

Automated Complexity Analysis of Rewrite Systems

Der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH
Aachen University vorgelegte Dissertation zur Erlangung des akademischen
Grades eines Doktors der Naturwissenschaften von

Master of Science
Florian Frohn
aus Aachen

Abstract

Besides functional correctness, one of the most important prerequisites for the success of a piece of software is efficiency: The desired results need to be computed not only correctly, but also in time. Thus, analyzing the runtime complexity of software is indispensable in practice.

On the other hand, analyzing the complexity of large programs manually is infeasible. Hence, automated complexity analysis techniques are needed. In this way, performance pitfalls can be highlighted automatically like other bugs which can nowadays be found by compilers or static analyzers.

However, statically analyzing the complexity of real-world programs poses several problems. For example, most programming languages lack formal semantics. Moreover, different programming languages offer different features, so static analyses for one language do not necessarily apply to others. A common solution for these problems is to transform programs into low-level formalisms like *term* or *integer rewrite systems* that can be analyzed without worrying about language-specific peculiarities.

Unfortunately, state-of-the-art complexity analysis techniques for rewrite systems have several limitations. Firstly, most of them are restricted to the inference of upper bounds on the worst-case complexity. Moreover, existing complexity analyzers only reach their full potential if an eager evaluation strategy is fixed, which is sometimes undesired in practice. Finally, many techniques for integer rewriting just support tail recursion.

This thesis partially overcomes these limitations. To this end, the first techniques for the inference of lower bounds on the worst-case complexity are introduced. One important use case of such lower bounds is to witness denial-of-service vulnerabilities (whose absence can be proven via upper bounds), which arise if the runtime of a program exceeds expectations.

Regarding upper bounds, this thesis shows how complexity analysis techniques for term rewriting which require an eager evaluation strategy can also be used without assuming eager evaluation. Similarly, it shows how complexity analysis techniques for integer rewriting which are restricted to tail recursion can also be used to analyze non-tail-recursive systems.

Consequently, complexity-preserving transformations to rewrite systems can now be used for more applications – like the detection of denial-of-service vulnerabilities – and richer classes of programs. This is true for both programs operating on integers as well as programs operating on tree-shaped data, which can naturally be transformed to integer and term rewrite systems, respectively. Hence, this thesis covers a large subset of the features provided by real-world programming languages.

The presented techniques were implemented in the tools **AProVE** and **LoAT**, which can analyze the complexity of rewrite systems fully automatically. Extensive experiments demonstrate the feasibility of the presented techniques in practice.

Abstract

Neben Korrektheit ist Effizienz eine der wichtigsten Voraussetzungen für den Erfolg von Software: Das gesuchte Ergebnis muss nicht nur korrekt, sondern auch rechtzeitig berechnet werden. Deshalb ist es in der Praxis unverzichtbar, die Laufzeitkomplexität von Software zu analysieren.

Allerdings ist es unpraktikabel, die Komplexität von Programmen manuell zu analysieren. Folglich werden automatische Techniken benötigt. So können Performance-Probleme genauso automatisiert gefunden werden wie andere Fehler, die von Tools zur statischen Analyse erkannt werden.

Jedoch bringt es etliche Probleme mit sich, die Komplexität realistischer Programme statisch zu analysieren. Zum Beispiel haben die meisten Programmiersprachen keine formale Semantik. Zudem unterscheiden sich verschiedene Sprachen mitunter stark, sodass statische Analysen für eine Sprache oft nicht auf andere übertragbar sind. Eine verbreitete Lösung ist, Programme in *Reduktionssysteme*, insbesondere *Term-* oder *Integer-Ersetzungssysteme*, zu übersetzen und deren Komplexität zu analysieren.

Leider haben aktuelle Techniken zur Analyse von Reduktionssystemen zahlreiche Einschränkungen. Zunächst sind die meisten Techniken auf die Inferenz oberer Schranken für die Worst-Case-Komplexität beschränkt. Außerdem können aktuelle Tools ihre Stärken nur unter der Annahme einer strikten Auswertungsstrategie voll ausspielen. Schließlich unterstützen viele Techniken für Integer-Ersetzung ausschließlich Endrekursion.

Diese Arbeit überwindet die genannten Limitierungen teilweise. Sie stellt unter anderem die ersten Techniken zur Inferenz unterer Schranken für die Worst-Case-Komplexität vor. Eine wichtige Anwendung für solche Schranken ist das Finden von Denial-of-Service-Angriffsmöglichkeiten, die entstehen, wenn die Laufzeit eines Programms größer ist als erwartet.

Bezüglich oberer Schranken zeigt diese Arbeit, wie Techniken für Termersetzung, die eine strikte Auswertungsstrategie erfordern, genutzt werden können, ohne eine bestimmte Strategie vorauszusetzen. Ebenso zeigt sie, wie Techniken für Integer-Ersetzung, die auf Endrekursion eingeschränkt sind, auch für nicht endrekursive Systeme genutzt werden können.

Somit können Übersetzungen in Reduktionssysteme nun für ausdrucksstärkere Klassen von Programmen und neue Anwendungsfälle eingesetzt werden, z.B. die Erkennung von Denial-of-Service-Angriffsmöglichkeiten. Dies gilt insbesondere für Programme, die auf Integern oder baumförmigen Datenstrukturen arbeiten, da sich diese auf natürliche Art und Weise in Integer- bzw. Termersetzung übersetzen lassen. Folglich deckt diese Arbeit große Teile der Features realistischer Programmiersprachen ab.

Die präsentierten Techniken wurden in den Tools AProVE und LoAT implementiert, die die Komplexität von Reduktionssystemen vollautomatisch analysieren können. Ausführliche Experimente demonstrieren ihre praktische Anwendbarkeit.

Acknowledgements

First I want to thank Jürgen Giesl for giving me the opportunity to be a PhD student in his research group. The close collaboration with him was not only extraordinarily instructive, but also very productive. He gave me the opportunity to explore every idea that came to my mind, the good ones as well as the bad ones, which made the last years as interesting as they were. I am also very grateful that Samir Genaim, a remarkable researcher from our community, agreed to be my second supervisor.

I want to thank the current and former members of our research group for many interesting discussions and collaborations throughout the last years. In particular, I want to thank Marc Brockschmidt, Fabian Emmes, Carsten Otto, and Thomas Ströder for introducing me into the work of our research group; Carsten Fuhs for filling the gaps in AProVE’s documentation by writing the most detailed emails I have ever read; Cornelius Aschermann for many interesting hours in front of the whiteboard; and Jera Hensel and David Korzeniewski for being pleasant office mates. Special thanks also go to Marcel Hark, not only for many fruitful discussions about complexity analysis, but also for proofreading parts of this thesis.

Regarding the members of the MOVES and Theory of Hybrid Systems groups, I want to apologize for leaving with a considerable cake-debt. I always enjoyed sharing cake, coffee, and interesting discussions with you. In particular, I want to thank Christoph Matheja and Benjamin Kaminski for patiently answering all my (not necessarily clever) questions about probabilistic programs when we once tried to transfer some ideas from my work to a probabilistic setting.

I also want to thank the students who worked with me as student assistants or when writing their theses. Most remarkably, without Stefan Dollase, one would find “TODO: **refactor**” much more often in the code of AProVE’s Eclipse Plugin, without Matthias Naaf, LoAT would not exist, and without Thies Strothmann, there would be many more bugs in AProVE’s Java frontend.

Thanks to all my friends who spent countless hours in Aachen’s climbing and bouldering gyms as well as Europe’s climbing areas with me throughout the last years. Big parts of this thesis would not have been possible without taking mental rests during climbing sessions with you.

Special thanks go to my parents for supporting me my whole life and for always believing in me.

Last but not least, I want to thank my girlfriend Lena for supporting me in so many ways. I am sorry for not listening, but thinking about research far too often and I am glad that you pardon me every time.

Contents

| | | |
|-----------|---|-----------|
| I | Preface | 1 |
| 1 | Introduction | 3 |
| 1.1 | Structure of the Thesis | 6 |
| 1.2 | Rewrite Systems and their Relation to Programming Languages | 7 |
| 1.3 | Contributions and Publications | 12 |
| 1.4 | Related Work | 21 |
| 2 | Preliminaries | 25 |
| 2.1 | Terms and Related Concepts | 26 |
| 2.2 | Complexity Problems | 30 |
| II | Complexity Analysis of Integer Rewrite Systems | 37 |
| 3 | Introduction | 39 |
| 4 | Lower Bounds for Integer Transition Systems | 41 |
| 4.1 | Program Model | 43 |
| 4.2 | Estimating the Number of Iterations | 49 |
| 4.3 | Simplifying ITSs | 53 |
| 4.4 | Non-Linear ITSs | 60 |
| 4.5 | Asymptotic Lower Bounds | 68 |
| 4.6 | Solving Limit Problems via SMT | 80 |
| 4.7 | Related Work | 84 |
| 4.8 | Experiments | 86 |
| 4.9 | Conclusion and Future Work | 89 |
| 5 | Upper Bounds for Recursive Natural Transition Systems | 91 |
| 5.1 | Program Model | 92 |
| 5.2 | Size Bounds as Runtime Bounds | 96 |
| 5.3 | Complexity Bounds for RNTSs | 97 |
| 5.4 | Related Work | 117 |
| 5.5 | Experiments | 119 |
| 5.6 | Conclusion and Future Work | 121 |

