.4 we have shown a similar result when $U - I$ has the *finite monoid property*). We state some auxiliary lemmas first.

[1] Technically, they prove it just for *integer* loops, but the result applies to the rational case as well (one only has to *simplify* some considerations away from the proof).

LEMMA 4.41. *Let $\mathcal{Q}$ be an affine-linear loop as in (4.41) such that, for some $N > 0$, $\mathcal{Q}^N$ is $\mathtt{RF}(b)$. Then $\mathcal{Q}$ is $\mathtt{RF}(N(b+1))$.*

Proof. If $\mathcal{Q}^N$ is $\mathtt{RF}(b)$, then $\mathcal{Q}^N \cap \{\mathbf{x}'' \mid \mathbf{x} \in \mathtt{pre}^b(\mathcal{Q}^N)\}$ has a LRF ρ, and thus

$$\mathbf{x} \in \mathtt{pre}^b(\mathcal{Q}^N) = \mathtt{pre}^{Nb}(\mathcal{Q}) \;\Rightarrow\; \rho(\mathbf{x}) \geq 0 \wedge \rho(\mathbf{x}) - \rho(\mathcal{U}^N(\mathbf{x})) > 0\,. \tag{4.42}$$

Note that $\rho(\mathbf{x}) - \rho(\mathcal{U}^N(\mathbf{x}))$ can be written as

$$\sum_{j=0}^{N-1} \rho(\mathcal{U}^j(\mathbf{x})) - \sum_{j=0}^{N-1} \rho(\mathcal{U}^{j+1}(\mathbf{x})) \tag{4.43}$$

This is because every term $\rho(\mathcal{U}^i(\mathbf{x}))$, except for $i = 0$ and $i = N$, appear in (4.43) with positive and negative signs. Hence, if we let $\rho_1(\mathbf{x}) = \sum_{j=0}^{N-1} \rho(\mathcal{U}^j(\mathbf{x}))$ then:

$$\mathbf{x} \in \mathtt{pre}^{Nb}(\mathcal{Q}) \;\Rightarrow\; \rho_1(\mathbf{x}) - \rho_1(\mathcal{U}(\mathbf{x})) > 0\,.$$

Moreover, ρ_1 is the sum of terms $\rho(\mathcal{U}^i(\mathbf{x}))$ which are bounded from below on $\mathtt{pre}^{N(b+1)}(\mathcal{Q})$. Hence, we have a LRF for $\mathcal{Q} \cap \{\mathbf{x}'' \mid \mathbf{x} \in \mathtt{pre}^{N(b+1)}(\mathcal{Q})\}$ and thus $\mathcal{Q}$ is $\mathtt{RF}(N(b+1))$. □

LEMMA 4.42. *Let $\mathcal{Q}$ be a loop as in (4.41), and assume U is diagonal with entries in $\{0, 1\}$. Then, if $\mathcal{Q}$ is terminating, it is $\mathtt{RF}(2)$.*

Proof. Without loss of generality we may assume that $U_{11} = \cdots = U_{kk} = 1$ and $U_{jj} = 0$ for $j > k$, otherwise we could reorder the variables to put it into this form. Clearly, the update adds $\mathbf{c}_1 = \mathbf{c}[1..k]$ to the first k elements of $\mathbf{x}$, and sets the rest to $\mathbf{c}_2 = \mathbf{c}[k+1..n]$. Consequently, such a loop is non-terminating iff the space $V = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{x}[k+1..n] = \mathbf{c}_2\}$ intersects the loop guard $\mathcal{G} \equiv [B\mathbf{x} \leq \mathbf{b}]$, and the vector $\mathbf{u} = (c_1, \ldots c_k, 0, \ldots, 0)^{\mathrm{T}}$ is a recession direction of the guard, i.e., $B\mathbf{u} \leq \mathbf{0}$. To see this: suppose these conditions hold, then starting from any state $\mathbf{x}_0 \in V$, the state after i iterations will be $\mathbf{x}_i = \mathbf{x}_0 + i\mathbf{u}$, which is in $\mathcal{G}$ since $\mathbf{x}_0 \in \mathcal{G}$ and $\mathbf{u} \in \mathtt{rec.cone}(\mathcal{G})$, and thus the execution does not terminate; for the other direction, suppose it does not terminate, then there must be a non-terminating execution that starts in $\mathbf{x}_0 \in V$, this execution generates the states $\mathbf{x}_0 + i\mathbf{u} \in \mathcal{G}$ and thus $\mathbf{u}$ is a recession directions of $\mathcal{G}$.

Now suppose the loop is terminating, we show that it is $\mathtt{RF}(2)$. Let us analyse a run of the loop starting with some valid transition $\left(\begin{smallmatrix}\mathbf{x}_0\\ \mathbf{x}_1\end{smallmatrix}\right)$. We have two cases:

1. If $\mathbf{x}_1 \notin \mathcal{G}$, then the run terminates in 1 iteration.
2. If $\mathbf{x}_1 \in \mathcal{G}$, then V intersects with $\mathcal{G}$, since $\mathbf{x}_1[k+1..n] = \mathbf{c}_2$, and thus $B\mathbf{u} \leq \mathbf{0}$ should not hold, otherwise the loop is non-terminating. This means that there is a constraint $\vec{b}_i \cdot \mathbf{x} \leq b_i$ of the guard such that $\vec{b}_i \cdot \mathbf{u} > 0$. Let $\vec{a}$ be as $\vec{b}_i$ but setting components $k+1..n$ to zero, we still have $\vec{a} \cdot \mathbf{u} > 0$ because these components are 0 in $\mathbf{u}$. We show that this trace is linearly ranked by $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + \max(b_i, 0)$. Suppose the initial state is $\mathbf{x}_0$, and write it as $\left(\begin{smallmatrix}\mathbf{x}_0[1..k]\\ \mathbf{x}_0[k+1..n]\end{smallmatrix}\right)$. Consider a trace that starts in $\mathbf{x}_0$, it is easy to see that the ith state, for $i \geq 1$, is $\mathbf{x}_i = \left(\begin{smallmatrix}\mathbf{x}_0[1..k] + i\mathbf{c_1}\\ \mathbf{c}_2\end{smallmatrix}\right)$. Then, we have $\rho(\mathbf{x}_i) - \rho(\mathbf{x}_{i+1}) = \vec{a} \cdot \mathbf{u} > 0$, moreover ρ is non-negative on all state except the last of the trace (which is not in the guard).

This analysis implies that any terminating trace is either of length 1, or has a LRF $\rho(\mathbf{x}) = \vec{a} \cdot \mathbf{x} + \max(b_i, 0)$, and together with the fact that the loop is deterministic we conclude that it is $\mathtt{RF}(2)$. □

Now we are in a position for proving our main result of this section.

LEMMA 4.43. *If U^p, for some $p > 0$, is diagonalizable and all its eigenvalues are in $\{0, 1\}$, then loop (4.41) is either non-terminating or $\mathtt{RF}(3p)$.*

Proof. Recall that the update is $\mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{c}$, then $\mathcal{U}^p(\mathbf{x}) = U^p\mathbf{x} + \mathbf{v}$, for a vector $\mathbf{v} = (I + U + \cdots + U^{p-1})\mathbf{c}$. Taking into account the guard,

$$\mathcal{Q}^p \equiv (B\mathbf{x} \leq \mathbf{b} \wedge \cdots \wedge B\mathcal{U}^{p-1}(\mathbf{x}) \leq \mathbf{b}) \wedge \mathbf{x}' = \mathcal{U}^p(\mathbf{x})\,. \tag{4.44}$$

We write this guard concisely with the notation $B^{\langle p\rangle}\mathbf{x} \leq \mathbf{b}^{\langle p\rangle}$. Since, by assumption, U^p is diagonalizable, there is a non-singular matrix P and a diagonal matrix D such that $P^{-1}U^pP = D$ and D has only 1's and 0's on the diagonal (P is a change-of-basis transformation). We consider a loop $\widehat{\mathcal{Q}^p}$ which is similar to $\mathcal{Q}^p$ but transformed by P, that is:

$$\widehat{\mathcal{Q}^p} \equiv BP^{\langle p\rangle}\mathbf{x} \leq \mathbf{b}^{\langle p\rangle} \wedge \mathbf{x}' = D\mathbf{x} + P^{-1}\mathbf{v}\,. \tag{4.45}$$

Properties like termination and linear ranking are not affected by such a change of basis.

This is because if $\binom{\mathbf{x}_0}{\mathbf{x}_1}$ is a transition of $\mathcal{Q}^p$ then $\binom{P^{-1}\mathbf{x}_0}{P^{-1}\mathbf{x}_1}$ is a transition of $\widehat{\mathcal{Q}^p}$, and if $\binom{\mathbf{x}_0}{\mathbf{x}_1}$ is a transition of $\widehat{\mathcal{Q}^p}$ then $\binom{P\mathbf{x}_0}{P\mathbf{x}_1}$ is a transition of $\mathcal{Q}^p$. This means that there is a one-to-one correspondence between the traces. Moreover, if function $\vec{a}\cdot\mathbf{x} + b$ ranks a transition of $\mathcal{Q}^p$ then $(\vec{a}P^{-1})\cdot\mathbf{x} + b$ ranks the corresponding transition of $\widehat{\mathcal{Q}^p}$, and if it ranks a transition $\widehat{\mathcal{Q}^p}$ then $(\vec{a}P)\cdot\mathbf{x} + b$ ranks the corresponding transition of $\mathcal{Q}^p$. We conclude that, if terminating, $\mathcal{Q}^p$ is $\mathtt{RF}(b)$ iff $\widehat{\mathcal{Q}^p}$ is $\mathtt{RF}(b)$.

Now, $\widehat{\mathcal{Q}^p}$ has the diagonal form discussed in Lemma 4.42, and thus, in the case that it terminates, it is $\mathtt{RF}(2)$ and so is $\mathcal{Q}^p$. Then using Lemma 4.41 we conclude that $\mathcal{Q}$ is indeed $\mathtt{RF}(3p)$. □

Chapter 5

Implementation

This chapter describes iRankFinder, a termination analyser that includes the algorithms and techniques discussed in chapters 3 and 4, as well as state-of-the-art techniques discussed in Section 2.3. Besides, it describes a toolkit for the rapid development of web-interfaces for research prototype tools, which has been have developed in the context of this thesis and used to build a web-interfaces for iRankFinder and other tools.

5.1 iRankFinder

iRankFinder is an open-source termination and non-termination analyser for TSs that implements, among other things, the techniques described in chapters 3 and 4. In addition, it includes components for invariant generation, assertion checking, and control-flow refinement that can be used independently from termination analysis. It is written in Python 3 and uses the *Parma Polyhedra Library* (PPL) [Bagnara et al. 2008] for manipulating polyhedra and Z3 [de Moura and Bjørner 2008] for solving constraints. It can be used via a web-interface or from a command-line.

5.1.1 Input Syntax

There are several TS syntax used by different termination analysis tools, and there are sets of benchmarks that are available in one syntax but not in others. iRankFinder implements a new TS syntax that we call `flow-chart` and is described below, however, it also supports two widely used TS syntax:

- `smt2`: this syntax is used in the termination competition and was formally defined by Brockschmidt and Rybalchenko. We have modified the tool `SMTPushDown`[1], that includes a parser for this syntax, to output the TS as a `flow-chart`. Files in this syntax are identified by iRankFinder with the extension "`.smt2`".

- `koat-kittel`: this syntax was first introduced by Falke et al. [2011] for the `KITTeL` termination analyser, and then used in some variation by Brockschmidt et al. [2016b] for the complexity analysis tool `KoAT`. It is the standard format used in the complexity analysis competition as well. Files in this syntax are identified by iRankFinder with the extension "`.koat`".

[1] https://github.com/jesusjda/SMTPushdown/

```
{
  vars: [x, y, z],
  pvars: [x', y', z'],
  initnode: n0,
  nodes: {
    n1: { cfr_properties: [[x >= 1],[y =< z - 1],[y >= z]] },
    n3: { asserts: [[[x <= 0]]] }
  },
  transitions: [
    {
      source: n0,
      target: n1,
      name: Q0,
      constraints: [x' = x, y' = y, z' = z]
    },
    {
      source: n1,
      target: n2,
      name: Q1,
      constraints: [x >= 1, x' = x, y' = y, z' = z]
    },
    {
      source: n1,
      target: n3,
      name: Q2,
      constraints: [x <= 0, x' = x, y' = y, z' = z]
    },
    {
      source: n2,
      target: n1,
      name: Q3,
      constraints: [y <= z - 1, x' = x+1, y' = y + 1, z' = z]
    },
    {
      source: n2,
      target: n1,
      name: Q4,
      constraints: [y >= z, x' = x - 1, y' = y, z' = z]
    }
   ]
}
```

Figure 5.1: An example of TS in `flow-chart` syntax. It corresponds to TS $\mathcal{T}$ of Figure 3.2.

The parser of iRankFinder directly parses the syntax `koat-kittel` and `flow-chart` and converts them to an internal representation. iRankFinder accepts basic `C` programs as well, and translates them to `koat-kittel` syntax using tool `LLVM2KTTEL` that was also developed in the context of the `KITTeL` termination analyser of Falke et al. [2011].

Our `flow-chart` syntax is based on JSON structures, and thus it is easy to extend. Apart from the definition a TS, it also allows passing other information to the analyser such as user-defined properties to be used for CFR, assertions to be checked, etc. An example of a TS in this syntax is depicted in Figure 5.1, it corresponds to the TS $\mathcal{T}$ of Figure 3.2. Next, we describe the details of the `flow-chart` syntax using this example.

A TS in `flow-chart` syntax is a JSON-like dictionary where some keys are mandatory and some are optional (the mandatory keys are marked with $\star$):

- `vars`*: a list of tokens representing a sequence of global variable names (V in Definition 2.4).
- `pvars`: a list of tokens representing a sequence of primed variable names (V' in Definition 2.4). The ith element of `pvars` is the primed version of the ith element in `vars`. The default value is generated by adding a quote to the values in `vars`.

- `domain`: a token `Q` or `Z` representing the domain of variables – rationals or integers. The default value is `Z`.

- `initnode`: a token representing the identifier of the initial node of the TS. The default value is the source node n_s of the first edge, and if n_s has incoming edges then an auxiliary node n_0 with a single edge to n_s is added to the TS.

- `transitions`*: a list of dictionaries (with specific keys) representing edges of the TS. The essential keys are:

  - `source`*: a token representing the identifier of source node.
  - `target`*: a token representing the identifier target node.
  - `name`: an identifier for the edge.
  - `constraints`*: a list of linear constraints representing the corresponding transitions polyhedron. The constraints are over variables from `vars` and `pvars`, other variables are treated as existentially quantified (i.e., local to the edge). The parser allows non-linear constraints as well, but they are ignored by iRankFinder (this should be used carefully since while it is safe for termination, it is not safe for non-termination).

  An edge can be disabled by adding the key `ignore` with the value **true**.

- `nodes`: a dictionary where keys are node identifiers, and the value of each key is another dictionary that provides extra information for the corresponding node:

  - `asserts`: a list of assertions that must hold when the execution reaches the corresponding node. iRankFinder can check assertions if the corresponding option is enabled.
  - `cfr_properties`: constraints representing user-defined properties to be used for CFR.
  - `threshold`: constraints representing thresholds [Lakhdar-Chaouch et al. 2011] to be used in the widening operation of the invariants generator.

Files in this syntax are identified by iRankFinder with the extension "`.fc`".

5.1.2 Invariant Generation

iRankFinder includes an invariant generation component that implements a standard fix-point algorithm with two abstract domains: polyhedra [Cousot and Halbwachs 1978] and intervals [Cousot and Cousot 1977]. For polyhedra, it allows using the threshold-based techniques for widening [Lakhdar-Chaouch et al. 2011]. Invariants can be inferred at different stages: before starting the termination analysis, before applying CFR, after applying CFR, etc. Invariants are used to check user assertions as well, if required by the user, by checking that each assertion is implied by the corresponding invariant. The invariant generation component can be applied independently from termination analysis, and invariants can be written to the input TS as annotations for corresponding nodes.

5.1.3 Control-Flow Refinement

The CFR techniques of Chapter 3 are implemented in iRankFinder, including the three different schemes. For partial evaluation of CHCs, i.e., for procedure `PE` of Section 3.1, it uses a slightly modified version of the tool developed by Gallagher [2019]. It also allows applying `PE` iteratively on its output, which might achieve further refinements, but also risks increasing the size of the resulting TS (there are examples where this is needed). iRankFinder translates TSs into CHC programs as accepted by the tool of Gallagher [2019], and also translates the resulting specialised CHC programs to TSs. For properties, it allows using any subset of those defined in Section 3.1.4, as well as user-defined properties.

The CFR component can be used independently from termination analysis as well. It accepts TSs in any of the syntax mentioned above and writes the specialised TSs into a file using the selected syntax. Indeed, it is already used as a black-box in other tools as we report in Chapter 6.

5.1.4 Termination and Non-termination Analysis

The core algorithm used for proving termination and non-termination is based Algorithm 2 which incorporates the CFR procedure (see Section 3.2).

Non-termination Analysis

For non-termination analysis, it implements the technique described in Section 4.2.3 to seek recurrent sets, and another simpler technique that checks (using SMT queries) if a closed walk can start and end in the same state (i.e., has a fix-point). For reachability, it collects all constraints on a path from the initial node to a node of the recurrent set and asks an SMT solver to find a solution for these constraints together with those of the recurrent set. This is done in a loop that enumerates such paths and has a bound on the number of times an edge can be taken (to guarantee termination of the loop). Since iRankFinder allows local variables in transition polyhedra, they might be instantiated to concrete values to make the corresponding closed walk deterministic – this is important for TSs with integer variables, see section 4.2.2 and 4.2.3.

Termination Analysis

For termination analysis, it implements an algorithm for LRFs [Podelski and Rybalchenko 2004], and several algorithms corresponding to the QLRFs of Alias et al. [2010], Bradley et al. [2005a] and Ben-Amram and Genaim [2014] (see Section 2.3.1). It also allows the user to specify if the (quasi-)ranking functions should use the same linear function for all nodes, or to use possibly different one for each node (functions can be different only wrt. free constant, or wrt. coefficients as well). As we have seen in Example 2.10 this might affect precision, but also performance.

iRankFinder implements as well a new kind of a quasi-ranking function that generalises the notion of *poly-ranking* of Bradley et al. [2005c]. It uses tuples of d linear functions instead of only linear functions as done for QLRFs (it is a kind of *quasi-lexicographic linear ranking function*). For a fixed d, it assigns to each node $\mathtt{n_i}$ a tuple $\langle \rho_{1,\mathtt{n_i}}, \ldots, \rho_{d,\mathtt{n_i}} \rangle$

such that: (i) there is at least one edge $\mathtt{n_s} \xrightarrow{\mathcal{Q}} \mathtt{n_t} \in E_S$ for which any $\mathbf{x}'' \in \mathcal{Q}$ satisfies

$$\rho_{1,\mathtt{n_s}}(\mathbf{x}) - \rho_{1,\mathtt{n_t}}(\mathbf{x}') \geq 1 \tag{5.1}$$

$$\forall 1 < i \leq d \;.\; \rho_{i,\mathtt{n_s}}(\mathbf{x}) - \rho_{i,\mathtt{n_t}}(\mathbf{x}') + \rho_{i-1,\mathtt{n_s}}(\mathbf{x}) \geq 1 \tag{5.2}$$

$$\rho_{d,\mathtt{n_s}}(\mathbf{x}) \geq 0 \tag{5.3}$$

and (ii) any other other $\mathbf{x}''$ (that comes from other edges) satisfies

$$\forall 1 \leq i \leq d \;.\; \rho_{i,\mathtt{n_s}}(\mathbf{x}) - \rho_{i,\mathtt{n_t}}(\mathbf{x}') \geq 0 \tag{5.4}$$

Conditions (5.1-5.3) specify the decreasing transitions, while Condition (5.4) specifies the non-increasing ones. Conditions (5.1-5.3) are the same required for *nested ranking functions* [Ben-Amram and Genaim 2017; Leike and Heizmann 2015] which is a special case of MΦRF. Thus, we will refer to this new quasi-ranking function as *nested* QLRF. Intuitively, it finds MΦRFs that eliminate at least one edge in each iteration. The resulting lexicographic ranking function has tuples in its components.

It is easy to check that for $d = 1$ we get the QLRF definition of Bradley et al. [2005a]. For $d > 1$ iRankFinder can handle some examples that are handled by the QLRFs of Larraz et al. [2013], for example, we succeed to prove termination of the TS of Example 2.17 but fail on the one of Example 2.16.

5.1.5 Handling Strict Inequalities

Throughout the thesis, we used only non-strict inequalities. However, iRankFinder allows using *strict* inequities of the following form when defining transition polyhedra:

$$\sum_{i=1}^{n} a_i x_i < b$$

If the domain of variables is $\mathbb{Z}$, it turns each $<$ to $\leq$ and adds -1 to the right-hand side (assuming all a_i and b are integer, otherwise it multiplies the inequality by a large positive integer first to make them integer). If the domain of variables is $\mathbb{Q}$, it turns each $<$ to $\leq$ as well, which can be seen as replacing an *open polyhedron* $\mathcal{Q}$ by its topological closure. However, while this is clearly sound for termination it is not sound for non-termination. Interestingly, this transformation even preserves completeness for LRFs and some QLRFs because, for example, it is easy to show that ρ is a LRF for an *open polyhedron* iff it is a LRF for its topological closure.

Since the underlying algorithms for LRFs and QLRFs are based on the use of Farkas' Lemma [Schrijver 1986, p. 93], which is valid only for non-strict inequities, we could use Motzkin Transposition Theorem [Schrijver 1986, Corollary 7.1k, p. 94] which is similar and valid for non-strict constraints as well (this is not implemented in iRankFinder).

5.1.6 Using iRankFinder

The easiest way to use iRankFinder is through its web-interface[2] that is depicted in Figure 5.2. It includes a set of predefined TSs that can be used and allows users to provide their own TSs. A click on the `Settings` button opens a settings window as depicted in Figure 5.3, which includes a list of different configuration options.

[2] `http://irankfinder.loopkiller.com`

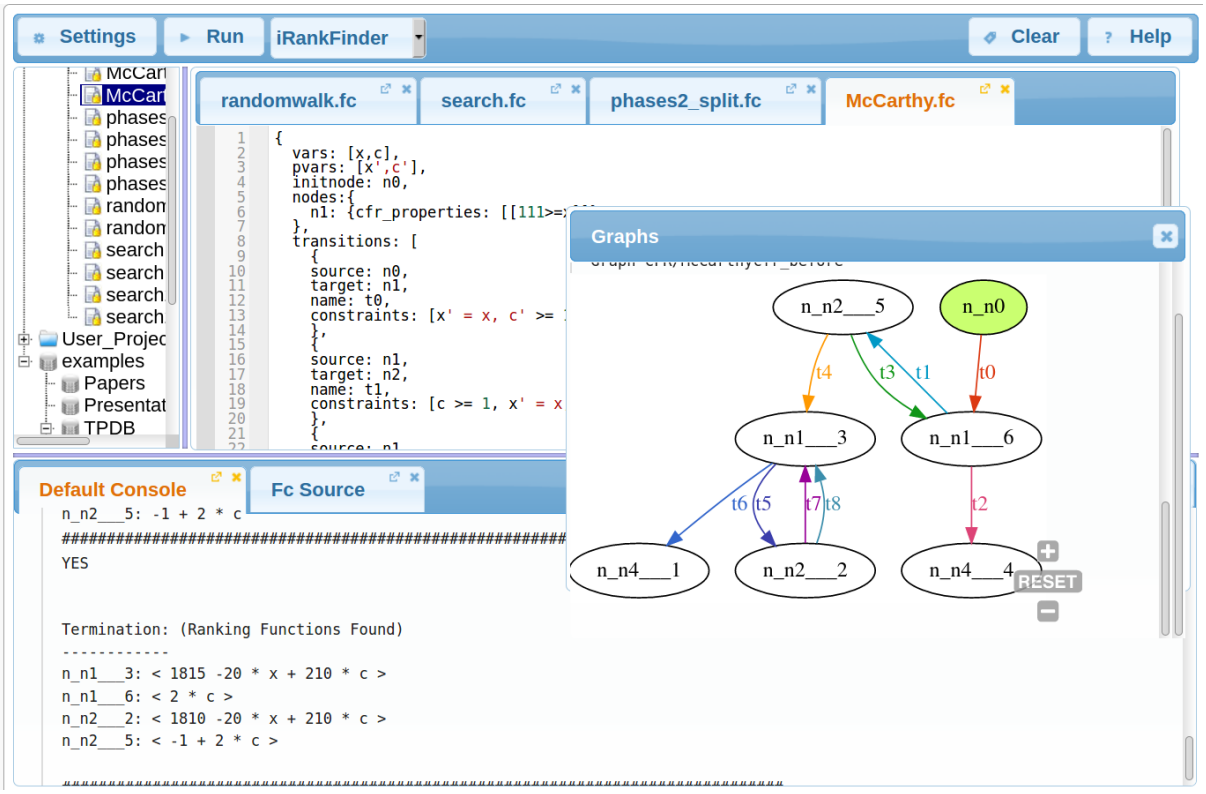


Figure 5.2: iRankFinder web-interface.

This web-interface has been developed using the toolkit described in Section 5.2. Another possibility is to use iRankFinder from a command-line (in `Linux` or `OSX`). Installation and usage instructions for this option are available at the main GitHub repository[3].

We also provide a `Docker`[4] script that can be used to generate a corresponding (`Linux` based) image with all tools installed, including the web-interface. This can be done by first downloading the corresponding `Dockerfile`[5], and then executing the following command in the same directory where `Dockerfile` is saved:

```
> docker build -t irankfinder .
```

Creating the image will take some time, after which the image can be started by executing the following command (only once):

```
> docker run -d -p 8081:80 --name irankfinder irankfinder
```

Now iRankFinder can be locally used through its web-interface that is available at `http://localhost:8081`, or from a `Linux` shell by first logging into the image using

```
> docker exec -it irankfinder bash
```

and then executing

```
> irankfinder.sh -h
```

which lists corresponding usage information.

[3] http:://github.com/costa-group/iRankFinder
[4] http://www.docker.com
[5] http://github.com/costa-group/iRankFinder/blob/master/installer/Dockerfile

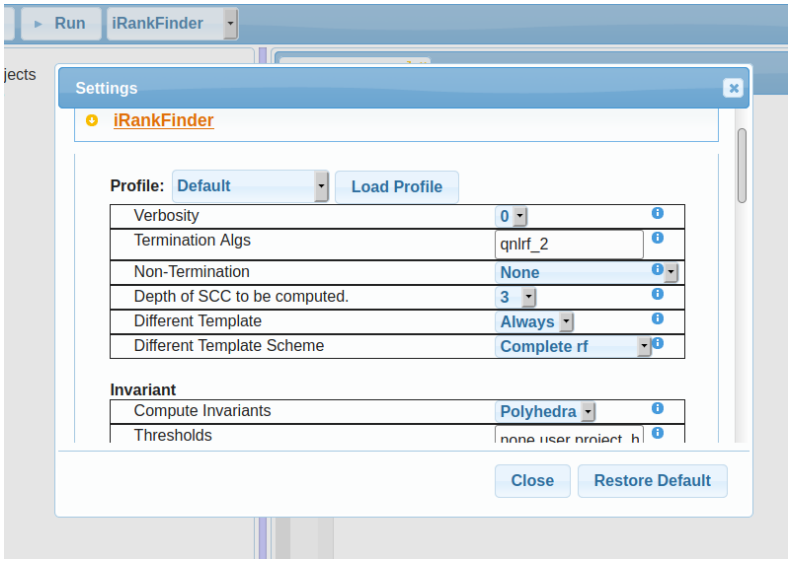


Figure 5.3: iRankFinder settings window.