| | | |
|------------|---|------------|
| III | Complexity Analysis of Term Rewrite Systems | 123 |
| 6 | Introduction | 125 |
| 7 | Preliminaries | 127 |
| 8 | Lower Bounds for Term Rewriting by Loop Detection | 133 |
| 8.1 | Loop Detection for Linear Bounds | 134 |
| 8.2 | Loop Detection for Exponential Bounds | 139 |
| 8.3 | Incompleteness of Loop Detection | 143 |
| 8.4 | Innermost Decreasing Loops | 150 |
| 8.5 | Related Work | 153 |
| 8.6 | Conclusion and Future Work | 154 |
| 9 | Lower Bounds for Term Rewriting by Induction | 157 |
| 9.1 | From Term Rewriting to Rewrite Lemmas | 159 |
| 9.2 | Speculating Conjectures | 166 |
| 9.3 | Proving Conjectures | 172 |
| 9.4 | Inferring Bounds for Valid Conjectures | 174 |
| 9.5 | Inferring Bounds for TRSs | 183 |
| 9.6 | Indefinite Rewrite Lemmas | 186 |
| 9.7 | Argument Filtering | 189 |
| 9.8 | Proving Innermost Conjectures | 194 |
| 9.9 | Induction Technique vs. Loop Detection | 197 |
| 9.10 | Related Work | 199 |
| 9.11 | Conclusion and Future Work | 200 |
| 10 | Deciding Constant Upper Bounds | 203 |
| 10.1 | Constant Upper Bounds via Narrowing | 204 |
| 10.2 | Constant Bounds for Innermost Rewriting | 211 |
| 10.3 | Related Work | 213 |
| 10.4 | Conclusion and Future Work | 215 |
| 11 | Strategy Switching – From Full to Innermost Rewriting and Vice Versa | 217 |
| 11.1 | Non-Dup Generalized Innermost Rewriting | 219 |
| 11.2 | Approximating Sparseness | 226 |
| 11.3 | Related Work | 237 |
| 11.4 | Conclusion and Future Work | 239 |
| 12 | Experiments | 241 |
| 12.1 | Full Rewriting | 242 |
| 12.2 | Innermost Rewriting | 248 |

| | | |
|-----------|---|------------|
| IV | Postface | 255 |
| 13 | Conclusion and Outlook | 257 |
| A | Missing Proofs | 259 |
| B | Examples Mentioned in Experimental Evaluations | 265 |
| B.1 | Evaluation of Chapter 5..... | 266 |
| B.2 | Evaluation of Part III | 267 |
| | Bibliography | 271 |

Part I

Preface

Introduction

Analyzing the runtime complexity of software is highly relevant in practice: Even software that complies with its (functional) specification can be practically unusable if it does not get its job done within a reasonable amount of time. If a piece of software performs reasonably well in everyday use, it may still require an unacceptable amount of computation time when fed with malicious, unexpected inputs, resulting in denial-of-service vulnerabilities. Note that these observations also carry over to other kinds of resources like memory or bandwidth.

On the other hand, analyzing the complexity of software manually is a cumbersome and complex task, even for small programs. For realistically sized programs, a manual analysis is essentially infeasible. Thus, *automated* techniques are required to take the burden of analyzing a program's complexity from its developers.

Recent advances in program analysis yield efficient methods to find *upper*

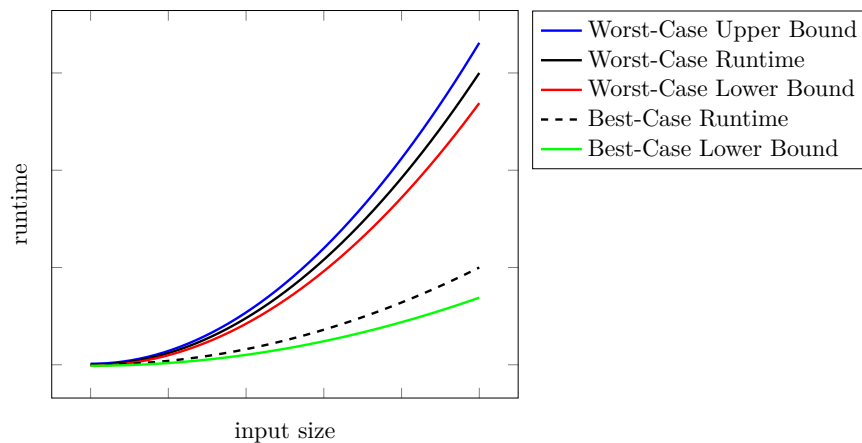


Figure 1.1: Different Kinds of Runtime Bounds

CHAPTER 1. INTRODUCTION

bounds on the complexity of various kinds of programs automatically (e.g., [3, 6, 27, 48, 50, 51, 82, 114]). Here, one usually considers “worst-case complexity”, i.e., for any initial state s , one analyzes the length of the longest execution starting from s . But in many cases, in addition to upper bounds, it is also important to find *lower* bounds for this notion of complexity. To clarify our notion of lower bounds, consider Figure 1.1. Note that the complexity of a program is usually expressed in terms of a function which maps the size of the input (which corresponds to the horizontal axis in Figure 1.1) to the runtime of the program (corresponding to the vertical axis in Figure 1.1) for an input of that size. Thereby, the size of the input is defined according to some *size measure* or *norm* which maps the input of a program to the natural numbers. For example, lists might be measured by their length or trees might be measured by their height or their number of nodes. However, as most common size measures are not injective, there are usually *several* inputs of the same size. Hence, instead of one function, one obtains two functions: One for the *worst-case* complexity (solid black line in Figure 1.1), which maps a number n to the *maximal* runtime for all inputs of size n and one for the *best-case* complexity (dashed black line in Figure 1.1), which maps a number n to the *minimal* runtime for all inputs of size n .

As mentioned before, most existing automated complexity analysis techniques focus on the inference of *worst-case upper bounds* (blue line in Figure 1.1). Apart from that, some techniques are concerned with the inference of *best-case lower bounds* (green line in Figure 1.1), e.g., [3, 48].¹ While the relevance of worst-case upper bounds is widely accepted, best-case lower bounds also have interesting applications. For example, they can guarantee that a task will take long enough to compensate the overhead of running it remotely [3, 40]. In contrast to worst-case upper and best-case lower bounds, *worst-case lower bounds* have been neglected by the research community so far. In this thesis, we exclusively focus on worst-case complexity, i.e., besides worst-case upper bounds, we are also interested in the inference of worst-case lower bounds (red line in Figure 1.1). In the following, we just use the terms “upper” resp. “lower bound” to refer to worst-case upper and lower bounds, respectively.

To see the importance of lower bounds, note that by combining them with an analysis for upper bounds they can be used to infer *tight* complexity bounds. Lower bounds also have important applications in security analysis, e.g., to detect possible denial-of-service vulnerabilities. In contrast, upper bounds are useful to prove the absence of such vulnerabilities. Thus, techniques to infer lower and upper bounds on the worst-case complexity of programs complement each other.

However, statically analyzing (the complexity or any other property of) programs written in real-world programming languages like `Java` or `C` directly is not ideal for various reasons.

¹I am not aware of any techniques for the inference of *best-case upper bounds*, which is the reason why they are missing in Figure 1.1.

- Programming languages offer various features which are convenient for programmers, but complicate static analyses. For example, virtual method calls are crucial for object-oriented programming. In contrast, for static analyses it is desirable to know the actual implementation of the invoked method.
- Most programming languages do not have formal semantics. Thus, static analyses have to “fill the gaps” in the informal specification by “best guesses”.
- The semantics of programming languages change over time.
- Different programming languages offer different features, so static analyses for one language do not apply to other languages.

For these – and various other – reasons, it is favorable to decouple language specific features and the (underspecified and changing) semantics of programming languages from static analyses whenever possible. While intermediate compiler representations of programming languages like JBC or LLVM are a first step towards this goal, they are still rather designed for practice than for verification. Thus, it is advantageous to further transform such intermediate representations to formally specified models of computation like *rewrite systems*. Two important flavors of rewrite systems are *term rewrite systems* and *integer rewrite systems*. While the former correspond to first-order functional programs, the latter correspond to integer programs, i.e., programs where all variables are of type `int`.

Transformations from, e.g., Haskell [63] to term rewrite systems and from C [119] to integer rewrite systems have been used for termination analysis for many years. In contrast to the compilation from, for example, C to LLVM, such transformations are usually not designed to preserve the semantics of the original program. Instead, their intention is to preserve the analyzed property like, for instance, the termination behavior or the complexity of the original program. Consequently, existing transformations from, e.g., the area of termination analysis, are not necessarily suitable for complexity analysis. Developing complexity-preserving transformations from programming languages to rewrite systems is an active field of research [51, 76, 101].

Clearly, this research has to be complemented by powerful complexity analysis techniques for the resulting rewrite systems. In this thesis, we present techniques to analyze the worst-case complexity of the two previously mentioned flavors of rewriting (integer and term rewriting). Here, we consider lower as well as upper bounds.

1.1 Structure of the Thesis

In Section 1.2, we compare rewrite systems with programming languages to clarify to what extent complexity analysis techniques for rewrite systems can also be used to analyze the complexity of real-world programs. Thereby, we highlight open problems regarding automated complexity analysis of rewrite systems and we mention which of these problems are tackled in this thesis.

Section 1.3 gives a high-level overview of the contributions of this thesis. To this end, it informally states the underlying ideas of the complexity analysis techniques presented afterwards in Part II and Part III. Moreover, it clarifies the significance of the presented techniques by mentioning use cases and unique features in comparison to related work. Finally, it provides information about all prior publications of the author and explains their relation to this thesis.