Finally, several components of iRankFinder are available as separate packages, since they might be interesting for the community, in particular:

- `pyParser`[6] for parsing TSs in different syntax as described in Section 5.1.1, and converting them to some internal representation; and
- `pyLPi`[7] for constructing arithmetic expressions, constraints, polyhedra, and manipulating them with corresponding operations.

This allows developers to use them independently from iRankFinder.

5.2 EasyInterface

In this section, we describe EasyInterface, an open-source toolkit[8] for the rapid development of GUIs for research prototype tools. This toolkit has been developed in the context of this thesis as part of the Envisage[9] project, and also used for the web-interface of iRankFinder and other tools in our research group.

EasyInterface is a toolkit that aims at simplifying the process of building GUIs for research prototypes tools. It provides an easy way to make existing (command-line) applications available via different environments such as a web-interface, within Eclipse, etc. It also defines a text-based output language, called *EasyInterface Output Language* (EIOL), that can be used to improve the way results are shown to the user without requiring any knowledge in GUI/Web programming. For example, if the output of an application is (a structured version of) "*highlight line number 10 of file ex.c*" and "*when the user clicks on line 10, open a dialog box with the text ...*", the web-interface will interpret this and convert it to corresponding visual effects. The advantage of using this output

[6] http://github.com/jesusjda/pyParser
[7] http://github.com/jesusjda/pyLPi
[8] https://github.com/abstools/easyinterface
[9] http://www.envisage-project.eu/

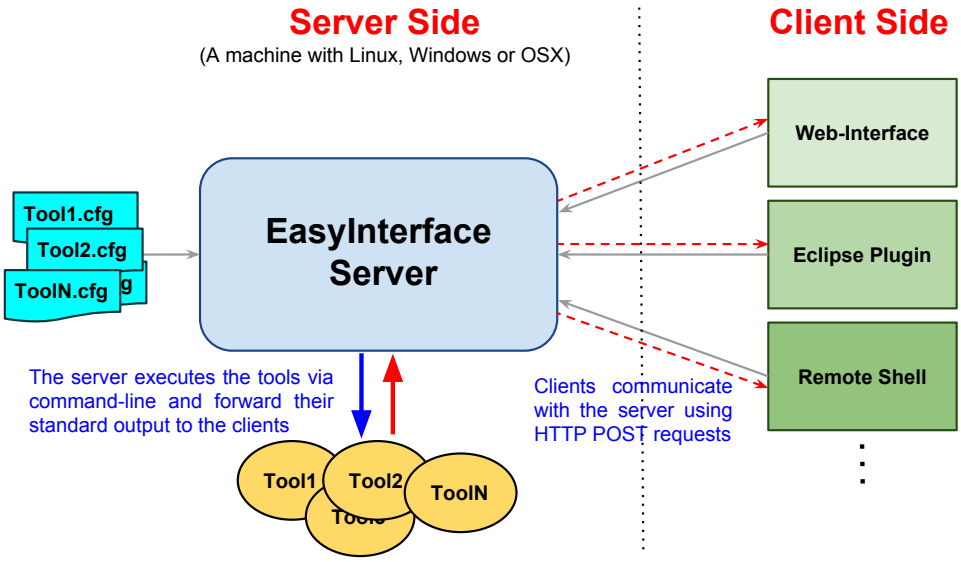


Figure 5.4: The architecture of EasyInterface.

language is that it will be interpreted equally by all environments of EasyInterface, e.g., the web-interface and the Eclipse plugin.

5.2.1 General overview

The overall architecture of EasyInterface is depicted in Figure 5.4. It includes two main components: (1) *server side*: a machine with several tools (the circles with `Tool1`, `Tool2`, etc.) that can be executed from a command-line and that their output go to the standard output. These are the tools that we want to make available for the outside world; and (2) *client side*: several clients that make it easy to communicate with the server side to execute a tool, they might run on the same machine as the server or on other machines, e.g., the web-interface can be installed in the server, as a web-page, but accessed from anywhere using a web browser.

The problem that we want to solve at the server side is to *provide a uniform way to remotely execute locally installed tools*. This problem is solved by the EasyInterface server, which is a collection of PHP programs that run on top of an HTTP server. This server allows specifying how a local tool can be executed and which parameters it takes using simple configuration files (`Tool1.cfg`, `Tool2.cfg`, etc.).

Figure 5.5 is a XML snippet from a configuration file that describes a tool with a unique identifier `myapp`. The `cmdlineapp` tag is a template that describes how to execute the tool from a command-line. Here `_ei_parameters` is a template parameter that will be replaced by an appropriate value before executing the command. The `parameters` tag includes a list of parameters accepted by the tool. For example, the parameter "`c`" above takes one of the values 1 or 2. Parameters can be logically grouped, using the field `group`, and the corresponding interfaces might display them together. The `groups` tag defines the list of groups. Once the configuration file is installed on the server, we can access the tool using an HTTP POST request that includes JSON-formatted data like the snippet in Figure 5.6.

When the server receives such a request, it generates a corresponding shell command according to the specification in the configuration file; e.g., for "`/path-to/myapp -c 1`",

```
<app id="myapp" visible="true">
 ...
 <execinfo method="cmdline">
  <cmdlineapp>/path-to/myapp _ei_parameters</cmdlineapp>
 </execinfo>
 <parameters prefix="-" check="false">
  ...
  <selectone name="c" group="global">
   <option value="1" />
   <option value="2" />
  </selectone>
 </parameters>
 <groups>
  <group id="global">Global parameters</group>
 </groups>
</app>
```

Figure 5.5: An example configuration file of a tool.

```
1 {
2   command: "execute",
3   app_id: "myapp",
4   parameters:*) {
5      c: ["1"],
6      ...
7   },
8   ...
9 }
```

Figure 5.6: An example of a server request.

and then the server executes the command and redirects the standard output back to the client. The server can accept other kinds of requests as discussed later (e.g., to ask which tools are available). In addition to tools, the server side can also include sets of examples to provide users with basic examples on which they can apply the different tools.

Although we now have an easy way to execute tools on the server side, our aim is to simplify this process further by providing GUIs that (1) connect to the server and ask for the list of available tools; (2) let the user select a tool to execute as well as the values of the available parameters for that tool; (3) generate the corresponding request and send it to the server; and (4) display the returned output to the user. The EasyInterface toolkit provides three such clients: a *web-interface* that can be executed in a browser and looks like an IDE; an Eclipse-plugin that runs within the Eclipse IDE; and a remote shell that can be used from the command-line. We focus our discussion on the web-interface.

The web-interface is depicted in Figure 5.7. It is designed like an IDE where users can edit programs, etc. Next to the `Apply` button there is a combo-box with a list of all available tools obtained from the associated servers. In the settings window, the user can select values for the different parameters to be passed to the tool. Note that these parameters are specified in the corresponding configuration file on the server side, they are and automatically converted to combo-boxes, etc., by the web-interface. To execute a selected tool, the user should click the `Apply` button. The web-interface will then send a request to the associated server to execute the corresponding tool and print the received

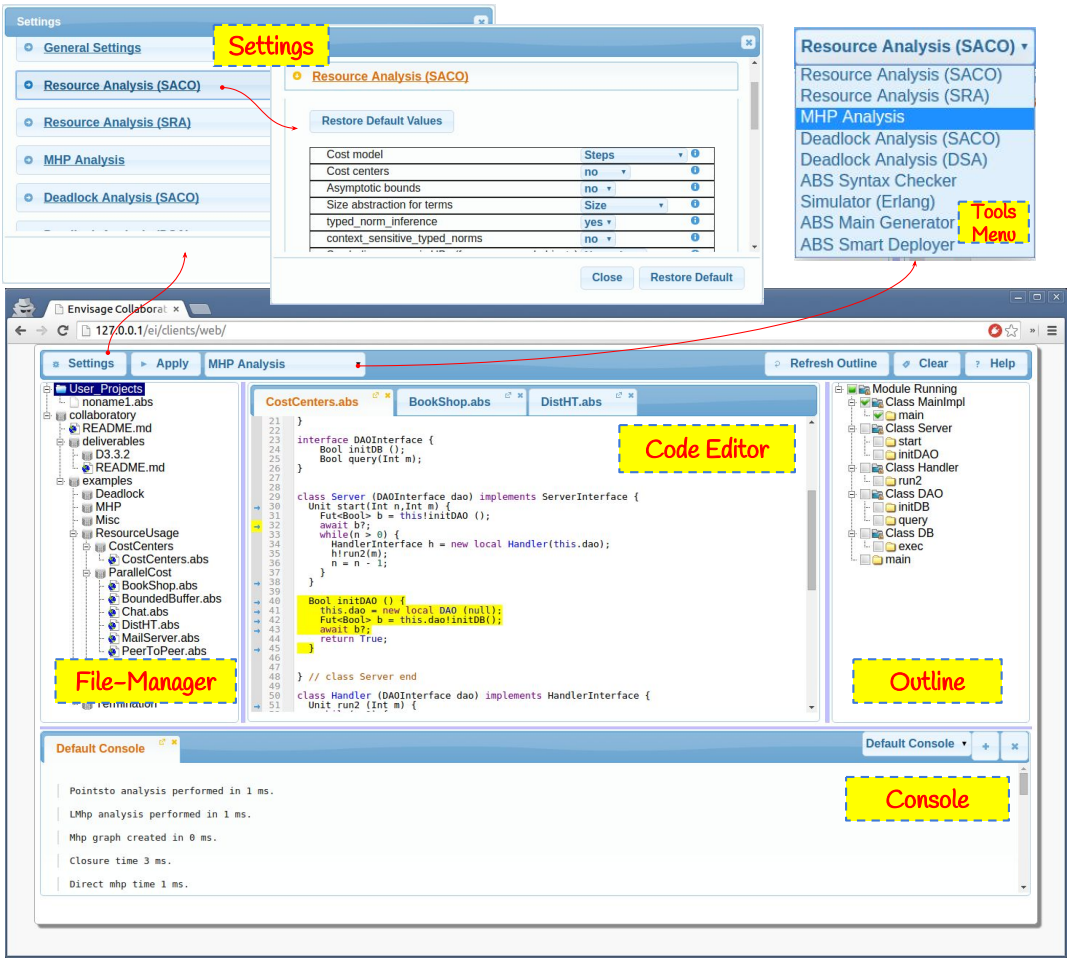


Figure 5.7: EasyInterface Web-Interface Client.

output back in the console area.

Since the web-interface and the Eclipse-plugin are GUI based clients, EasyInterface also gives tools the possibility to generate output that has some graphical effects, such as open dialog-boxes, highlighted code lines, added markers, etc. To use this feature, the tools should be modified to use the EIOL. Figure 5.8 is an example of the output format.

Here, the `highlightlines` tag indicates that lines 5–10 of file `/path-to/sum.c` should be highlighted. The `oncodelineclick` tag indicates that when clicking on line 17, a dialog-box with a corresponding message should be opened. Note that the tool is only modified once to produce such output, and will have similar effect in all EasyInterface clients that support this output language. In the next sections we discuss some features of the server side, the web-interface client, and the EIOL in more detail.

5.2.2 Using EasyInterface

In this section, we describe the methodology that EasyInterface implies for its users by explaining the basics of its different parts and the way they are supposed to be used. Note that the installation of EasyInterface is practically immediate: it only consists in cloning the GitHub repository and making its root directory accessible via an HTTP server. An installation guide is available in the GitHub repository.

```
<highlightlines dest="/path-to/sum.c">
  <lines> <line from="5" to="10"/> </lines>
</highlightlines>
...
<oncodelineclick dest="/path-to/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
  <eicommands>
    <dialogbox boxtitle="Hey!">
      <content format="text"> some message </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

Figure 5.8: EIOL example.

```
<stream execid="EI65231" time="60sec">
 <content format="text">
  The program is running in the background,
  the output goes here ...
 </content>
</stream>
```

Figure 5.9: EIOL stream command example.

Adding Tools and Examples to the Server

As described in Section 5.2.1, adding a tool to the server is simply done by providing a configuration file such as the one in Figure 5.5. This file includes two main components: (1) the command-line template that describes how to execute the corresponding tool; and (2) the parameter section that describes which command-line parameters the tool will take.

EasyInterface supports several types of parameters such as a parameter with a single value, a parameter with multiple values, a Boolean parameter, etc. Each parameter has a name, a set of valid values, and a set of default values. When receiving a request to execute a tool (such as the JSON-formatted data in Figure 5.6), the server generates a sequence of command-line parameters from those specified in the request and replaces the template parameter `_ei_parameters` by this list. For example, if there is a parameter named "`c`" and its provided value is "`1`", then what is passed to the tool is "-c 1". The prefix "-" that is attached to the parameter name can be specified using the `prefix` attribute in the parameters section. In addition, the `check` attribute indicates if the server should reject requests with invalid parameter values. Apart from `_ei_parameters` the command-line template might include other template parameters, all are first replaced by corresponding values and then the resulting shell command is executed and its output is redirected back to the client. Next we describe some of the available template parameters:

- `_ei_files`: tools typically receive input files (e.g., programs) to process. The execution request (i.e., the JSON-formatted data of Figure 5.6) can include such files, and, in order to pass them to the corresponding tool, the server first saves them locally in a temporary directory and replaces `_ei_files` by a list of corresponding local paths.

- `_ei_outline`: since EasyInterface was first developed for tools that process programs, e.g., program analysis tools, the execution request can include so-called

```
<download execid="EI65231" filename="file.zip" />
```

Figure 5.10: EIOL download command example.

outline entities. These are elements of the input programs such as method names, class names, etc., and they are used to, for example, indicate from where the tool should start the analysis. The server replaces `_ei_outline` by the list of the provided outline entities.

- `_ei_execid`: this corresponds to a unique *execution identifier* that is assigned (by the server) to the current execution request, which can be used for future references to this execution as we see next.

- `_ei_stream`: there are tools that do not generate output immediately, such as simulators. In this case we would like to keep the tools running in the background and fetch their output periodically without maintaining the current connection to the server. The template parameter `_ei_stream` corresponds to a temporary directory where the tool can write its output and where clients can fetch this output by corresponding requests to the server. These requests should include the corresponding *execution identifier*. For example, the tool could output the command of Figure 5.9 (in the EIOL) and terminate, while keeping a process running in the background (complying with the server's permissions) writing its output chunks to the `_ei_stream` directory.

  This command indicates that the output, in `text` format, should be fetched every 60 seconds using the execution identifier specified in `execid`. Note that it is the responsibility of the client (e.g., the web-interface) to fetch the output once it receives the above command.

- `_ei_download`: some tools generate output in the form of large files, e.g., compiled code. Instead of redirecting this output directly to the client it can be more convenient to return download links. This template parameter corresponds to a temporary directory where such files can be stored and later fetched by sending a special request to the server with the file name and the corresponding *execution identifier*, similarly to the stream mode above. The EIOL includes a special command for downloading such files (see Figure 5.10).

  Once the client (e.g., the web-interface) receives this command, it sends a request to download the file `file.zip` that is associated with the execution identifier "`EI65231`".

- `_ei_sessionid`: this corresponds to a unique session identifier that is assigned to the user, and can be used to track the user's activity among several requests. It is implemented using PHP sessions.

- `_ei_clientid`: this corresponds to the client identifier, e.g., `webclient`, `eclipse`, etc. It can be used to generate client directed output which uses the EIOL for the selected clients and plain text for others.

The server configuration files also include options for controlling security issues, timeout for tools, etc.

```
<examples>
 <exset id="iter">
  <folder name="Examples_1">
   <folder name="iterative">
    <file  name="sum.c" url="https://.../sum.c" />
    ...
   </folder>
   <folder name="rec">
    <file  name="sum.c" url="https://.../fib.c" />
    ...
   </folder>
  </folder>
 </exset>
 <exset id="set2">
  <folder name="Examples_2">
   <github owner="abstools" repo="absexamples"
           branch="master" path="collaboratory" />
  </folder>
 </exset>
</examples>
```

Figure 5.11: An example configuration file of two sets of examples.

Apart from tools, one can also install example sets on the server side, which are meant to be used by clients (e.g., the web-interface) to provide users with some examples on which they can apply the available tools. Adding example sets is done by adding an XML structure to the sever configuration files like Figure 5.11.

Figure 5.11 defines two sets of examples. The first uses a directory structure, while the second refers to a github repository. Note that in the first case the example files are not necessarily installed on the server, for each we must provide a link from which it can be downloaded.

Using the Web-Interface Client

The web-interface client of EasyInterface is a JavaScript program that runs in a web browser, a screenshot is depicted in Figure 5.7. It uses JQuery[10] as well as some other libraries like JSTree[11] and CodeMirror[12]. It is designed like an IDE, this is because EasyInterface was first developed for with static program analysis tools in mind. It includes the following components: (1) the *code editor*, where users can edit programs; (2) the *file-manager* that contains a list of predefined examples and user files; (3) the *outline view* that includes an outline (e.g., methods and classes) of one or more files; (4) the *console* where the results of executing a tool is printed by default; and (5) the *tool bar* that includes several buttons to execute a tool, etc. Next we describe the work-flow, and give more details on these components.

The work-flow within the web-interface is as follows: (a) write a new program or open an existing one from the file-manager; (b) click on the `Refresh Outline` button to generate the outline of the currently open program, and select some entities from this outline; (c) select a tool from the tools menu; and (d) click on the `Settings` button to set the values of the different parameters; (e) click on the `Apply` button to execute the selected tool. At this point the web-interface sends a request to the corresponding server to execute the selected tool (passing it the currently opened file, parameter values,

[10] http://jquery.com
[11] http://www.jstree.com
[12] http://codemirror.net

selected outline entities, etc.), and the output is printed in the console area. If the tool's output is in the EIOL, then it passes through an interpreter that converts it to some corresponding graphical output. The user can apply a tool (and generate an outline) on several files by selecting the corresponding option from the context menu in the file-manager (opened with a right click on an entry in the file-manager).

The code editor is based on CodeMirror, it can be easily configured (in the web-interface configuration file) to use syntax-highlighting for different languages.

The tools menu includes a list of tools that can be executed by the user. This list can be set in the web-interface configuration file, simply by providing URLs of corresponding EasyInterface servers and, for each, indicating if all available tools should be included or only some selected ones. The default configuration of the web-interfaces fetches all tools that are installed on the server running on the machine where the web-interface is installed.

When clicking the `Settings` button, a settings window is opened where the user can choose values for the different parameters of the different tools (see the top part of Figure 5.7). This is automatically generated using the parameters defined in the server configuration file (the web-interface fetches this information from the corresponding server). Predefined configurations of these parameters are available by selecting a profile. Those profiles can be defined in the server configuration file.

The predefined examples included in the file-manager can be set in the web-interface configuration file, simply by providing URLs to corresponding EasyInterface servers and identifiers for the sets to be included. As we have seen, such a set can simply be given as a reference to a GitHub repository. The file-manager can also allow users to create their own files, upload files from local storage, and clone public and private GitHub repositories.

The Outline area is supposed to include an outline of some of the programs files (available in the file-manager), and thus it depends on the structure of those programs. For example, fro Java programs it might include classes, interfaces, methods, . EasyInterface already provides an easy way to change the outline generator. All we have to do is (1) to provide a tool (installed on an EasyInterface server, like any other tool) that takes a set of files and generates an XML structure that represents an outline, the web-interface will convert this XML to a tree-view; and (2) to modify the web-interface configuration file in order to use this tool for generating the outline.

Using The Output Language

The EIOL is a text-based language that allows tools to generate more sophisticated output. It is supported in both the web-interface and the Eclipse clients. In this section we will explain the basics of this language by example. An output in EIOL is an XML of the form in Figure 5.12.
where [`EICOMMAND`]* is a list of commands to be executed; and [`EIACTION`]* is a list of actions to be declared. An [`EICOMMAND`] is an instruction like: *print a text on the console*, *highlight lines 5-10*, etc. An [`EIACTION`] is an instruction like: *when the user clicks on line 13, highlight lines 20-25*, etc. In the rest of this section we consider some representative commands (Figure 5.13) and actions (Figure 5.14).

Recall that when the EIOL is used, the web-interface does not redirect the output to the console area and thus we need a command to print in the console area. The Figure 5.13a is an example of a command that prints "Hello World" in the console area.

```
<eiout>
 <eicommands>
    [EICOMMAND]*
 </eicommands>
 <eiactions>
    [EIACTION]*
 </eiactions>
</eiout>
```

Figure 5.12: EIOL general scheme.

The value of the `consoleid` attribute is the console identifier in which the given text should be printed (e.g., in the web-interface the console area has several tabs, so the identifier refers to one of those tabs). If a console with the provided identifier does not exist yet, a new one is created, with a title as specified in `consoletitle`. If `consoleid` is not given the output goes to the default console. Inside `printonconsole` we can have several `content` tags which include the content to be printed. The attribute `format` indicates the format of the content. In the above example it is plain `text`, other formats are supported as well, e.g., `html`, `svg`, and `graphs`. The `graphs` option refers to diagrams that are drawn using DyGraphs[13], where the data is provided inside the `content` tag using a JSON based format.

Figure 5.13b shows a command to highlight code lines. Attributes `dest` and `outclass` are as in the `addmarker` command. Each `line` tag defines a region to be highlighted. In the example above, lines 5–10 get highlighted. We can also use the attributes `fromch` and `toch` to indicate the columns in which the highlight starts and ends respectively.

The Figure 5.13c is an example of a command that adds a marker next to a code line in the editor area. The attribute `dest` indicates the *full path* of the file in which the marker should be added. The attribute `outclass` indicates the nature of the marker, which can be `info`, `error`, or `warning`. This value typically affects the type/color of the icon to be used for the marker. The tag `lines` includes the lines in which markers should be added, each line is given using the tag `line` where the `from` attribute is the line number (`line` can be used to define a region in other commands, this is why the attribute is called `from`). The text inside the `content` tag is associated to the marker (as a tooltip, a dialog box, etc., depending on the client).

The command in Figure 5.13d can be used to open a dialog box with some content. The dialog box will be titled as specified in `boxtitle` and it will include the content as specified in the `content` environments. The attributes `boxwidth` and `boxheight` are optional, they determine the initial size of the window.

A *CodeLine action* defines a list of commands to be executed when the user clicks on a line of code (more precisely, on a marker placed next to the line). The commands can be any of those seen above. Figure 5.14a is an example of such an action.

First note that the XML description above should be placed inside the `eiactions` tag. When the above action is executed, e.g., by the web-interface client, a marker will be shown next to line 17 of the file specified in the attribute `dest`. If the user clicks on this marker the commands inside the `eicommands` tag will be executed, and if the user clicks again the effect of these commands is undone. In the case above, a click highlights lines 17–19 and opens a dialog box, and another click removes the highlights and closes the dialog box.

[13] http://dygraphs.com

```
<printonconsole consoleid="1" consoletitle="A Title">
  <content format="text">Hello World</content>
</printonconsole>
```

(a) Printing in the console area.

```
<highlightlines dest="path" outclass="info">
  <lines> <line from="5" to="10"/> </lines>
</highlightlines>
```

(b) Highlighting lines.

```
<addmarker dest="path" outclass="info">
  <lines> <line from="4" /> </lines>
  <content format="text">
    some associated text
  </content>
</addmarker>
```

(c) Adding a marker.

```
<dialogbox outclass="info" boxtitle="A Title"
           boxwidth="100" boxheight="100">
  <content format="html">
     some associated text
  </content>
</dialogbox>
```

(d) Opening a dialog box.

Figure 5.13: Some EIOL commands.

OnClick actions are similar to CodeLine actions. The difference is that instead of being assigned to a line of code, they are assigned to an HTML tag that we have previously generated. Let us suppose that the tool has already generated the content of Figure 5.14b in the console area. Note that the text "10 errors" is wrapped by a span tag with an identifier `err1`. The OnClick action can assign a list of commands to a click on this text as in Figure 5.14c. The selectors used in the tag `selector` are as in JQuery[14].

[14] http://jquery.com

```
<oncodelineclick dest="path" outclass="info" >
  <lines> <line from="17" /> </lines>
  <eicommands>
    <highlightlines>
      <lines> <line from="17" to="19"/> </lines>
    </highlightlines>
    <dialogbox boxtitle="Hey!">
      <content format="html">
         some text
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

(a) When clicking a code line.

```
<content format="html"/>
   <span id="err1">10 errors</span> were found.
</content>
```

(b) Trigger text.

```
<onclick>
  <elements> <selector value="#err1"/> </elements>
  <eicommands>
   <dialogbox boxtitle="Errors">
    <content format="html">some text</content>
   </dialogbox>
  </eicommands>
</onclick>
```

(c) When clicking in #err1 of the above Figure.

Figure 5.14: EIOL actions examples.

Chapter 6

Experimental Evaluation

In this chapter, we experimentally evaluate iRankFinder, in particular, the techniques presented in chapters 3 and 4 that have been implemented in iRankFinder as described in Section 5.1. The raw data used for this evaluation, and additional information, can be found at the following page: http://irankfinder.loopkiller.com/DomenechPhD.

For our evaluation we have used standard sets of benchmarks taken from the *Termination Problem Data Base* (TPDB)[1]:

- (`TPDB-A`) a set of 416 TSs in `smt2` syntax coming from corresponding Java programs (directory `Integer_Transition_Systems/From_Aprove_2014`).
- (`TPDB-B`) a set of 806 TSs in `smt2` syntax coming from not necessarily structured programs (directory `Integer_Transition_Systems/From_T2`).
- (`TPDB-C`) a set of 335 C programs with integer variables (directory `C_Integer`).
- (`TPDB-D`) a set of 781 TSs in `kittle-koat` syntax, coming from different sources (directory `Complexity_TS`).

The first 3 sets are used in the termination analysis competition, and the last one is used in the complexity analysis competition[2].

We divide the evaluation into several sections: Section 6.1 evaluates the effect of our CFR techniques on termination analysis; Section 6.2 evaluates the effect of our CFR techniques on cost analysis; Section 6.3 evaluates the non-termination analysis techniques of Chapter 4, and also the effect of our CFR techniques on non-termination; and Section 6.4 evaluates the effect of our CFR techniques on generating invariants and proving assertions, using the examples of Sharma et al. [2011].