Section 1.4 briefly discusses the state-of-the-art. Moreover, it mentions related fields of research and their relevance for this thesis. A more detailed discussion of relevant related work can be found after the presentation of the respective complexity analysis techniques, i.e., in Sections 4.7, 5.4, 8.5, 9.10, 10.3, and 11.3. Chapter 2 introduces the required preliminaries, in particular a novel framework for the formalization of complexity analysis techniques.

After a short introduction (Chapter 3), Part II presents two complexity analysis techniques for programs operating on integers in Chapter 4 and Chapter 5. Since the kind of programs considered in these chapters differ, each of them starts with the definition of the respective program model (Section 4.1 and Section 5.1) and ends with a separated experimental evaluation (Section 4.8 and Section 5.5).

Part III entirely focuses on complexity analysis techniques for term rewriting. After an introduction (Chapter 6) and the presentation of the program model and other preliminaries (Chapter 7), four different complexity analysis techniques for term rewriting are introduced. The first two techniques focus on lower bounds (Chapter 8 and Chapter 9), the third technique considers upper bounds (Chapter 10), and the last technique has applications for both lower and upper bounds (Chapter 11). Since all of these techniques consider the same program model, they are evaluated together in Chapter 12.

Finally, we conclude in Chapter 13, where we also give an outlook on possible future developments. Thereby, we focus on extensions of the presented techniques to richer classes of rewrite systems, whereas possible improvements w.r.t. the same class of rewrite systems are discussed in the respective chapters in Sections 4.9, 5.6, 8.6, 9.11, 10.4, and 11.4.

In Appendix A, we provide those theorems, lemmas, and proofs which are omitted in the main part of this thesis. In Appendix B, we provide the names of those examples which are mentioned in our experimental evaluation as their results are particularly interesting in some sense.

1.2 Rewrite Systems and their Relation to Programming Languages

In this section, we give an overview over existing flavors of rewriting and compare them with real-world programming languages. In particular, we mention the restrictions of the different kinds of rewriting and of existing techniques to analyze them. Moreover, we point out which of these restrictions are tackled in this thesis.

Table 1.1 shows an overview of various flavors of rewrite systems, including all kinds of rewrite systems which are considered in this thesis. To clarify their relevance w.r.t. the analysis of real-world programming languages, their features are compared with object-oriented programming languages (OOP) like **Java** and functional programming languages (FP) like **Haskell**.

All rewrite systems in Table 1.1 are special cases of *logically constrained term rewrite systems* (LCTRSs) [59] which extend classical term rewrite systems with arbitrary built-in data types like integers (\mathbb{Z}) or floating point numbers (\mathbb{F}). Thus, LCTRSs allow, e.g., rules like the following, which (together with a corresponding implementation of `fib`) computes a list of Fibonacci numbers.

$$\text{fiblist}(n, r) \rightarrow \text{fiblist}(n - 1, \text{cons}(\text{fib}(n), r)) \quad [n > 0] \quad (1.1)$$

It uses arithmetic (“ $n - 1$ ” on the right-hand side), nested procedure calls (`fib` below `fiblist` on the right-hand side), data structures (as `cons` on the right-hand side is used to represent lists), and logical constraints to restrict the control flow (i.e., the condition “ $n > 0$ ” means that the rule can only be applied if n is instantiated with a positive number). While (1.1) just uses integer arithmetic, as mentioned above LCTRSs also allow other built-in data types.

Thus, LCTRSs offer most features of real-world programming language, with

| | arithmetic | | | data struct. | | procedures | | |
|-------|------------|--------------|--------------|--------------|---|------------|----------|-----------------|
| | N | \mathbb{Z} | \mathbb{F} | 🌲 | ◇ | void | non-void | non-eager eval. |
| LITS | X | X | | | | | | |
| ITS | X | X | | | | X | | |
| RNTS | X | | | | | X | X | |
| IRS | X | X | | | | X | X | X |
| TRS | | | | X | | X | X | X |
| ITRS | X | X | | X | | X | X | X |
| LCTRS | X | X | X | X | | X | X | X |
| OOP | X | X | X | X | X | X | X | |
| FP | X | X | X | X | X | X | X | X |

Table 1.1: Rewrite systems compared with programming languages

CHAPTER 1. INTRODUCTION

the exception of non-tree-shaped data structures (\diamond). Examples for such data structures are graphs in object-oriented languages or cyclic lists in functional languages like **Haskell**. Such data structures are considered in *graph rewriting* [16, 30, 31], which is beyond the scope of this thesis. Like in classical term rewriting, tree-shaped data structures (\blacktriangle) can be represented by terms.

While the tool **Ctrl** [94] can analyze termination (and some other properties) of LCTRSs with various built-in data types, there are no complexity analysis tools for LCTRSs. The same holds for *integer term rewrite systems* (ITRSs) [58], which predate LCTRSs and can be seen as LCTRSs with the single built-in data type \mathbb{Z} (i.e., (1.1) is also a valid ITRS rule). Termination of ITRSs can be analyzed by **AProVE** [62].

LCTRSs without any built-in data types are classical *term rewrite systems* (TRSs). Thus, TRSs are suitable to implement algorithms operating on tree-shaped data structures like lists or trees. For example, the TRS rule

$$\text{traverse}(\text{tree}(l, r)) \rightarrow \text{tree}(\text{traverse}(l), \text{traverse}(r))$$

implements a tree traversal algorithm. In contrast, (1.1) is not a valid TRS rule because of its use of arithmetic and logical constraints. Due to their support for tree-shaped data structures, real-world programs operating on such data structures can be transformed to TRSs in a natural way. Moreover, built-in data types like integers can be simulated in TRSs by encoding them as terms. For example, a natural number n can be represented by the term $\text{succ}^n(\text{zero})$ (i.e., the n -fold application of a “successor” function symbol succ to a function symbol representing zero). Then (1.1) becomes

$$\text{fiblist}(\text{succ}(n), r) \rightarrow \text{fiblist}(n, \text{cons}(\text{fib}(n), r)).$$

Such an approach is used by the translation from **Haskell** to term rewriting in [63]. However, as discussed in [58], a significant drawback of such an encoding is the loss of domain specific knowledge.

A novel technique to deduce upper bounds for TRSs is introduced in Chapter 10. In contrast to the many previous complexity analysis techniques for TRSs, it focuses on the inference of *constant* upper bounds and, in particular, it serves as a semi-decision procedure for this purpose.

While the inference of upper bounds on the complexity of TRSs has been widely studied, so far there were no techniques to infer lower bounds. This gap is closed in Chapter 8 and Chapter 9.

As in the case of real-world programming languages, an *evaluation strategy* needs to be fixed in order to evaluate LCTRSs, ITRSs, or TRSs.² For example, **Java** is evaluated eagerly, i.e., all arguments are fully evaluated before a method is invoked. In contrast, **Haskell** uses *lazy evaluation*, i.e., the arguments of a function f are not evaluated until their value is required to continue the evalua-

²While [58] just considers eagerly evaluated ITRSs, one could define other evaluation strategies as in [59].

1.2. REWRITING VS. PROGRAMMING LANGUAGES

tion of f . While eager evaluation is standard for imperative and object-oriented programming languages, the evaluation strategies of functional programming languages are less uniform. For example, MAUDE's evaluation strategy is highly customizable and OCaml, a widely used member of the ML language family, uses eager evaluation by default, but also supports lazy evaluation. Existing complexity analysis techniques for term rewriting support eager evaluation (which is conventionally called *innermost rewriting*) and *full rewriting* where the evaluation strategy is completely unrestricted. Thus, the latter also covers mixtures of eager and lazy evaluation which may occur in languages like MAUDE and OCaml. Hence, it is not surprising that, e.g., MAUDE programs can be transformed to term rewrite systems whose full runtime complexity serves as an upper bound on the complexity of the original program [117]. However, the power of existing complexity analysis tools for full rewriting lags behind their impressive results for innermost rewriting. The technique presented in Chapter 11 significantly reduces this discrepancy.

Integer rewrite systems (IRSs) are ITRSs without tree-shaped data structures. As an example, the rule

$$\text{ack}(n, m) \rightarrow \text{ack}(n - 1, \text{ack}(n, m - 1)) \ [n > 0 \wedge m > 0] \quad (1.2)$$

is a valid IRS rule, as it neither uses data structures like lists in (1.1) nor any other built-in data types except integers. Hence, IRSs correspond to programs where all variables range over the integers and thus they are a suitable target language for translations from real-world programs operating on integers. Moreover, programs operating on both integers and data structures can be transformed to IRSs by abstracting data structures to natural numbers using a suitable *size abstraction*.

While there are some complexity analysis tools for (unrestricted) IRSs (or equivalent formalisms) like CoFloCo [48, 50] and PUBS [3], many tools can only analyze subclasses of IRSs with limited support for procedures: *Integer transition systems* (ITSs) essentially only support procedures without return values (i.e., with return type **void**) and *linear integer transition systems* (LITSs) do not support procedures at all. Thus, (1.2) is not a valid ITS rule, as the result of the inner **ack** call is passed to the outer **ack** call as an argument on the right-hand side, i.e., the return type of **ack** is **int**. In contrast,

$$\text{fib}(n) \rightarrow \text{fib}(n - 1) + \text{fib}(n - 2) \ [n > 2] \quad (1.3)$$

is a valid ITS rule, as the result of **fib** is not passed to any other function. Hence, here the runtime complexity is independent of **fib**'s result and thus corresponds to the complexity of the program

CHAPTER 1. INTRODUCTION

```

void fib (n) {
    if (n > 2) {
        fib (n-1);
        fib (n-2);
    }
}

```

where the only function `fib` has the return type **void**. However, the non-linear recursion in (1.3) requires procedure calls. Hence, (1.3) is not a valid LITS rule. In contrast, the rule

$$\text{fac}(n, r) \rightarrow \text{fac}(n-1, r \cdot n) \ [n > 0]$$

is a valid LITS rule, as it corresponds to the loop

```

while (n > 0) {
    r = r * n;
    n = n - 1;
}

```

i.e., it corresponds to a program without procedure calls.