6.1 CFR for Termination Analysis

In this section, we evaluate the effect of Algorithm 2 on iRankFinder, i.e., we compare the results of iRankFinder with and without CFR using different settings. We also compare to other termination analysis tools. We have used sets `TPDB-A`, `TPDB-B` and `TPDB-C`, but excluded benchmarks known to be non-terminating: 151 for `TPDB-A`, 93 for `TPDB-B`, and 93 for `TPDB-C`. They are used in Section 6.3 to evaluate non-termination.

[1] http://termination-portal.org/wiki/TPDB
[2] http://termination-portal.org/wiki

Props	$\texttt{CFR}_\texttt{B}$	$\texttt{CFR}_\texttt{S}$	$\texttt{CFR}_\texttt{A}$	Set
P_1	36 (40.43)	38 (34.20)	39 (38.00)	Succ: 159 (8.16) Fail: 106 (15.75) Succ-CFR: 40 TPDB-A
P_2	11 (31.12)	15 (28.22)	14 (31.71)	
P_3	36 (40.51)	38 (34.16)	39 (38.14)	
P_4	37 (82.07)	39 (53.63)	40 (59.44)	
P_5	27 (45.68)	28 (33.51)	30 (37.98)	
P_6	37 (51.39)	39 (39.80)	40 (45.19)	
P_7	27 (48.90)	28 (34.30)	30 (39.17)	
P_1	131 (330.06)	147 (306.17)	139 (329.02)	Succ: 123 (17.33) Fail: 335 (165.94) Succ-CFR: 182 TPDB-B
P_2	25 (272.19)	128 (268.76)	93 (278.81)	
P_3	131 (330.36)	147 (307.95)	139 (331.65)	
P_4	136 (411.52)	159 (343.86)	152 (372.06)	
P_5	61 (313.59)	148 (311.79)	119 (332.19)	
P_6	156 (364.58)	172 (345.83)	166 (369.26)	
P_7	61 (330.80)	144 (313.83)	120 (331.66)	
P_1	65 (41.31)	51 (28.63)	67 (35.74)	Succ: 74 (3.68) Fail: 168 (21.25) Succ-CFR: 90 TPDB-C
P_2	19 (30.68)	13 (23.95)	17 (29.45)	
P_3	65 (41.28)	51 (28.53)	67 (35.82)	
P_4	82 (70.79)	68 (45.39)	82 (56.62)	
P_5	40 (35.89)	42 (28.26)	46 (33.28)	
P_6	79 (45.53)	76 (32.25)	81 (38.49)	
P_7	43 (31.84)	43 (28.45)	46 (28.75)	

Table 6.1: Summary of evaluation of CFR for termination analysis using LRFs.

Props	$\texttt{CFR}_\texttt{B}$	$\texttt{CFR}_\texttt{S}$	$\texttt{CFR}_\texttt{A}$	LLRF	
P_1	8 (32.25)	7 (38.77)	9 (37.75)	Succ: 229 (32.26) Fail: 36 (24.69) Succ-CFR: 9	TPDB-A
P_2	7 (29.28)	4 (36.40)	7 (35.37)		
P_3	8 (32.34)	7 (38.81)	9 (37.60)		
P_4	8 (44.42)	8 (42.54)	9 (40.38)		
P_5	7 (38.88)	4 (41.62)	8 (41.31)		
P_6	8 (41.45)	7 (43.46)	9 (42.81)		
P_7	7 (39.69)	4 (41.72)	8 (41.41)		
P_1	21 (263.17)	14 (285.09)	17 (285.37)	Succ: 373 (93.20) Fail: 85 (265.51) Succ-CFR: 29	TPDB-B
P_2	7 (258.57)	3 (274.69)	4 (275.65)		
P_3	21 (263.43)	14 (285.32)	17 (287.19)		
P_4	25 (255.90)	16 (283.49)	19 (284.91)		
P_5	23 (238.51)	13 (272.41)	16 (276.12)		
P_6	28 (248.67)	19 (279.66)	21 (282.36)		
P_7	25 (232.75)	13 (272.79)	16 (273.98)		
P_1	27 (49.87)	18 (65.97)	25 (69.03)	Succ: 175 (23.60) Fail: 67 (47.27) Succ-CFR: 34	TPDB-C
P_2	12 (39.18)	5 (59.86)	10 (60.68)		
P_3	27 (50.02)	18 (65.98)	25 (69.17)		
P_4	34 (47.98)	21 (63.31)	30 (67.58)		
P_5	19 (32.97)	10 (56.79)	17 (60.12)		
P_6	29 (35.38)	20 (59.46)	26 (63.19)		
P_7	20 (33.24)	10 (56.93)	17 (60.23)		

Table 6.2: Summary of evaluation of CFR for termination analysis using LLRFs.

We first analysed all benchmarks for termination *without* CFR in two *settings*: using only LRFs; and using LLRFs with the definition of *nested* QLRF of Section 5.1.4 and $d = 3$. For those on which iRankFinder failed to prove termination, we analysed them again *with* CFR using different sets of properties and CFR schemes. The results for LRFs and LLRFs are summarised in tables 6.1 and 6.2.

Both tables consist of 3 (vertical) blocks, one for each set of benchmarks. On the right side of each block, we indicate: the total number of benchmarks on which iRankFinder succeeded and failed *without* CFR (excluding non-terminating ones), and the corresponding total time in minutes; and the additional number of benchmarks on which iRankFinder succeeded when using CFR. For example: in the first block of Table 6.1, the last column indicates that iRankFinder succeeded (resp. failed) on 159 (resp. 106) benchmarks and the total analysis time was 8.16min (resp. 15.75min); and that when using CFR it succeeds on additional 40 benchmarks.

For each set, columns correspond to different CFR schemes, and rows to properties used for CFR. These properties are subsets of those defined in Section 3.1:

$$
\begin{aligned}
P_1 &= \mathtt{Props}_{\mathtt{c}} \\
P_2 &= \mathtt{Props}_{\mathtt{h}} \\
P_3 &= \mathtt{Props}_{\mathtt{c}} \cup \mathtt{Props}_{\mathtt{h}} \\
P_4 &= \mathtt{Props}_{\mathtt{c}} \cup \mathtt{Props}_{\mathtt{cv}} \cup \mathtt{Props}_{\mathtt{h}} \cup \mathtt{Props}_{\mathtt{hv}} \\
P_5 &= \mathtt{Props}^{\mathtt{d}}_{\mathtt{h}} \\
P_6 &= \mathtt{Props}^{\mathtt{d}}_{\mathtt{h}} \cup \mathtt{Props}_{\mathtt{c}} \\
P_7 &= \mathtt{Props}^{\mathtt{d}}_{\mathtt{h}} \cup \mathtt{Props}_{\mathtt{h}}
\end{aligned}
$$

Each entry in the tables includes the number of benchmarks that iRankFinder can now prove terminating, and the total time it took to analyse all benchmarks, using the scheme and properties as indicated in the corresponding row and column. Each benchmark was given a 5 minutes time limit. In terms of precision:

- For `TPDB-A`: without using CFR iRankFinder proves termination of 159 (resp. 229) benchmarks when using LRFs (resp. LLRFs), and with CFR iRankFinder proves termination of 40 (resp. 9) more benchmarks.
- For `TPDB-B`: without using CFR iRankFinder proves termination of 123 (resp. 373) benchmarks when using LRFs (resp. LLRFs), and with CFR iRankFinder proves termination of 182 (resp. 29) more benchmarks.
- For `TPDB-C`: without using CFR iRankFinder proves termination of 74 (resp. 175) benchmarks when using LRFs (resp. LLRFs), and with CFR iRankFinder proves termination of 90 (resp. 34) more benchmarks.

In total, out of 609 (resp. 188) benchmarks that iRankFinder cannot prove terminating without CFR using LRFs (resp. LLRFs), with CFR it proves terminating of 312 (resp. 72) more benchmarks, which is an improvement of about 51.2% (resp. 38.2%).

Experiments show that properties $\mathtt{Props}^{\mathtt{d}}_{\mathtt{h}}$ and $\mathtt{Props}_{\mathtt{c}}$ are important for precision and that using them with scheme $\mathtt{CFR}_{\mathtt{S}}$ results in the best performance/precision trade-off. The time spent on CFR can be up to 42% of the total time, depending on the scheme and properties used – as expected, $\mathtt{CFR}_{\mathtt{B}}$ takes more time than $\mathtt{CFR}_{\mathtt{A}}$ and $\mathtt{CFR}_{\mathtt{S}}$.

Times in the tables, however, should be interpreted carefully, since when analysing *without* CFR and when using $\mathtt{CFR}_{\mathtt{B}}$ the analysis stops as soon as it fails to prove termination of a SCC, while when using $\mathtt{CFR}_{\mathtt{S}}$ and $\mathtt{CFR}_{\mathtt{A}}$ it keeps trying to prove termination of other SCCs.

Set	iRankFinder	iRankFinder (CFR)	VeryMax	T2
TPDB-A	229	238	219	231
TPDB-B	373	402	409	382
TPDB-C	175	209	211	176
TOTAL	777	849	839	789

Table 6.3: Total number of benchmarks for which each tool could prove termination.

	iRankFinder	iRankFinder (CFR)	VeryMax	T2
iRankFinder	–	0	20	33
iRankFinder (CFR)	72	–	29	66
VeryMax	82	19	–	73
T2	47	11	23	–

Table 6.4: Comparison of tools on individual benchmarks. Each cell indicates the number of benchmarks the tool at the corresponding row could prove terminating, and the tool at the corresponding column could not.

We have also compared to other termination analysers of TSs: VeryMax [Borralleras et al. 2017] and T2 [Brockschmidt et al. 2016a]. The results are summarised in tables 6.3 and 6.4. Note that these tables include two versions of iRankFinder: with and without CFR. Table 6.3 includes the total number of benchmarks for which each tool could prove terminating. We can see that, in total, iRankFinder with CFR could prove termination of more examples than any other tool. Table 6.4 includes a more detailed comparison. The number x at row i and column j is the number of benchmarks for which tool i could prove termination and tool j could not. We can see that the column of iRankFinder with CFR has the lowest numbers, i.e., it is the one that loses less precision.

6.2 CFR for Cost Analysis

In this section, we evaluate the effect of our CFR techniques on cost analysis, in particular using the tool KoAT and the set TPDB-D (781 benchmarks). We did not analyse TPDB-A, TPDB-B and TPDB-C since they are not suitable for complexity analysis, e.g., most of them include concrete initial values which would always induce constant asymptotic upper-bounds, and thus it is not easy to quantify improvements.

For each benchmark, we applied KoAT without CFR and with CFR as a pre-processing step using properties P_1-P_7 of the previous section. Each benchmark was given a 5 minutes time limit for CFR and 3 minutes time limit for KoAT. Table 6.5 summarises the improvements obtained.

Rows correspond to sets of properties, and columns to asymptotic upper-bounds which are possible outcomes of KoAT. A number x in row P_i and column $O(f(n))$ means the following: when using CFR with properties P_i, x benchmarks for which KoAT infers $O(f(n))$ without CFR, have been improved to a lower asymptotic upper-bounds. For example, using P_1, the use of CFR improves 13 benchmarks from $O(n^2)$ to a lower asymptotic upper-bound and using P_2, the use of CFR improves 17 benchmarks from $O(\infty)$ to a lower asymptotic upper-bound, etc. The last row includes the total number

Props	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n^4)$	$\geq O(n^5)$	$O(EXP)$	$O(\infty)$	TOTAL
P_1	3	14	3	0	0	0	18	38
P_2	2	15	4	0	0	0	17	38
P_3	1	16	3	0	0	0	17	37
P_4	2	16	2	0	1	0	19	39
P_5	3	15	3	1	3	0	14	39
P_6	3	16	3	0	2	0	19	43
P_7	3	15	4	1	2	0	15	40
–	3	16	4	1	3	0	22	49

Table 6.5: Evaluation of CFR for Cost Analysis.

of benchmarks for which the use of CFR achieves improvements for the corresponding asymptotic upper-bound, using any of properties P_1-P_7. Overall, using CFR we get tighter asymptotic upper-bounds for 49 benchmarks, which are about 6% of all benchmarks; and using properties P_6, which consists of $\texttt{Props}_{\texttt{h}}^{\texttt{d}}$ and $\texttt{Props}_{\texttt{c}}$, give the best precision improving in 43 cases.

Note that in this evaluation, `KoAT` timeouts more often when using CFR, however, this is expected as CFR is applied to the whole TS. In a new version of `KoAT`, that is under development, the developers incorporated our CFR procedure in a way that is similar to scheme $\texttt{CFR}_{\texttt{s}}$ that was used for termination. We expect this to improve performance.

We have also evaluated our CFR procedure on all examples of Figure 1 and Figure 2 of Gulwani et al. [2009] and got similar refinements. Their tool is not available so we cannot experimentally compare to them. We did not run `PUBs` on this set since it is not directly designed for TSs. Transforming TSs to CRSs without losing precision requires more work (e.g., inferring loop summaries which are usually done by the frontend that uses `PUBs`). We did not run `CoFloCo` on this set since it includes a CFR component for CRSs. However, as we have seen in the examples of Section 3.2, our CFR procedure can improve the precision of both `PUBs` and `CoFloCo`.

6.3 Non-Termination via Recurrent Sets (and CFR)

For experimentally evaluating Algorithm 3 for non-termination of TSs, we have integrated the procedure described in Section 4.2.3 in iRankFinder (see Section 5.1.4). In a nutshell, when iRankFinder fails to prove termination, it enumerates closed walks using only transitions whose termination was not proven and then applies Algorithm 3 to seek recurrent sets in these closed walks. In addition, it checks that the recurrent set is reachable as described in Section 5.1.4. When the domain of the variables is $\mathbb{Z}$, iRankFinder tries to avoid non-determinism by sampling integer values for (local) variables that cause non-determinism. Recall that soundness cannot be guaranteed for $\mathbb{Z}$ without making closed walks deterministic (see sections 4.2.2 and 4.2.3).

We have analysed 707 benchmarks from sets `TPDB-A`, `TPDB-B`, and `TPDB-C`. They include benchmarks where iRankFinder failed to prove termination, and others that are known to be non-terminating (those excluded in Section 6.1). We have applied the analysis in two settings, the first assumes that variables range over $\mathbb{Q}$ and the second assumes that variables range over $\mathbb{Z}$. Clearly the case of $\mathbb{Q}$ is not interesting since all these benchmarks use integer variables, however, it will give an indication on the effectiveness

Set	iRankFinder			
	$\mathbb{Q}$		$\mathbb{Z}$	
	Found	Reach.	Found	Reach.
TPDB-A	154	133	123	100
TPDB-B	330	301	308	276
TPDB-C	101	93	97	83
TOTAL	585	527	528	459

Table 6.6: Evaluation of the non-termination algorithm of Section 4.2.3.

of our method for making loops deterministic as explained below.

The results are summarised in Table 6.6. For each set of benchmarks, we indicate the number of benchmarks for which iRankFinder finds a recurrent set, and the number of those it succeeds to prove reachable. The results are shown for both $\mathbb{Q}$ and $\mathbb{Z}$. Obviously, for $\mathbb{Q}$ we find more recurrent sets than for $\mathbb{Z}$. Roughly, the difference between columns "Found" of $\mathbb{Q}$ and $\mathbb{Z}$ indicates the number of benchmarks where it did not succeed to make the corresponding closed walks deterministic, so we could not apply the analysis for $\mathbb{Z}$ on those benchmarks (in total 57).

Overall, for $\mathbb{Q}$ iRankFinder succeeds to find recurrent sets for 585 TSs, and proves that 527 are reachable; and for $\mathbb{Z}$ it succeeds to find recurrent sets for 528 TSs, and proves that 459 are reachable. We can see that although our procedure for reachability is very simply, it obtains reasonable results in practice.

We have also evaluated the effect of using CFR on non-termination. For this, we have analysed 164 (resp. 232) benchmarks where iRankFinder could not prove non-termination when variables range over $\mathbb{Q}$ (resp. $\mathbb{Z}$), using CFR schemes and properties as in Section 6.1. Table 6.7 summaries our results, its structure is similar to that of tables 6.3 and 6.4 of Section 6.1. Each entry indicates the number of benchmarks for which iRankFinder finds a reachable recurrent set, using the corresponding CFR scheme and properties. In terms of precision, CFR improves:

- For TPDB-A: iRankFinder CFR proves non-termination of 6 (resp. 10) more benchmarks when using the domain $\mathbb{Q}$ (resp. $\mathbb{Z}$).
- For TPDB-B: iRankFinder CFR proves non-termination of 16 (resp. 25) more benchmarks when using the domain $\mathbb{Q}$ (resp. $\mathbb{Z}$).
- For TPDB-C: iRankFinder CFR proves non-termination of 7 (resp. 5) more benchmarks when using the domain $\mathbb{Q}$ (resp. $\mathbb{Z}$).

In total, out of 164 (resp. 232) benchmarks that iRankFinder without CFR cannot prove non-terminating using the domain $\mathbb{Q}$ (resp. $\mathbb{Z}$), it proves non-termination of 29 (resp. 40) benchmarks when using CFR, which is an improvement of about 17.6% (resp. 17.2%). The table also indicates that scheme $\texttt{CFR}_{\texttt{S}}$ is not useful for non-termination and that properties $\texttt{Props}^{\texttt{d}}_{\texttt{h}}$ and $\texttt{Props}_{\texttt{c}}$ are important for precision. This is because non-termination, in principle, improves the reachability check rather than the process of finding a recurrent set.

We have also compared to other non-termination analysis tools: VeryMax [Borralleras et al. 2017], T2 [Brockschmidt et al. 2016a] and LoAT [Frohn and Giesl 2019]. The results are summarised in tables 6.8 and 6.9. Note that these tables include iRankFinder

Props	CFR_B	CFR_S	CFR_A	$\mathbb{Q}$		
P_1	4	0	2			
P_2	2	0	1			
P_3	4	0	2			
P_4	2	0	2			
P_5	2	0	1			
P_6	5	0	4			
P_7	2	0	1			
P_1	11	1	4			
P_2	3	1	1			
P_3	11	1	5			
P_4	12	0	4			
P_5	6	0	1			
P_6	14	1	5			
P_7	6	1	2			
P_1	6	0	6			
P_2	2	0	2			
P_3	6	0	6			
P_4	6	0	5			
P_5	4	0	4			
P_6	7	0	6			
P_7	4	0	4			

Props	CFR_B	CFR_S	CFR_A	$\mathbb{Z}$		
P_1	8	0	7			
P_2	5	0	4			
P_3	8	0	7			
P_4	7	0	7			
P_5	4	0	4			
P_6	9	0	9			
P_7	4	0	4			
P_1	20	0	15			
P_2	11	0	10			
P_3	20	0	15			
P_4	22	0	14			
P_5	15	0	10			
P_6	23	0	15			
P_7	15	0	10			
P_1	4	0	4			
P_2	2	0	2			
P_3	4	0	4			
P_4	4	0	4			
P_5	4	0	3			
P_6	4	0	4			
P_7	4	0	3			

Table 6.7: Evaluation of CFR for Non-Termination.

Set	iRankFinder		VeryMax	T2	LoAT
	no CFR	CFR			
TPDB-A	100	110	131	115	140
TPDB-B	276	301	280	323	356
TPDB-C	83	88	98	77	97
TOTAL	459	499	509	515	593

Table 6.8: The number of benchmarks for which each tool could prove Non-Termination.

	iRankFinder	iRankFinder (CFR)	VeryMax	T2	LoAT
iRankFinder	–	0	68	53	8
iRankFinder (CFR)	40	–	77	65	16
VeryMax	97	88	–	72	33
T2	109	81	46	–	5
LoAT	142	107	71	71	–

Table 6.9: Comparison of tools on individual benchmarks. Each cell indicates the number of benchmarks the tool at the corresponding row could prove non-terminating, and the tool at the corresponding column could not

with and without CFR, and in both cases we refer to applying it using $\mathbb{Z}$ and checking reachability as well (which is the setting used by the other tools).

Table 6.8 includes the total number of benchmarks for which each tool could prove non-terminating. We can see that, in total, iRankFinder with CFR is comparable to `VeryMax` and `T2`. Table 6.9 includes a more detailed comparison. The number x at row i and column j is the number of benchmarks for which tool i could prove non-termination and tool j could not. We can see that iRankFinder (with CFR) could prove non-termination of many programs that other tool cannot, though, in general, it is less precise. Note that we expected these tools to give better results than iRankFinder, since their techniques were developed particularly for non-termination of TSs, unlike ours that was developed for SLC loops and then generalised for TSs. However, we can see that our results are in some sense comparable to the results of `VeryMax` and `T2`.

6.4 Other Experiments with CFR

We used the invariants generator of iRankFinder to prove the assertions in 13 programs from Sharma et al. [2011]. Without CFR, it proved the assertions for 5 of them, and with CFR it did so for all benchmarks. Also here, $\texttt{Props}^{\texttt{d}}_{\texttt{h}}$ and $\texttt{Props}_{\texttt{c}}$ provided the most precise results. The tool of Sharma et al. [2011] is not available so we cannot experimentally compare to them on other benchmarks.

Chapter 7

Related Work

This chapter discusses works related to the topics addressed in this thesis. We divide the discussion into three parts, each concentrates on a topic, as follows: ranking functions (Section 7.1), non-termination analysis (Section 7.2), and control-flow refinement (Section 7.3).

7.1 Terminating Analysis Using Ranking Functions

Ranking functions have been used for proving termination of programs since the early work of Turing [1949], and corresponding practical and theoretical aspects have been studied extensively. Ranking functions that are based on the use of linear function or tuples of linear function received special attention, mainly because synthesising them can be done efficiently using polynomial-time linear programming techniques. In this category, LRFs and LLRFs are widely used in practical termination analysers.

Probably the most popular work for synthesising LRFs is the one of Podelski and Rybalchenko [2004], mainly due to its use in the Terminator tool [Cook et al. 2006], which demonstrated the use of LRFs in termination analysis of complex, real-world programs. However, the problem of synthesising LRFs has been addressed by other researches using similar ideas [Colón and Sipma 2001; Mesnard and Serebrenik 2008; Podelski and Rybalchenko 2004; Sohn and Gelder 1991]. Bagnara et al. [2012] discuss the relationship between some of these techniques in more details.

All these work observe that synthesising LRFs can be done by inferring inequalities implied by the corresponding transition polyhedra, which can be done in polynomial-time using linear programming: the techniques of Sohn and Gelder [1991] and Mesnard and Serebrenik [2008] are based on the duality theorem of linear programming [Schrijver 1986, p. 92], and those of Colón and Sipma [2001] and Podelski and Rybalchenko [2004] are based on Farkas' lemma [Schrijver 1986, p. 94]. These methods are complete when variables range over the rationals but not the integers.

Ben-Amram and Genaim [2013] are the first to provide a complete algorithm for inferring LRFs for the integers. It is based on first computing the integer-hull of the transition polyhedra and then applying the algorithms of the rational case. They also show that the underlying decision problem is co-NP complete. Cook et al. [2013a] also mention the use of integer-hull to handle the integer case, however, they do not address the complexity of the corresponding decision problem. Bradley et al. [2005b] addressed the integer case as well, however, their technique is not based on linear programming and is not complete.

Since LRFs do not suffice for all programs, several algorithms for synthesising LLRFs have been developed. The earliest work that we are aware of is that of Colón and Sipma [2002] which employs linear programming techniques, and LLRFs have been studied in several works since then [Alias et al. 2010; Ben-Amram and Genaim 2014; Bradley et al. 2005a]. All these works can be viewed uniformly using the notion of QLRFs as we did in Section 2.3.1. They define the notion of LLRF in slightly different ways, which means that for a given program a LLRF might exist according to one definition but not other definitions. Ben-Amram and Genaim [2015] discusses the relative power and complexity issues of these works.

As in the case of LRFs, these techniques have polynomial-time algorithms that are based on linear programming, though Bradley et al. [2005a] use non-linear constraints solving since they simultaneously infer supporting invariants. Completeness is guaranteed when variables range over the rationals, and over the integers, Ben-Amram and Genaim [2014] show that completeness can be achieved by computing the integer-hull as for the case of LRFs. The work of Larraz et al. [2013] uses the most general definition for LLRFs. Their underlying algorithm is based on the use of Max-SMT rather than linear programming and is not complete. They infer supporting invariants as well. There are other techniques [Brockschmidt et al. 2013; Cook et al. 2013b; Harris et al. 2011] that are based on a CEGAR loop and non-linear constraint solving.

The class MΦRFs is a special case of LLRF that has recently triggered the interest of several researchers. Ben-Amram and Genaim [2017] show that for SLC loops MΦRFs are equivalent of the notion of *nested* ranking functions [Leike and Heizmann 2015], and provide a complete polynomial-time synthesis algorithm when variables range over the rationals. For the case of integer variables, completeness can be achieved by computing the integer-hull first. They also show that for SLC loops MΦRFs have the same power as the most general notion of LLRFs [Larraz et al. 2013]. MΦRFs for general loops (which cover TSs) have been considered by Leike and Heizmann [2015] and Li et al. [2016], where both use non-linear constraint solving. Bagnara and Mesnard [2013] study the notion of "eventual linear ranking functions", which are MΦRFs of depth 2. The approach described by Borralleras et al. [2017] is also able to infer MΦRFs for general loops incrementally, by solving corresponding safety problems using Max-SMT.

There are other works [Cousot and Cousot 2012; Urban 2013; Urban and Miné 2014] that address the problem of proving termination by ranking functions, in particular Urban and Miné [2014] that combines piecewise-linear functions with lexicographic orders. None considers recurrent sets together with ranking-function termination proofs. The combination of piecewise-linear functions with lexicographic orders as used by Urban and Miné [2014] subsumes MΦRFs, however, being more general, and using an approach which is more generic, Urban and Miné [2014] does not offer any particular insights about MΦRFs and makes no claims of completeness. Cousot [2005] used Lagrangian relaxation for inferring possibly non-linear ranking functions. In the linear case, Lagrangian relaxation is similar to the affine form of Farkas' lemma.

7.2 Non-termination Analysis

Non-termination provers are described in several works, and the underlying techniques of some are based on finding recurrent sets in one form or another, while others are based on reducing the problem to proving non-reachability of terminating states.

Gupta et al. [2008] describe an algorithm that first generates candidate lassos, and then seeks recurrent sets on each lasso using constraint solving. Larraz et al. [2014] suggest a technique that infers recurrent sets using the notion of quasi-invariants. These are invariants that are guaranteed to hold only from some point of the execution on. This technique is probably the most powerful among all works we are aware of. Payet and Spoto [2009]; Payet et al. [2014] reduce non-termination of simple Java programs to non-termination of corresponding simple constraint logic programming. Bakhirkin et al. [2015] compute recurrent sets using backwards analysis and trace partitioning, and Bakhirkin and Piterman [2016] search for non-termination witnesses in a corresponding abstract graph. Brockschmidt et al. [2011] define several notions of non-termination and reduces them to satisfiability of corresponding constraints. Iosif et al. [2014] describe a method for inferring the *weakest non-termination precondition* for octagonal SLC loops. Leike and Heizmann [2018] suggest a non-termination witness for SLC loop that is called a *geometric non-termination argument*, which induces recurrent sets as well. This is very related to our notion of monotonic recurrent sets that we have discussed in Section 4.2.2. Frohn and Giesl [2019] suggest a non-termination analysis that is based on loop acceleration. Other works [Chen et al. 2014; Le et al. 2015; Velroyen and Rümmer 2008] are based on reducing the problem to that of proving the non-reachability of terminating states.