In Chapter 5, we show how complexity analysis tools for ITSs can also be used to analyze programs with non-**void** procedures, such that all existing complexity analysis tools for ITSs can also be used to analyze non-tail recursive programs (note that tail recursive procedures can be inlined, i.e., support for procedures is mainly required to handle non-tail recursion). Thereby, we restrict ourselves to eagerly evaluated programs with arithmetic on natural numbers (i.e., IRSs where all variables implicitly range over \mathbb{N}), resulting in the notion of *recursive natural transition systems* (RNTSs). Extensions to non-eager evaluation and full integer arithmetic are left to future work.

As in the case of TRSs, all existing techniques to analyze the worst-case complexity of IRSs are restricted to upper bounds. In Chapter 4, we partially close this gap by introducing the first technique for the inference of lower bounds for ITSs.

To summarize, in the long run powerful complexity analysis techniques for LCTRSs are desirable. The reason is that LCTRSs cover most features of real-world programming languages. Thus, they allow for natural complexity-preserving transformations from real-world programs. However, we are not yet there: The state of the art is essentially restricted to upper bounds for rewriting with either tree-shaped data or integers, in the latter case often without support for non-tail recursion. Moreover, the support for non-eager evaluation is limited. The goal of this thesis is to advance towards powerful complexity analysis techniques for LCTRSs. To this end, we complement existing techniques for upper bounds with techniques for the inference of lower bounds. Moreover, we

1.2. REWRITING VS. PROGRAMMING LANGUAGES

partially overcome existing restrictions w.r.t. non-tail recursion and non-eager evaluation.

1.3 Contributions and Publications

In this section, I summarize the major contributions of my thesis. Moreover, I clarify which parts of the thesis were previously published, give an overview on my publications, and explain how I contributed to these publications.

The first considerable contribution is the (yet unpublished) framework for the formalization of complexity analysis techniques from Chapter 2. Despite its simplicity, it is general enough to cover all results presented in this thesis. Thus, it is an important building block for a homogeneous presentation of the complexity analysis techniques from Part II and Part III.

1.3.1 Contributions of Chapter 4 and [56]

Chapter 4 introduces the first automatic technique to infer lower bounds on the worst-case complexity of integer programs. As argued in the beginning of this chapter, I believe that worst-case lower bounds nicely complement worst-case upper bounds and have important applications that justify the relevance of Chapter 4, especially in the context of cybersecurity.

The core of the technique presented in Chapter 4 is an *under-approximating program simplification framework* for linear (Section 4.3) and non-linear (Section 4.4) ITSs. It eliminates loops using an *acceleration* technique which estimates the effect of executing loops multiple times. To this end, it uses *recurrence solving* and *metering functions* (Section 4.2), a novel adaption of classical *ranking functions*. A similar technique is also used to eliminate recursion. Moreover, our program simplification framework eliminates straight-line code via *chaining*. In this way, one eventually obtains a *simplified program* with trivial control flow. Lower bounds on the complexity of such simplified programs can then be computed using the calculus to solve *limit problems* presented in Section 4.5. To improve performance and scalability, limit problems can also be encoded into SMT formulas in many cases, cf. Section 4.6.

A preliminary version of Chapter 4, which was restricted to linear ITSs and did not contain the SMT encoding from Section 4.6, has been published in [56]. I designed the overall program simplification framework from [56, Section 4], that is, the combination of loop acceleration with chaining which allows to transform complex into simplified programs (cf. Section 4.3). Moreover, I significantly contributed to the calculus from [56, Section 5] to obtain asymptotic bounds for simplified programs by generalizing initial ideas that resulted from discussions among the authors. Finally, I coordinated the implementation of the presented technique in the tool LoAT, wrote the first version of the paper, and was heavily involved in the process of polishing it until it was ready for publication.

After the publication of [56], I applied our implementation LoAT to ITSs resulting from an automated transformation of Java programs [51] within the CAGE project [110]. CAGE was a joint project of our research group at RWTH

1.3. CONTRIBUTIONS AND PUBLICATIONS

Aachen University with Draper³ and the University of Innsbruck. The goal of this project was to develop tools and techniques to find and prove the absence of denial of service and side channel vulnerabilities in large **Java** programs. Here, the calculus from Section 4.5 turned out to be a bottleneck for automatically generated ITSs with many conditions which are redundant or do not influence the complexity of the ITS. To solve this problem, I designed and implemented the novel SMT encoding from Section 4.6.

The (not yet implemented) handling of non-linear ITSs (cf. Section 4.4) is also completely new. However, non-linear ITSs do not fit into the formalism from [56], where programs are essentially represented as graphs. Thus, to achieve a homogeneous presentation, I rephrased all techniques and results from [56] using a rule-based formalism, i.e., another contribution of Chapter 4 is the use of a more general and more expressive formalism in comparison to [56].

1.3.2 Contributions of Chapter 5 and [103]

The work of Chapter 5 originated from the observation that existing complexity analysis techniques for integer programs are significantly more modular than those for term rewriting. For instance, [27] decomposes integer programs into small pieces which are analyzed independently. Afterwards, *size bounds*, a specific form of (often super-linear) invariants, are used to compose the obtained sub-results. In this way, [27] can successfully analyze programs with super-linear complexity without generating super-linear ranking functions.

Ranking functions allow to over-approximate how often a certain program instruction is executed in a program run. Thus, even if an instruction can be executed quadratically often in a run of the *whole* program, it may only be executed linearly often if one just considers a *part* of the program, i.e., linear ranking functions are often sufficient in a modular setting as in [27]. Generating super-linear ranking functions is a hard synthesis problem, whereas linear ranking functions can often be found efficiently using state-of-the-art SMT solvers. Thus, a decomposition of the program as in [27] greatly increases the applicability of the resulting complexity analysis technique.

In contrast, techniques for term rewriting like [15, 105] often have to generate super-linear ranking functions in order to analyze programs with super-linear complexity. The reason is that they do not rely on a notion like size bounds which allows to estimate the effects of program parts. Thus, linear ranking functions are rarely sufficient for the inference of super-linear bounds. To the best of my knowledge, the only complexity analysis technique for term rewriting which allows the inference of super-linear bounds by deducing linear bounds for program parts is the *Dependency Graph Decomposition* from [15]. However, in contrast to techniques like [27], the Dependency Graph Decomposition does not allow to fully separate individual functions from the rest of the program. Moreover, the approaches from [86, 87] based on the *potential method* [120]

³<http://www.draper.com/>

CHAPTER 1. INTRODUCTION

would enable a modular complexity analysis for term rewriting. However, they have not been successfully automated yet.

Consequently, I proposed to abstract term rewrite systems to integer programs via a *size abstraction*, which became the first contribution of [103]. In this way, the modularity of complexity analysis tools for integer programs can also be exploited for the analysis of term rewrite systems. Although I wrote the first draft of the corresponding part of [103], its details were mainly worked out by my co-authors. Thus, the size abstraction from [103, Section 3] is not presented within this thesis.

However, while term rewriting supports full recursion, many complexity analysis tools for integer programs only support restricted forms of recursion [8, 27, 71, 114] (see Section 1.2 for a more detailed discussion of these restrictions). Thus, these tools are not applicable to the recursive integer programs resulting from the transformation presented in [103, Section 3]. To solve this problem, I proposed the technique presented in Chapter 5, which allows to apply arbitrary complexity analysis techniques for non-recursive integer programs also to recursive integer programs. To this end, it analyzes blocks of mutually recursive functions \mathcal{P} individually in a bottom-up fashion by first inferring bounds for \mathcal{P} 's runtime and the size of \mathcal{P} 's result (cf. Section 5.2) and then eliminating all calls to \mathcal{P} from the analyzed program (cf. Section 5.3). Thereby, we only consider integer programs where all variables range over \mathbb{N} , since the abstraction from [103, Section 3] does not yield negative values.

Thus, the first major contribution of Chapter 5 is that it allows to lift each complexity analysis tool for non-recursive integer programs to recursive programs, as it is completely independent of the underlying complexity analysis tool. The second major contribution of Chapter 5 is its modularity: Every function is analyzed once and only once, independently from the rest of the program. Finally, just like the size bounds from [27], the size bounds inferred in Chapter 5 are often super-linear. To achieve this, we use a novel technique which searches for size bounds by constructing an ITS whose complexity corresponds to the result of the analyzed function. In this way, we can again exploit the modularity of techniques like [27] to infer (often super-linear) size bounds by synthesizing linear ranking functions. Consequently, our technique is particularly suitable for the analysis of programs with super-linear growth of data. Note that such programs are challenging for existing tools for the analysis of recursive integer programs like CoFloCo [48, 50]. The reason is that the inference of super-linear invariants is complicated and expensive, such that most tools restrict themselves to linear invariants which are insufficient to track super-linear growth of data.