The idea of shrinking a set of states until finding a recurrent set, like ours, can be found in several of these works, the main difference is that they typically remove states that ensure termination while our procedure might remove non-terminating states (so that, when it finds a recurrent set, it is not necessarily the largest one).

7.3 Control-Flow Refinement

CFR for some variations of CFGs has been considered before in the context of cost analysis [Flores-Montoya and Hähnle 2014; Gulwani et al. 2009], mainly to improve the precision. Roughly, they explore different combinations of the paths of a given loop to discover execution patterns and then transform the loop to follow these patterns. The use of partial evaluation for CFR is not directly comparable to these works; however, experiments show that we achieve similar results. In particular, in the examples of Section 3.2, we discussed some cases that Flores-Montoya and Hähnle [2014] cannot handle, and in Section 6.2 we have seen that we achieve similar refinements for the examples of Gulwani et al. [2009]. Albert et al. [2019] suggest a CFR technique that uses the information in the termination proof to transform and simplify the control-flow of the original program. It is mainly used in the context of a cost analysis of C program, and can be seen as a generalisation of the ideas of Section 3.3.2 – it has been developed in parallel to ours.

Sharma et al. [2011] use CFR to improve invariants generation. Their technique is based on finding a *splitter predicate* such that when it holds one part of the loop is executed and when it does not hold another part is executed. The loop is then rewritten as two consecutive loops, where the predicate is required to hold in one and not to hold in the other. iRankFinder could prove when using CFR, all assertions in the examples of Sharma et al. [2011]. Moreover, we conjecture that the partial evaluation algorithm achieves the same refinement as the technique of Sharma et al. [2011], provided that the splitter predicates (or their negations) are among the properties provided to the algorithm. Sharma et al. [2011] generate candidate splitter predicates from the conditions occurring

in loops, using the weakest precondition operator to project them onto the loop head, which corresponds closely to our property generation $\texttt{Props}^{\texttt{d}}_{\texttt{h}}$.

There is a close relationship between partial evaluation of logic programs (sometimes called partial deduction) and abstract interpretation of logic program wrt. to a goal (top-down abstract interpretation); both can be expressed in a single generic framework parameterised by an abstract domain and an unfolding strategy [Leuschel 2004; Puebla et al. 1999, 2006]. The combination of the two techniques can be mutually beneficial, as shown by Puebla et al. [2006]. In top-down abstract interpretation, the aim is to derive call- and answer-patterns, which are described by abstract substitutions over program variables, whereas in partial evaluation, the aim is to unfold parts of the computation and derive a program specialised for the given goal. The versions in a poly-variant partial evaluation correspond to multiple call-patterns in top-down abstract interpretation. Viewing CFR as an instance of a generic framework, the choice of abstract domain and unfolding strategy are crucial, and therefore this paper focuses on those aspects. There are many ways to achieve poly-variance in partial evaluation, and obtaining a good set of versions leading to useful refinements requires careful choice of the abstract domain, which in our case is the power set of the given set of properties. The control of unfolding is also critical, as unfolding choice predicates leads to a trade-off between specialisation and blow-up in the size of the specialised program.

Logic program specialisation has been previously applied as a component in program verification tools, with a goal similar to the one in this paper. Namely, the specialisation of a program with respect to a goal (corresponding to a property to be proved) can enable the derivation of more precise invariants, contributing to a proof of the property [De Angelis et al. 2012; Fioravanti et al. 2012; Kafle et al. 2018; Leuschel and Massart 2000]. Poly-variant specialisation is often crucial, allowing (in effect) the inference of disjunctive invariants. Again, for CFR the choices for abstraction and unfolding strategy are important, to achieve the right balance between precision and the size of the specialised programs. De Angelis et al. [2012] use an operation called constrained generalisation; this identifies constraints on a predicate that determines control flow. A generalisation operator on constraints is designed to preserve the control flow. This has a relation to property-based abstraction but we find it more natural and controllable to let the properties determine the control flow rather than the other way round. However, further evaluation of different abstractions is needed. We performed some experiments on examples from Section 3.2 using a general-purpose logic program specialiser (ECCE [Leuschel et al. 2006]) but the abstraction used in that tool did not result in any useful CFR.

Chapter 8

Conclusions and Future Work

The research in this thesis started with our interest in the problem of proving termination using ranking functions, in particular ranking functions that are based on linear or tuples of linear functions, such as LRFs and LLRFs. Our initial goals were: (i) to explore on properties of classes of ranking functions that did not receive enough attention from the community, in particular MΦRFs; and (ii) to explore on opportunities and techniques for improving the precision of termination analysis that is based on such ranking functions, in particular using CFR. However, our research has led us to additional directions: (a) the first one that is particularly surprising is about a relation between non-termination analysis and ranking functions; (b) the second is about a new program representation that provides a completely new look at MΦRFs and termination of SLC loops in general; and (c) the third one is related to the applicability of the CFR techniques, developed for improving the precision of termination analysis, to other program analysis.

Our research on using CFR to improve the precision of termination analysis, started by applying existing techniques developed mainly for cost analysis [Flores-Montoya and Hähnle 2014; Gulwani et al. 2009] on some classical examples (such as the one discussed in Section 1.2). The purpose was to simplify the control-flow to allow simpler termination proofs. The transformed programs triggered our interest in checking if it was possible to obtain similar results using general-purpose specialisation techniques, such as partial evaluation. The initial experiments were promising and we decided to explore further. We followed with our initial goal of applying CFR to improve the precision of termination analysis, but given the generality of the new direction, we also became interested in exploring its use in other domains, and also in providing an infrastructure that facilitates using it in such domains.

In this part of our research, we proposed the use of partial evaluation as a general-purpose technique to achieve CFR. Our CFR procedure is developed for TSs, and uses a partial evaluation technique, for Horn clause programs, that is based on specialising programs wrt. a set of supplied properties [Gallagher 2019]. The right choice of properties is a key factor for achieving the desired CFR, and thus we suggested several heuristics for inferring properties automatically. For cost analysis, we also suggested a way to automatically generate properties from corresponding MΦRFs.

We provide an implementation that can be used as a pre-processing step for any static analysis tool that uses TSs. Besides, we have shown how to apply it only on parts of the TS for which we could not obtain a termination proof, and thus improve the performance of iRankFinder. Experimental evaluation in the context of termination analysis demonstrated that our CFR procedure enables termination proofs for many TSs

that could not be handled without CFR by iRankFinder. As evidence, iRankFinder with CFR can now obtain termination proofs more than any other tool that we have compared to. Experiments show that CFR can be useful for non-termination analysis, though, the evidence is not as clear as for termination analysis. For cost analysis, CFR helped `KoAT` to improve the asymptotic upper-bounds of many benchmarks, however, more work is needed in this direction since now it is applied as a pre-processing step, which in turn has a significant performance overhead. We have also evaluated the effect of CFR on the precision of invariant generation, where with our CFR we could prove the required assertions for all challenging examples of Sharma et al. [2011].

Our research on MΦRFs started with the purpose of improving our understanding of MΦRFs, in particular of the problem of deciding whether a given SLC loop has a MΦRF without a given bound on the depth. The outcomes are important insights that shed light on the structure of these ranking functions.

At the heart of our work is an algorithm that seeks MΦRFs, which is based on iteratively eliminating transitions, until eliminating them all or stabilising on a set of transitions that cannot be reduced anymore. In the first case, a MΦRF can be constructed, and, surprisingly, in the second case the stable set of transitions turns to be a recurrent set that witnesses non-termination. This reveals an equivalence between the problems of seeking MΦRFs and seeking recurrent sets of a particular form. This last result has been generalised for TSs as well, and our experiments show that it can be used to prove non-termination of many TSs.

Apart from the relation to seeking recurrent sets, the insights of our work help characterise classes of loops for which there is always a MΦRF, when terminating. We demonstrated this for two classes that have been considered previously. Besides, our insights led to a new representation for SLC loops in which our algorithm has a very simple formalisation that, unlike previous algorithms, yields witnesses for the non-existence of MΦRFs of a given depth. Moreover, this new representation makes some non-trivial observations regarding (bounded) SLC loop straightforward. We believe that this representation can be useful for other related problems.

As a byproduct of our research, we developed an open-source termination analyser called iRankFinder, that implements all techniques developed in this thesis as well as other state-of-the-art techniques. Some of the components of iRankFinder can be used independently, in particular the CFR component that can be used to incorporate CFR in static analysers with little effort. Besides, we developed an open-source toolkit, called EasyInterface, that simplifies the process of building GUIs for research prototype tools, and thus improve the dissemination of the corresponding research. It was used to build a web-interface for iRankFinder.

8.1 Future Work

Our research in this thesis leaves several further research directions, including a good number of new *open questions*, which we hope will trigger the interest of the community.

Control-Flow Refinement

Automatic inference of properties is crucial for obtaining the desired transformations. We have developed some heuristics in this thesis, but further research in this direction is required. A possible direction is to explore properties that are based on different

kinds of ranking functions as discussed in Section 3.3.2. Another possibility is to explore properties that are implicit in a given transitions polyhedron $\mathcal{Q}$, and possibly discover them by studying structural properties of polyhedra. We have started initial research in this direction, where the generators of the set of non-negative functions $\texttt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$ (see Section 4.2.1) are used as properties. This is interesting since any condition implied by $\mathcal{Q}$ is a conic combination of these generators.

In this thesis, we have concentrated on TSs, in a future direction one could apply our CFR techniques for program representations that allow recursion as well. Technically, this would not require much work since the partial evaluation technique of Gallagher [2019] specialise CHCs that include recursion already.

In this thesis, we concentrated on numerical programs, a possible future direction can concentrate on using CFR for program analysis tools where the data is not numerical. Here one should also adapt the partial evaluation techniques to support such specialisations. This seems doable for the partial evaluation of Gallagher [2019] since it is based on using abstract properties like those used in abstract domains of program analysis.

Ranking Functions

In Section 4.2.2, we have seen that our notion of recurrent sets is narrower than the standard one. It is natural to explore the difference between the two kinds of recurrent sets, and in particular in the question if non-terminating SLC loops always have monotonic recurrent sets. Besides, further research can explore the relation between our monotonic recurrent sets and the geometric non-termination argument introduced by Leike and Heizmann [2018], as at a first sight they seem to be equivalent.

Further research is required on understanding properties of Algorithm 3 when it does not terminate. In particular, the properties of the closed convex-set $\mathcal{Q}_\omega$ to which the algorithm converges. Interesting questions to explore in this context are if $\mathcal{Q}_\omega$ is a recurrent set and if the emptiness of $\mathcal{Q}_\omega$ implies termination. These questions are also related to the question if a terminating SLC loop can make executions of any length for the same input. This is true for TSs, but it is an open question of SLC loops. Answering this last question would shed light on many problems related to the termination problem of SLC loops.

Ben-Amram and Genaim [2017] showed that MΦRFs as powerful as the most general definition of LLRFs. It is natural to ask if MΦRFs are the most powerful termination witness, that is based on tuples of linear functions, for SLC loops. We conjuncture that the answer is positive – we have recently proved this for the special case of tuples of length 2.

An obvious future direction is to study the problem of deciding whether a TS has a MΦRF, both from algorithmic and theoretical complexity perspectives. In initial unpublished work, we have proven that the corresponding decision problem is NP-hard for the rational setting, but we could not obtain a further classification. Further exploration of the MΦRF problem for SLC loops is also required since it is not solved for the general case yet, for this one could follow the directions we suggested in Section 4.3.4.

Bibliography

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. URL https://doi.org/10.1007/s10817-010-9174-1.

Elvira Albert, Miquel Bofill, Cristina Borralleras, Enrique Martin-Martin, and Albert Rubio. Resource analysis driven by (conditional) termination proofs. *Theory and Practice of Logic Programming*, 19(5-6):722–739, 2019. URL https://doi.org/10.1017/S1471068419000152.

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, *Static Analysis Symposium, SAS'10*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010. URL https://doi.org/1007/978-3-642-15769-1_8.

Roberto Bagnara and Fred Mesnard. Eventual linear ranking functions. In *Principles and Practice of Declarative Programming, PPDP'13*, pages 229–238. ACM Press, 2013. URL https://doi.org/10.1145/2505879.2505884.

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):3–21, 2008. URL https://doi.org/10.1016/j.scico.2007.08.001.

Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215:47–67, 2012. URL https://doi.org/10.1016/j.ic.2012.03.003.

Alexey Bakhirkin and Nir Piterman. Finding recurrent sets with backward analysis and trace partitioning. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'16*, volume 9636 of *LNCS*, pages 17–35. Springer, 2016. URL https://doi.org/10.1007/978-3-662-49674-9_2.

Alexey Bakhirkin, Josh Berdine, and Nir Piterman. A forward analysis for recurrent sets. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis Symposium, SAS'15*, volume 9291 of *LNCS*, pages 293–311. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48288-9_17.

Amir M. Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In *Principles of programming languages, POPL'13*, pages 51–62. ACM, 2013. URL https://doi.org/10.1145/2480359.2429078.

Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, 2014. URL https://doi.org/10.1145/2629488.