The technique presented in Chapter 5 was designed in the course of a Bachelor thesis which I supervised. A preliminary version of Chapter 5 was published in [103, Section 4]. I coordinated the implementation, wrote the first version of [103, Section 4] and, together with my co-authors, kept improving the presentation until it was ready for publication. Moreover, I was in charge of the proofs.

1.3. CONTRIBUTIONS AND PUBLICATIONS

Unfortunately, the formalization from [103, Section 4] turned out to be disadvantageous for proving the correctness of our approach. Although the technique seems rather intuitive, the proofs span over 30 pages. Thus, I decided to re-formalize the technique within this thesis, resulting in the significantly shorter proofs in Chapter 5. Thus, in comparison to [103], the major contribution of Chapter 5 is the streamlined formalization, resulting in shorter and more reliable proofs.

More precisely, I adapted the definition of the rewrite relation from [103] such that arithmetic expressions are evaluated “on demand”, whereas they were evaluated eagerly in [103]. Note that this does not affect the complexity of the considered rewrite systems, as evaluating arithmetic expressions is “for free” in [103] and Chapter 5. However, it significantly simplifies the proofs, as evaluating arithmetic (sub-)expressions changes the structure of a term (e.g., “`fib(1 + 1)`” becomes “`fib(2)`”). With the definition from [103], this structural change has to be taken into account in the proofs whenever they reason about rewrite steps, which is not required with the alternative definition from Chapter 5.

Moreover, the formalization in Chapter 5 is significantly more modular than the formalization in [103]. To see this, recall that Chapter 5 describes a bottom-up technique, which starts with the analysis of auxiliary functions and then eliminates all calls to these functions. While this elimination was done in a single step in [103], it is now decomposed into four steps whose correctness is proven individually.

1.3.3 Contributions of Chapter 8 and [55, Section 4]

Chapter 8 introduces *loop detection*, one of the first techniques to infer lower bounds on the worst-case complexity of term rewrite systems automatically. From a practical point of view, the main contribution of Chapter 8 is the wide applicability of loop detection: In our experiments (cf. Chapter 12) it proves linear lower bounds (see Section 8.1) for almost all analyzed examples. Moreover, exponential lower bounds (see Section 8.2) can be proven in many cases. In Section 8.4, we also adapt loop detection to *innermost* rewriting, which further increases its applicability in practice, as real-world programs are often evaluated eagerly.

From a theoretical point of view, it is remarkable that the notion of *decreasing loops*, which is the foundation of loop detection, generalizes the notion of *loops*. Searching for loops is one of the most important techniques to prove non-termination of term rewrite systems. Thus, loop detection can also prove that the complexity of a term rewrite system is unbounded. Moreover, Section 8.3 contains the non-trivial proof that loop detection is not a semi-decision procedure for the inference of linear lower bounds, even for quite restrictive classes of term rewrite systems. Note that this question arises naturally due to the power of loop detection in our experiments. As we are not aware of a suitable counterexample, the incompleteness of loop detection is proven indirectly by showing that the question whether a TRS has at least linear complexity is

CHAPTER 1. INTRODUCTION

undecidable.

A preliminary version of Chapter 8 was published in [55, Section 4]. I proposed the notion of decreasing loops for linear lower bounds, their adaption to exponential lower bounds, and the reduction used in the proof of Section 8.3. Moreover, I implemented loop detection in our tool AProVE. Finally, I wrote the first draft of [55, Section 4] including the initial versions of most proofs and, together with my co-authors, kept revising it until it was ready for publication. In comparison to [55], Chapter 8 fixes two inadequacies. The proof of [55, Lemma 53] used the incorrect assumption that the number of variables in $t\sigma$ is bounded by the number of variables in t if t is a linear term and σ is a substitution which is used to narrow t with a linear term rewrite system \mathcal{R} . A minimal counterexample to this assumption is

$$t = f(x) \text{ and } \mathcal{R} = \{f(c(y, z)) \rightarrow a\},$$

where $f(x)$ can be narrowed to a using the substitution $\sigma = \{x/c(y, z)\}$. Clearly, the number of variables in $t\sigma = f(c(y, z))$ exceeds the number of variables in $t = f(x)$. This issue is fixed in the proof of the corresponding Lemma 8.18. Moreover, Theorem 8.12 revises [55, Theorem 37], where we stated that the existence of d compatible decreasing loops implies that the complexity of the term rewrite system is in $\Omega(d^n)$. While it is correct to deduce that the complexity of the term rewrite system is at least exponential (i.e., it is in $\Omega(c^n)$ for some $c > 0$), we cannot infer the base of the exponential function.

The second significant contribution of Chapter 8 in comparison to [55] is the adaption of loop detection to innermost *relative* rewriting. The adaption of loop detection to innermost rewriting from [55, Appendix A] does not consider relative rules. However, the adaption to relative rewriting is non-trivial, since the existence of an innermost decreasing loop as defined in [55, Appendix A] only implies a linear lower bound if the relative part of the term rewrite system terminates.

1.3.4 Contributions of Chapter 9, [54], and [55, Section 3]

After the loop detection technique from Chapter 8, Chapter 9 introduces the second technique for the inference of worst-case lower bounds for term rewrite systems, the so-called *induction technique*. The most important contribution of the induction technique is that it is the first (and currently only) technique which is able to infer super-linear polynomial worst-case lower bounds for term rewrite systems. Hence, since linear runtime is often insufficient to enable denial-of-service attacks, it is especially valuable for the detection of denial-of-service vulnerabilities.

To infer (possibly super-linear or even exponential) lower bounds, the induction technique relies on *equational rewriting* and *rewrite lemmas*, which allows us to represent possibly infinite families of rewrite sequences concisely, cf. Section 9.1. Such families of rewrite sequences are suitable to witness lower bounds on the

1.3. CONTRIBUTIONS AND PUBLICATIONS

worst-case complexity. The core of the induction technique consists of a heuristic to *speculate* likely families of rewrite sequences (Section 9.2) and a technique to prove their *validity* by induction afterwards (Section 9.3). By inspecting the structure of the resulting inductive proof, we can deduce a lower bound on the complexity of the considered family of rewrite sequences (Section 9.4). By taking the size of the start terms of the rewrite sequences into account, this also yields a lower bound on the runtime of the analyzed TRS (Section 9.5).

Section 9.6 shows how to prove lower bounds if we only have partial information about the considered family of rewrite sequence by using so-called *indefinite lemmas*. More precisely, indefinite lemmas represent rewrite sequences whose result is unknown. Nevertheless, they give rise to non-trivial lower bounds in many cases.

To improve the applicability of the induction technique, Section 9.7 introduces the, to the best of my knowledge, first under-approximating *argument filtering technique* for term rewrite systems. Similar techniques have long been used for termination analysis and for the inference of upper complexity bounds, where they are used in an over-approximating setting (see, e.g., [38, 68, 105]). Hence, existing techniques are usually unsound in an under-approximating setting, i.e., they cannot be used if one is interested in lower instead of upper bounds.

Finally, Section 9.8 shows how to adapt the induction technique for the analysis of *innermost* instead of full rewriting.

Preliminary versions of Chapter 9 were published in [54] and [55, Section 3], where [54] focused on innermost rewriting and [55] focused on full rewriting. While the underlying idea of proving the existence of families of rewrite sequences by induction in order to infer lower runtime bounds dates back before my time as a PhD student (as it has, e.g., been illustrated informally in [80]), I conceptualized the presented technique to speculate potentially valid families of rewrite sequences (Section 9.2). Moreover, the techniques to deduce lower bounds on the costs of rewrite lemmas (Section 9.4) and whole term rewrite systems (Section 9.5) have been worked out by me. I also designed the under-approximating argument filtering technique from Section 9.7 and contributed to the conception of the technique presented in Section 9.6, which allows us to reason about families of rewrite sequences without knowing their result.

Furthermore, I wrote the first version of both [54] and [55, Section 3] and constantly contributed to both papers until their publication. Finally, I implemented the induction technique in our tool AProVE.

In comparison to [54] and [55, Section 3], Chapter 9 introduces a notion of complexity for *equational rewriting*. Then the first step of the analysis is to transform the standard term rewrite system that needs to be analyzed into an equational term rewrite system. Hence, all major theorems (as well as their proofs) can entirely focus on equational rewriting. While equational rewriting has also been used in [54, 55], there was no corresponding notion of complexity. Thus, many theorems in [54, 55] had to link statements about equational rewriting back to standard term rewriting to express complexity-

CHAPTER 1. INTRODUCTION

related properties. Hence, in comparison to [54, 55], the formalization as well as the proofs of Chapter 9 are streamlined and simplified.

Moreover, the techniques presented in Section 9.6 and Section 9.7 were missing in [55] and hence they are adapted to full rewriting for the first time within this thesis. Although this adaption is not very challenging, it is nevertheless interesting as the resulting technique from Section 9.6 is orthogonal to the corresponding technique for innermost rewriting from [54]. While [54] had to impose severe restrictions on the technique to prove rewrite lemmas by induction in the presence of indefinite lemmas, these restrictions are not needed in Section 9.6. In contrast, Section 9.6 requires that the analyzed system is left-linear, which is only a mild restriction in practice.

Another contribution of Chapter 9 in comparison to [54, 55] is Theorem 9.31, which is stronger than the corresponding Corollary 25 from [55] resp. Theorem 14 from [54]. The difference is that Theorem 9.31 yields a *concrete*, potentially multivariate lower bound which is *asymptotically* correct. In contrast [55, Corollary 25] and [54, Theorem 14] just yield a (univariate) asymptotic complexity class. However, to implement the induction technique, a concrete lower bound is required.

Finally, Algorithm 4 is significantly more detailed than the corresponding Algorithm 13 from [55].

1.3.5 Contributions of Chapter 10

The main contribution of Chapter 10, which is completely unpublished, is the proof that the question whether the runtime complexity of a term rewrite system is constant is semi-decidable for full (Section 10.1) as well as innermost (Section 10.2) rewriting. The resulting semi-decision procedure exploits that termination of a restricted form of *narrowing* is semi-decidable.

Chapter 10 complements the related result from Section 8.3 that the question whether the runtime complexity of a term rewrite system is at least linear is undecidable. To prove semi-decidability, some results from Section 8.3 have to be generalized from the restricted class of term rewrite systems that is considered in Section 8.3 to arbitrary term rewrite systems. Since the term rewrite systems considered in Section 8.3 are restricted to tail recursion, this adaption is non-trivial.

From a practical point of view, Chapter 10 is of interest for two reasons. First, it is able to prove constant upper bounds for term rewrite systems where the leading complexity analysis tools failed to do so up to now, cf. Chapter 12. Second, it has interesting applications in practice, as the runtime of non-trivial algorithms is usually *not* constant. Hence, the technique presented in Chapter 10 can be used to identify common bugs that result in constant runtime like, e.g., unsatisfiable loop conditions.

1.3. CONTRIBUTIONS AND PUBLICATIONS

1.3.6 Contributions of Chapter 11 and [53]

Chapter 11 introduces a *strategy switching* technique to prove that the innermost runtime complexity of a term rewrite system coincides with its full runtime complexity, i.e., that eager evaluation is the “worst” possible reduction strategy for a given TRS. In this way, strategy switching allows us to use all existing and future techniques for the inference of upper bounds on the innermost runtime complexity of TRSs to also analyze their full runtime complexity. This results in a significant gain in power for state-of-the-art complexity analysis tools, since their support for innermost rewriting is much more sophisticated than for full rewriting as discussed in detail at the beginning of Chapter 11. Hence, strategy switching has important applications regarding the analysis of programs with non-eager evaluation strategies, such as MAUDE programs [117].

Dual to its applications for the inference of upper bounds, strategy switching makes techniques for the inference of lower bounds for full rewriting available for the analysis of innermost rewriting.

Chapter 11 builds upon a result from [124], which essentially states that innermost rewriting is the least efficient evaluation strategy as long as a TRS does not duplicate *redexes*, i.e., subterms whose evaluation is not yet finished, cf. Section 11.1. To exploit this result, Section 11.2 presents a novel technique to prove that redexes will never be duplicated. Thereby, only rewrite sequences that correspond to the evaluation of a single function are taken into account (i.e., rewrite sequences starting with so-called *basic terms*). This is in line with the established definition of runtime complexity for term rewrite systems.

The core idea of the presented technique is to approximate all terms that might lead to duplication as well as all terms that might contain nested redexes. If those approximations do not “overlap”, then redexes cannot be duplicated, which proves that innermost rewriting is *always* the worst evaluation strategy and hence innermost and full runtime complexity coincide.

The technique presented in Chapter 11 has previously been published in [53]. There, we also extended the technique to handle so-called *non-constructor systems*. As this class of term rewrite systems is irrelevant in the context of program verification, this extension is not presented in this thesis. I contributed to [53] by developing the presented approximations, writing the first draft of the paper including all proofs, and implementing the presented technique. Moreover, I was heavily involved in the process of polishing the paper until its publication. The evaluation of our strategy switching technique in Chapter 12 significantly extends the experimental evaluation from [53], as it also shows the effect of combining it with techniques for the inference of lower bounds, whereas [53] focused on upper bounds.

1.3.7 Further Publications

Apart from the publications mentioned in the previous sections, I contributed to several other papers during my time as a PhD student.

CHAPTER 1. INTRODUCTION

- The tool paper [64] and its extended version [62] present AProVE, one of the leading fully automatic termination and complexity analysis tools. Besides being one of the main developers of AProVE, my main contribution to these papers was the implementation of significant extensions of AProVE's Eclipse plugin. Note that AProVE's Eclipse plugin was one of the main novelties presented in [62, 64].
- The paper [118] and its extended version [119] present a technique to prove memory safety and termination of C programs. Besides minor contributions to the implementation and proofreading, I was in charge of the experimental evaluation of [118].
- In [75] and its extended version [76], we extended our technique from [118, 119] in order to prove safety, termination, and upper complexity bounds for C programs with bitvector arithmetic (whereas we assumed **ints** to be mathematical integers in [118, 119]). Besides proofreading, I substantially contributed to the conception and implementation of the complexity analysis technique presented in [76].
- In [51], we presented a technique to infer upper bounds on the complexity of Java programs. I was the main author of this paper. As such, I was in charge of the development and implementation of the presented technique. Moreover, I wrote the first version of the paper and kept contributing to it until it was ready for publication.

1.4 Related Work

The goal of all techniques presented in this thesis is to compute *symbolic bounds* on the complexity of rewrite systems. More precisely, our goal is to derive a mathematical expression which is an upper or lower bound on the system’s worst-case complexity. Section 1.4.1 gives a rough overview of the large field of symbolic bounds. Later on, we will give more details about many of the mentioned techniques subsequent to related contributions of this thesis. Section 1.4.2 discusses other fields which are related, but orthogonal to symbolic bounds.

1.4.1 Symbolic Bounds

Techniques to analyze the complexity of programs automatically have been investigated since the 1970s [128]. As a result, there are many techniques to infer *upper* bounds on the *worst-case* complexity of various kinds of programs. Apart from that, there are few techniques to infer *lower* bounds on the *best-case* complexity. Regarding *lower* bounds on the *worst-case* complexity, the techniques presented in this thesis are the first of their kind to the best of my knowledge.

Note that techniques to infer lower bounds on the best-case complexity differ fundamentally from techniques for the inference of worst-case lower bounds. To deduce best-case lower bounds, one has to prove that a certain bound holds for *every* program run. Thus, as in the case of worst-case upper bounds, *over-approximating* techniques are used to ensure that the proven bound covers all program runs, i.e., even though such techniques under-approximate the runtime of the program, they over-approximate the set of all program runs.

In contrast, techniques to infer lower bounds on the worst-case complexity have to identify families of inputs (i.e., witnesses) that result in expensive program runs. Thus, for the inference of worst-case lower bounds, over-approximations are usually unsound, since one has to ensure that the witness of the proven lower bound corresponds to “real” program runs. Thus, *under-approximating* techniques have to be used in order to infer lower bounds on the worst-case complexity.

Most existing complexity analysis techniques either analyze

- programs operating on integers,
- term rewrite systems resp. first-order functional programs, or
- real-world programming languages.

Programs Operating on Integers Inferring upper bounds on the complexity of integer programs is a very active field of research, as witnessed by a large number of both tools and publications [3, 8, 9, 21, 27, 34, 35, 48, 50, 71, 72, 114]. One reason is that programs written in real-world programming languages

CHAPTER 1. INTRODUCTION

can be abstracted to integer programs in a natural way [5, 7, 44, 45, 51, 76, 115, 119]. We will discuss the relationship to existing techniques for the inference of upper bounds for integer programs in further detail in Sections 4.7 and 5.4. In contrast, there is little work on the inference of lower bounds on the best-case complexity of integer programs. The tools PUBS [3] and CoFloCo [48] can infer lower bounds on the best-case complexity of *cost relations*. Cost relations extend recurrence relations such that, e.g., non-determinism can be modeled. Moreover, [10] briefly mentions that their technique could also be adapted to infer best-case lower bounds instead of worst-case upper bounds for *abstract cost rules*, i.e., integer procedures with (possibly multiple) outputs. Since these techniques are modifications of the corresponding techniques for the inference of worst-case upper bounds and, as mentioned at the beginning of this section, incomparable to the techniques for the inference of worst-case lower bounds presented in this thesis, we will not discuss them in detail.

Term Rewrite Systems Here, the situation is even more one-sided than in the case of integer programs: There are numerous techniques for the inference of worst-case upper bounds [13, 15, 79, 84, 87, 104, 105, 127, 131], but no techniques for the inference of lower bounds are available. Remarkably, existing complexity analysis techniques support *two* notions of “complexity” for term rewrite systems. While the *derivational complexity* [15, 104, 127, 131] of term rewrite systems is of interest in the context of equational reasoning, *runtime complexity* [13, 15, 79, 87, 105] corresponds to the usual notion of complexity for programs. Thus, in this thesis, we focus on runtime complexity. However, since the runtime complexity of a term rewrite system is always a lower bound on its derivational complexity, the presented techniques for worst-case lower bounds can directly be used to analyze derivational complexity as well.

While derivational and runtime complexity differ w.r.t. the allowed start terms, one also obtains different notions of complexity for term rewriting by distinguishing different *evaluation strategies*. In the past, *innermost rewriting* [13, 15, 79, 87, 105] (i.e., rewriting with an eager evaluation strategy) and *full rewriting* [15, 79, 104, 127, 131] (i.e., rewriting with a completely unrestricted evaluation strategy) have been studied. All techniques presented in the current thesis apply to both, innermost and full rewriting.

Real-World Programming Languages From a practical point of view, techniques to analyze the complexity of real programming languages are certainly the most interesting ones. However, these techniques rarely analyze the complexity of programs directly. Instead, as mentioned in Chapter 1, they transform programs into simpler formalisms like integer programs [7, 51, 76], term rewrite systems [65], or combinations thereof [101].