Amir M. Ben-Amram and Samir Genaim. Complexity of Bradley-Manna-Sipma lexicographic ranking functions. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification, CAV'14*, volume 9207 of *LNCS*, pages 304–321. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21668-3_18. see also TR at http://arxiv.org/abs/1504.05018.

Amir M. Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, CAV'17*, volume 10427 of *LNCS*, pages 601–620. Springer, 2017. URL https://doi.org/10.1007/978-3-319-63390-9_32.

Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination through conditional termination. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'17*, volume 10205 of *LNCS*, pages 99–117, 2017. URL https://doi.org/10.1007/978-3-662-54577-5_6.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, pages 491–504. Springer, 2005a. URL https://doi.org/10.1007/11513988_48.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In Martín Abadi and Luca de Alfaro, editors, *Concurrency Theory, CONCUR'05*, volume 3653 of *LNCS*, pages 488–502. Springer, 2005b. URL https://doi.org/10.1007/11539452_37.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *International Colloquium on Automata, Languages and Programming, ICALP'05*, volume 3580 of *LNCS*, pages 1349–1361. Springer, 2005c. URL https://doi.org/10.1007/11523468_109.

Mark Braverman. Termination of integer linear programs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 372–385. Springer, 2006. URL https://doi.org/10.1007/11817963_34.

Marc Brockschmidt and Andrey Rybalchenko. TermComp proposal: Pushdown systems as a model for programs with procedures, 2014. URL https://www.microsoft.com/en-us/research/publication/termcomp-proposal-pushdown-systems-as-a-model-for-programs-with-procedures/.

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexceptions for java bytecode. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software, FoVeOOS'11*, volume 7421 of *LNCS*, pages 123–141. Springer, 2011. URL https://doi.org/10.1007/978-3-642-31762-0_9.

Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification, CAV'13*, volume 8044 of *LNCS*, pages 413–429. Springer, 2013. URL https://doi.org/10.1007/978-3-642-39799-8_28.

Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'16*, volume 9636 of *LNCS*, pages 387–393. Springer, 2016a. URL https://doi.org/10.1007/978-3-662-49674-9_22.

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, TOPLAS'16*, 38(4):13:1–13:50, 2016b. URL https://doi.org/10.1145/2866575.

Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems, TOPLAS'07*, 29(2):10–54, 2007. URL https://doi.org/10.1145/1216374.1216378.

Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. Proving nontermination via safety. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'14*, volume 8413 of *LNCS*, pages 156–171. Springer, 2014. URL https://doi.org/10.1007/978-3-642-54862-8_11.

Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001. URL https://doi.org/10.1007/3-540-45319-9_6.

Michael Colón and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, CAV'02*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002. URL https://doi.org/10.1007/3-540-45657-0_36.

Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. Polyhedral approaches to mixed integer linear programming. In Jünger et al., editor, *50 Years of Integer Programming 1958–2008*, pages 343–386. Springer, 2010. URL https://doi.org/10.1007/978-3-540-68279-0_11.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Programming Language Design and Implementation, PLDI'06*, volume 41:6, pages 415–426. ACM, 2006. URL https://doi.org/10.1145/1133981.1134029.

Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43 (1):93–120, 2013a. URL http://doi.org/10.1007/s10703-013-0186-4.

Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'13*, volume 7795 of *LNCS*, pages 47–61. Springer, 2013b. URL https://doi.org/10.1007/978-3-642-36742-7_4.

Patrick Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 1–24, 2005. URL https://doi.org/10.1007/978-3-540-30579-8_1.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. URL https://doi.org/10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. An abstract interpretation framework for termination. In John Field and Michael Hicks, editors, *Principles of Programming Languages, POPL'12*, pages 245–258. ACM, 2012. URL https://doi.org/10.1145/2103621.2103687.

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, *Principles of Programming Languages, POPL'78*, pages 84–96. ACM Press, 1978. URL https://doi.org/10.1145/512760.512770.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Specialization with constrained generalization for software model checking. In Elvira Albert, editor, *Logic-Based Program Synthesis and Transformation, LOPSTR'12*, volume 7844 of *LNCS*, pages 51–70. Springer, 2012. URL https://doi.org/10.1007/978-3-642-38197-3_5.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. URL https://doi.org/10.1007/978-3-540-78800-3_24.

Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of c programs using compiler intermediate languages. In Manfred Schmidt-Schauß, editor, *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. URL http://drops.dagstuhl.de/opus/volltexte/2011/3123.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992. URL https://doi.org/10.1007/BF01407835.

Fabio Fioravanti, Alberto Pettorossi, Maurizio Proietti, and Valerio Senni. Improving reachability analysis of infinite state systems by specialization. *Fundamenta Informaticae*, 119(3-4):281–300, 2012. URL https://doi.org/10.3233/FI-2012-738.

Antonio Flores-Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Darmstadt University of Technology, Germany, 2017. URL http://tuprints.ulb.tu-darmstadt.de/6746/.

Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Asian Symposium on Programming Languages and Systems, APLAS'14*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014. URL https://doi.org/10.1007/978-3-319-12736-1_15.

Florian Frohn and Jürgen Giesl. Proving non-termination via loop acceleration. In Clark Barrett and Jin Yang, editors, *Formal Methods in Computer Aided Design, FMCAD'19*, pages 221–230. IEEE, 2019. URL https://doi.org/10.23919/FMCAD.2019.8894271.

John P. Gallagher. Polyvariant program specialisation with property-based abstraction. In Alexei Lisitsa and Andrei P. Nemytykh, editors, *Verification and Program Transformation, VPT'19*, volume 299, pages 34–48. Open Publishing Association, 2019. URL https://doi.org/10.4204/eptcs.299.6.

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. URL https://doi.org/10.1007/s10817-016-9388-y.

Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *Programming Language Design and Implementation, PLDI'09*, volume 44, pages 375–385. ACM, 2009. URL https://doi.org/10.1145/1542476.1542518.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In George C. Necula and Philip Wadler, editors, *Principles of Programming Languages, POPL'08*, volume 43, pages 147–158. ACM, 2008. URL https://doi.org/10.1145/1328438.1328459.

William R Harris, Akash Lal, Aditya V Nori, and Sriram K Rajamani. Alternation for termination. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis Symposium, SAS'11*, volume 6337 of *LNCS*, pages 304–319. Springer, 2011. URL https://doi.org/10.1007/978-3-642-15769-1_19.

Radu Iosif, Filip Konečný, and Marius Bozga. Deciding conditional termination. *Logical Methods in Computer Science*, 10(3), 2014. URL http://doi.org/10.2168/LMCS-10(3:8)2014.

Bishoksan Kafle, John P. Gallagher, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. An iterative approach to precondition inference using constrained Horn clauses. *Theory and Practice of Logic Programming*, 18(3-4):553–570, 2018. URL https://doi.org/10.1017/S1471068418000091.

Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, ATVA'11*, pages 492–502. Springer, 2011. URL https://doi.org/10.1007/978-3-642-24372-1_38.

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using Max-SMT. In *Formal Methods in Computer-Aided Design, FMCAD'13*, pages 218–225. IEEE, 2013. URL https://doi.org/10.1109/FMCAD.2013.6679413.

Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using Max-SMT. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification, CAV'14*, volume 8559 of *LNCS*, pages 779–796. Springer, 2014. URL https://doi.org/10.1007/978-3-319-08867-9_52.

Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In David Grove and Steve Blackburn, editors, *Programming Language Design and Implementation, PLDI'15*, pages 489–498. ACM, 2015. URL https://doi.org/10.1145/2737924.2737993.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Chris Hankin and Dave Schmidt, editors, *Principles of Programming Languages, POPL'01*, pages 81–92. ACM, 2001. URL https://doi.org/10.1145/360204.360210.

Jan Leike and Matthias Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1):1–27, 2015. URL http://arxiv.org/abs/1503.00193.

Jan Leike and Matthias Heizmann. Geometric nontermination arguments. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'18*, volume 10806 of *LNCS*, pages 266–283. Springer, 2018. URL https://doi.org/10.1007/978-3-319-89963-3_16.

Michael Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems, TOPLAS'04*, 26(3):413–463, 2004. URL https://doi.org/10.1145/982158.982159.

Michael Leuschel and Thierry Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation, LOPSTR'99*, volume 1817 of *LNCS*, pages 63–82, 2000. URL https://doi.org/10.1007/10720327_5.

Michael Leuschel, Daniel Elphick, Mauricio Varea, Stephen-John Craig, and Marc Fontaine. The Ecce and Logen partial evaluators and their web interfaces. In John Hatcliff and Frank Tip, editors, *Partial Evaluation and semantics-based Program Manipulation, PEPM'06*, pages 88–94. ACM, 2006. URL https://doi.org/10.1145/1111542.1111557.

Yi Li, Guang Zhu, and Yong Feng. The L-depth eventual linear ranking functions for single-path linear constraint loops. In *Theoretical Aspects of Software Engineering, TASE'16*, pages 30–37. IEEE, 2016. URL https://doi.org/10.1109/TASE.2016.8.

Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of Logic programs. In Lee Naish, editor, *International Conference on Logic Programming, ICLP'97*, pages 64–77. MIT Press, 1997. URL https://ieeexplore.ieee.org/document/6279160.

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Principles of Programming Languages, POPL'10*, pages 211–222. ACM, 2010. URL https://doi.org/10.1145/1706299.1706326.

Frédéric Mesnard and Alexander Serebrenik. Recurrence with affine level mappings is P-time decidable for CLP(R). *Theory and Practice of Logic Programming, TPLP'08*, 8(1):111–119, 2008. URL https://doi.org/10.1017/S1471068407003122.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. URL https://doi.org/10.1007/s10990-006-8609-1.

Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for java bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5):83–96, 2009. URL https://doi.org/10.1016/j.entcs.2009.11.016.

Étienne Payet, Fred Mesnard, and Fausto Spoto. Non-termination analysis of java bytecode. *CoRR*, 2014. URL http://arxiv.org/abs/1401.5292.

Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI'04*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004. URL https://doi.org/10.1007/978-3-540-24622-0_20.

Germán Puebla, Manuel Hermenegildo, and John P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In Olivier Danvy, editor, *Partial Evaluation and Semantics-Based Program Manipulation, PEPM'99*, Technical report BRICS-NS-99-1, pages 75–84. University of Aarhus, 1999. URL http://oa.upm.es/14639/.

Germán Puebla, Elvira Albert, and Manuel V. Hermenegildo. Abstract interpretation with specialized definitions. In Kwangkeun Yi, editor, *Static Analysis Symposium, SAS'06*, volume 4134 of *LNCS*, pages 107–126. Springer, 2006. URL https://doi.org/10.1007/11823230_8.

Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification, CAV'11*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011. URL https://dl.acm.org/doi/10.5555/2032305.2032362.

Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes. In Daniel J. Rosenkrantz, editor, *Principles of Database Systems, PoDS'91*, pages 216–226. ACM Press, 1991. URL https://doi.org/10.1145/113413.113433.

Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems, TOPLAS'10*, 32(3), 2010. URL https://doi.org/10.1145/1709093.1709095.

Ashish Tiwari. Termination of linear programs. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004. URL https://doi.org/10.1007/978-3-540-27813-9_6.

Alan M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69. University Mathematical Laboratory, Cambridge University, 1949. URL http://www.turingarchive.org/browse.php/B/8.

Caterina Urban. The abstract domain of segmented ranking functions. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis Symposium, SAS'13*, volume 7935 of *LNCS*, pages 43–62. Springer, 2013. URL https://doi.org/10.1007/978-3-642-38856-9_5.

Caterina Urban and Antoine Miné. An abstract domain to infer ordinal-valued ranking functions. In Zhong Shao, editor, *European Symposium on Programming, ESOP'14*, volume 8410 of *LNCS*, pages 412–431. Springer, 2014. URL https://doi.org/10.1007/978-3-642-54833-8_22.

Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, TAP'08*, volume 4966 of *LNCS*, pages 154–170. Springer, 2008. URL https://doi.org/10.1007/978-3-540-79124-9_11.

Acronyms

EIOL *EasyInterface Output Language*. ix, 79, 82–84, 86–89

TPDB *Termination Problem Data Base*. 91

CFG *Control-Flow Graph*. 1, 2, 11, 12, 15, 26, 50, 59, 103

CFR *Control-Flow Refinement*. vii, viii, xi, 4–7, 25, 26, 30–46, 74–76, 91–99, 103–107

CHC *Constrained Horn Clause*. ix, 26–28, 30–32, 36, 76, 107

CRS *Cost Relations*. 41, 96

GUIs *Graphical User Interfaces*. 7, 79

LLRF *Lexicographic Linear Ranking Function*. ix, xi, 3–5, 18–21, 25, 33, 36, 45, 47, 49, 93, 94, 101, 102, 105, 107

LRF *Linear Ranking Function*. xi, 2–4, 17, 19, 21, 25, 28, 33, 36, 37, 39, 41, 42, 44, 47, 49, 64, 65, 67, 69, 70, 76, 77, 92, 94, 101, 102, 105

MΦRF *Multi-Phase Linear Ranking Function*. viii, ix, 3–7, 21, 22, 45–55, 57, 58, 60, 62, 64–69, 77, 102, 105–107

PPL *Parma Polyhedra Library*. 73

SCC *Strongly Connected Component*. 15–22, 32, 33, 35–37, 39–42, 44, 47, 56, 60, 94

SLC *Single-path Linear-Constraint*. viii, ix, 1–3, 5–7, 13, 14, 45, 47–50, 53, 55–62, 64–69, 99, 102, 103, 105–107

TS *Transition System*. ix, 1–3, 5–7, 11–14, 16, 17, 19–23, 25–28, 30–49, 56, 59, 60, 65, 73–77, 79, 91, 95–97, 99, 102, 105–107