Relying on powerful techniques for automated termination analysis of rewrite systems (e.g., [26, 62, 78, 96]), similar transformations have successfully been used to prove *termination* of programs written in real-world languages for many

1.4. RELATED WORK

years [5, 28, 29, 44, 45, 63, 65, 75, 107, 115, 119]. In comparison, complexity-preserving transformations from real-world languages to rewrite systems haven't seen less attention. Clearly, the appeal of such transformational approaches stands and falls with the power of the available complexity analysis tools for rewriting. Thus, we hope that the techniques presented in this thesis will inspire further complexity analysis techniques for real-world programs on the long run.

1.4.2 Related Fields

The two other most closely related fields of research are concerned with solving *recurrence relations* and predicting the *worst-case execution time* of programs.

Recurrence Relations Recurrence relations are of the form $x^{(n)} = \dots$ where the right-hand side of the equation again contains sub-expressions $x^{(\dots)}$. In this way, together with suitable initial conditions, recurrence relations recursively define a sequence $x^{(0)}, x^{(1)}, \dots$. From a different point of view, they can be seen as simple numeric “programs” to compute $x^{(n)}$ recursively and thus they are related to integer programs resp. integer rewrite systems. Like integer rewrite systems, they can be used to model the complexity of programs in a natural way in many cases. For example, the number of loop iterations required to traverse a non-empty list of length n is $1 + x^{(n-1)}$, resulting in the recurrence relation $x^{(n)} = 1 + x^{(n-1)}$ with the initial condition $x^{(0)} = 0$ (for traversing an empty list). Solving recurrence relations, i.e., finding a closed form for $x^{(n)}$, is supported by highly specialized solvers like PURRS [18] and many computer algebra systems like Maple [74] or Mathematica [130]. However, these tools usually aim to find *exact* solutions. In contrast, since describing the exact runtime of a program in terms of a closed-form expression is infeasible in most cases, techniques for the computation of symbolic bounds sacrifice exactness and compute approximations instead. Moreover, there is little tool support for multi-dimensional recurrence relations of the form $x^{(n_1, \dots, n_k)} = \dots$. However, since many variables may influence the complexity of programs, multi-dimensional recurrence relations naturally arise if one tries to express the complexity of a program in terms of recurrence relations. Finally, recurrence relations do not offer features like case analyses or non-determinism, which are heavily exploited by complexity-preserving transformations from real-world programming languages to rewrite systems like [7, 51, 76].

Worst-Case Execution Time Tools that tackle that Worst-Case Execution Time (WCET) problem [129] try to estimate the runtime of programs in (milli-)seconds. To this end, the underlying execution platform has to be taken into account, i.e., many low-level aspects like the architecture of the processor have to be modeled in detail, whereas tools for symbolic bound computation use simple, idealized cost models. On the other hand, WCET tools usually require user-provided annotations that describe the number of iterations of loops, whereas tools for symbolic bounds work fully automatically. Thus, the research

CHAPTER 1. INTRODUCTION

on symbolic bounds and WCET is largely orthogonal and joining techniques from both fields is subject of future work.

Preliminaries

In this chapter, we first introduce *terms* and various related concepts in Section 2.1. This lays the foundation for the definition of our notions of rewrite systems in Part II and Part III. Afterwards, we introduce the framework which will be used to formalize complexity analysis techniques throughout this thesis in Section 2.2. It is based on *complexity problems* (cf. Definition 2.16), which offer a flexible way to describe various notions of complexity for different models of computation. All complexity analysis techniques presented later on either yield a bound for a complexity problem directly, or they transform one complexity problem into another one using *processors* (cf. Definition 2.22). The goal of such a processor is to obtain a new complexity problem which is in some sense simpler, smaller, or easier to analyze than the original one.

2.1 Terms and Related Concepts

A *signature* Σ is a (possibly infinite) set of *function symbols* where each function symbol $f \in \Sigma$ has a fixed arity $\text{ar}_\Sigma(f) = n$. For each $n \in \mathbb{N}$, $\Sigma^n \subseteq \Sigma$ just contains the function symbols from Σ whose arity is n . A signature Σ and a set of variables \mathcal{V} allow us to build terms.

Definition 2.1 (Terms). Given a signature Σ and a set of variables \mathcal{V} , the set $\mathcal{T}(\Sigma, \mathcal{V})$ of all *terms* over Σ and \mathcal{V} is the smallest set containing

- (1) x if $x \in \mathcal{V}$ and
- (2) $f(t_1, \dots, t_n)$ if $f \in \Sigma^n$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for each $i \in \{1, \dots, n\}$.

$\mathcal{T}(\Sigma) = \mathcal{T}(\Sigma, \emptyset)$ is the set of all *ground terms* over Σ .

$\mathcal{V}(t)$ is the set of all variables and $\Sigma(t)$ is the set of all function symbols that occur in t . We write $\#_x(t)$ to denote the number of occurrences of $x \in \mathcal{V}$ in t .

A term t is *linear* if $\#_x(t) \leq 1$ for all $x \in \mathcal{V}$.

We lift the notations $\mathcal{V}(t)$ and $\Sigma(t)$ to sets of terms in the obvious way (so we have, e.g., $\mathcal{V}(T) = \bigcup_{t \in T} \mathcal{V}(t)$ for each $T \subseteq \mathcal{T}(\Sigma, \mathcal{V})$).

To avoid clumsy notations like $f(t_1, \dots, t_n)$, we sometimes use *row vectors*. To represent row vectors like (t_1, \dots, t_n) , we use bold symbols like \mathbf{t} and we refer to their length as $\text{len}(\mathbf{t}) = n$. For n -ary functions or function symbols f , we identify $f(\mathbf{t})$ and $f(t_1, \dots, t_n)$. Unary functions g are applied to vectors componentwise, i.e., $g(\mathbf{t}) = (g(t_1), \dots, g(t_n))$. Similarly, we extend relations \circ to vectors componentwise, i.e., we have $\mathbf{t} \circ \mathbf{s}$ if $\text{len}(\mathbf{t}) = \text{len}(\mathbf{s})$ and $\mathbf{t}|_i \circ \mathbf{s}|_i$ for all $i \in \{1, \dots, \text{len}(\mathbf{t})\}$ where $\mathbf{t}|_i$ is the i^{th} element of \mathbf{t} . The vector resulting from \mathbf{t} by replacing its i^{th} element with s is $\mathbf{t}[s]_i$. We sometimes use vectors as sets, i.e., we write $t \in \mathbf{t}$, $\mathbf{t} \subseteq T$, etc. Finally, we define $\sum \mathbf{t} = \sum_{i=1}^{\text{len}(\mathbf{t})} \mathbf{t}|_i$.

Example 2.2 (Signatures and Terms). $\Sigma = \{\text{cons}, \text{nil}, \text{succ}, \text{zero}\}$ is the signature which will be used to represent natural number and lists as terms throughout this thesis. The natural number n is represented by the term $\text{succ}^n(\text{zero})$, the empty list is represented by nil , and the list with head n and tail $xs \in \mathcal{V}$ is represented by $\text{cons}(\text{succ}^n(\text{zero}), xs)$.

Here and throughout this thesis, $f^n(x)$ denotes the n -fold application of f , i.e., $f^n(x) = \underbrace{f(\dots f(x) \dots)}_{n \times}$.

To reason about terms, it is often useful to talk about *subterms* in a concise way. To this end, the notion of *positions* can be used to address subterms directly.

2.1. TERMS AND RELATED CONCEPTS

Definition 2.3 (Positions and Subterms). Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$. Then the set $\text{pos}(t) \subseteq \mathbb{N}^*$ of t 's *positions* is defined inductively as follows:

$$\text{pos}(t) = \begin{cases} \epsilon & \text{if } t \in \mathcal{V} \\ \{\epsilon\} \cup \{i.\pi \mid i \in \{1, \dots, n\}, \pi \in \text{pos}(t_i)\} & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

We write $\pi \leq \pi'$ (resp. $\pi < \pi'$) if π is a (proper) prefix of π' . If $\pi \not\leq \pi'$ and $\pi' \not\leq \pi$, then π and π' are *independent* ($\pi \parallel \pi'$). For each $\pi \in \text{pos}(t)$, the term $t|_\pi$ is defined by:

$$t|_\pi = \begin{cases} t & \text{if } \pi = \epsilon \\ t_i|_{\pi'} & \text{if } \pi = i.\pi' \text{ and } t = f(t_1, \dots, t_n) \end{cases}$$

Furthermore, $t[s]_\pi$ is the term that results from t by replacing $t|_\pi$ with s , i.e.:

$$t[s]_\pi = \begin{cases} s & \text{if } \pi = \epsilon \\ f(t_1, \dots, t_i[s]_{\pi'}, \dots, t_n) & \text{if } \pi = i.\pi' \text{ and } t = f(t_1, \dots, t_n) \end{cases}$$

A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ is a *subterm* of t ($t \triangleright s$) if there is a position $\pi \in \text{pos}(t)$ such that $t|_\pi = s$. We also use the notation $t \triangleright_\pi s$. It is a *proper subterm* of t ($t \triangleright s$) if $\pi \neq \epsilon$.

Example 2.4. Let $t = \text{cons}(\text{zero}, \text{cons}(\text{succ}(\text{zero}), \text{nil}))$. The positions of t are $\text{pos}(t) = \{\epsilon, 1, 2, 2.1, 2.2, 2.1.1\}$. We have $t|_{2.1} = \text{succ}(\text{zero})$ and thus $t \triangleright \text{succ}(\text{zero})$. Moreover, we have $t[\text{zero}]_{2.1} = \text{cons}(\text{zero}, \text{cons}(\text{zero}, \text{nil}))$.

Substitutions allow us to instantiate variables in terms.

Definition 2.5 (Substitution). Let Σ be a signature and let \mathcal{V} be a set of variables. A substitution is a function $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$.

The *domain* of σ is $\text{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$. We lift substitutions to terms homomorphically, i.e., we define $\sigma(f(\mathbf{t})) = f(\sigma(\mathbf{t}))$. Substitutions with finite domain can be denoted as finite sets of pairs $\{x_1/\sigma(x_1), \dots, x_n/\sigma(x_n)\}$. We write $\{\mathbf{x}/\mathbf{t}\}$ for the substitution which replaces $\mathbf{x}|_i$ with $\mathbf{t}|_i$ for each $1 \leq i \leq \text{len}(\mathbf{x}) = \text{len}(\mathbf{t})$. As usual, we use substitutions in postfix notation, i.e., we define $t\sigma = \sigma(t)$. Moreover, $\sigma \diamond \sigma'$ denote the *composition* of σ and σ' , i.e., $x(\sigma \diamond \sigma') = (x\sigma)\sigma'$. Finally, for every $V \subseteq \mathcal{V}$, the *restriction of σ to V* is $\sigma|_V$ with $x\sigma|_V = x\sigma$ if $x \in V$ and $x\sigma|_V = x$, otherwise.

Note that substitution composition is defined in a different way than function composition: While $f \circ g(x) = f(g(x))$ holds for standard function composition, we have $\sigma \diamond \theta(x) = \theta(\sigma(x))$ for substitution composition. Hence, we use the non-

CHAPTER 2. PRELIMINARIES

standard symbol \diamond instead of \circ for substitution composition to avoid confusion.

Moreover, the meaning of “domain” in the context of substitutions is non-standard. Thus, for functions $f : S \rightarrow T$, we refer to S as f ’s *domain of definition* (instead of just “ f ’s domain”). As usual, T is called the *codomain* of f and the *image* of f is $\text{img}(f) = \{f(x) \mid x \in S\}$.

Lastly, while the restriction $f|_X : X \rightarrow Z$ of a function $f : Y \rightarrow Z$ (where $X \subseteq Y$) restricts f ’s domain of definition, the restriction $\sigma|_V$ of a substitution restricts σ ’s domain, i.e., $y\sigma|_V = y$ is well-defined if $y \in \mathcal{V} \setminus V$, whereas $f|_X(y)$ is undefined if $y \in Y \setminus X$.

Example 2.6. Consider the two substitutions $\sigma_1 = \{x/\text{succ}(x)\}$ and $\sigma_2 = \{x/\text{zero}, xs/\text{nil}\}$. Applying σ_1 and σ_2 to the term $\text{succ}(x)$ yields $\text{succ}(x)\sigma_1 = \text{succ}(\text{succ}(x))$ and $\text{succ}(x)\sigma_2 = \text{succ}(\text{zero})$. The composition $\sigma_1 \diamond \sigma_2$ of σ_1 and σ_2 is $\{x/\text{succ}(\text{zero}), xs/\text{nil}\}$.

Note that substitution composition is associative, since we have

$$t((\sigma \diamond \sigma') \diamond \sigma'') = (t(\sigma \diamond \sigma'))\sigma'' = ((t\sigma)\sigma')\sigma'' = (t\sigma)(\sigma' \diamond \sigma'') = t(\sigma \diamond (\sigma' \diamond \sigma'')).$$

Substitutions are the foundation of “pattern matching”, a well-known concept from, e.g., functional programming. Matching is related to *unification*, which is, e.g., used in logic programming.

Definition 2.7 (Matching and Unification). Let $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. We say that s *matches* t if there is a substitution σ such that $s\sigma = t$. Then σ is called the *matcher* of s and t .

We say that s and t *unify* if there is a substitution σ such that $s\sigma = t\sigma$. Then σ is a *unifier* of s and t . If for every unifier θ of s and t , there is a substitution μ such that $\theta = \sigma \diamond \mu$, then σ is the *most general unifier* of s and t (written $\text{mgu}(s, t) = \sigma$).

It is well known that $\text{mgu}(s, t)$ exists and is unique up to variable renaming for unifiable terms s and t .

Example 2.8. The term $s = \text{cons}(x, \text{nil})$ matches $t = \text{cons}(\text{zero}, \text{nil})$, since $s\{x/\text{zero}\} = t$. While s does not match $t' = \text{cons}(y, ys)$, s and t' unify since we have, e.g., $s\sigma = t'\sigma$ for $\sigma = \{y/x, ys/\text{nil}\}$. The substitution $\theta = \{x/\text{zero}, y/\text{zero}, ys/\text{nil}\}$ is also a unifier of s and t' and we have $\theta = \sigma \diamond \mu$ with $\mu = \{x/\text{zero}, y/\text{zero}\}$. In fact, such a substitution μ exists for *every* unifier θ of s and t' , since σ is the most general unifier of s and t' , i.e., we have $\text{mgu}(s, t') = \sigma$.

Contexts are terms with a single occurrence of a “placeholder” \square . They are useful to express that a term has a “dedicated” subterm without specifying its

2.1. TERMS AND RELATED CONCEPTS

position explicitly.

Definition 2.9. A *context* $C \in \mathcal{T}(\Sigma \cup \{\square\}, \mathcal{V})$ is a term such that $C|_{\pi} = \square$ for one and only one $\pi \in \text{pos}(C)$. We define $C[t] = C[t]_{\pi}$ for all terms t . W.l.o.g., we always assume that signatures do not contain the function symbol \square .

Thus, $C = \text{cons}(\square, xs)$ and $D = \text{cons}(\text{zero}, \square)$ are contexts and we have $C[\text{zero}] = D[xs] = \text{cons}(\text{zero}, xs)$.

2.2 Complexity Problems

In this section, we introduce the complexity analysis framework which will be used throughout this thesis. The goal of this framework is to represent different

- (1) models of computation (like integer and term rewriting),
- (2) notions of complexity, and
- (3) complexity analysis techniques

in a uniform way.

To achieve (1), observe that essentially every model of computation relies on some notion of *states* and defines how to transform the current state into a new one using some kind of *transition relation*. Thus, given a state space S , *binary relations* on S are suitable to solve (1). However, to achieve (2), it is important to distinguish “cheap” and “expensive” calculation steps. For example, in a rewrite system which models the memory consumption of a **Java** program, steps that correspond to arithmetic operations in the original program are most likely “for free”, whereas steps corresponding to the creation of new objects are costly. Thus, instead of binary relations, we use ternary relations where the additional component represents the *costs* of calculation steps.

Definition 2.10 (Weighted Relation). Let S be a set. We call $\rightarrow \subseteq S \times \mathbb{R} \times S$ a *weighted relation* on S . We write $s_0 \xrightarrow{\ell} s_1$ if $(s_0, \ell, s_1) \in \rightarrow$. Furthermore, we write $s_0 \xrightarrow{\ell}^n s_n$ if $s_0 \xrightarrow{\ell_1} \dots \xrightarrow{\ell_n} s_n$ and $\ell = \sum_{i=1}^n \ell_i$. Finally, we write $s_0 \xrightarrow{\ell}^* s_n$ if $s_0 \xrightarrow{\ell}^n s_n$ for some $n \in \mathbb{N}$ and $s_0 \xrightarrow{\ell}^+ s_n$ if $s_0 \xrightarrow{\ell}^n s_n$ for some $n \in \mathbb{N}$ with $n > 0$. We sometimes omit the costs ℓ if they are irrelevant.

Example 2.11. Let $S = \{a, b, c, d, e\}$. An example for a weighted relation on S is \rightarrow with:

$$a \xrightarrow{1} b \qquad a \xrightarrow{0} c \qquad b \xrightarrow{1} d \qquad c \xrightarrow{1} e$$

Now the complexity or *derivation height* of a state $s \in S$ is simply the maximal ℓ such that $s \xrightarrow{\ell}^* t$ for some state t .

Definition 2.12 (Derivation Height [81]). Let S be a set and let \rightarrow be a weighted relation on S . The *derivation height* $\text{dh}_{\rightarrow} : S \rightarrow \mathbb{R} \cup \{\omega\}$ of \rightarrow is defined as

$$\text{dh}_{\rightarrow}(s) = \sup\{\ell \in \mathbb{R} \mid t \in S, s \xrightarrow{\ell}^* t\}.$$

Since \rightarrow^* also permits “empty” program runs¹, we always have $\text{dh}_{\rightarrow}(s) \geq 0$.

¹Throughout this thesis, terms like “program run”, “evaluation sequence”, “rewrite sequence”, or just “sequence” are used as synonyms for (potentially infinite) chains $s_0 \rightarrow s_1 \rightarrow \dots$ w.r.t. a weighted relation \rightarrow .

2.2. COMPLEXITY PROBLEMS

Example 2.13. For the weighted relation from Example 2.11, we have $\text{dh}_{\rightarrow}(\mathbf{a}) = 2$ since we have $\mathbf{a} \xrightarrow{1} \mathbf{b} \xrightarrow{1} \mathbf{d}$. Moreover, we have $\text{dh}_{\rightarrow}(\mathbf{b}) = \text{dh}_{\rightarrow}(\mathbf{c}) = 1$ and $\text{dh}_{\rightarrow}(\mathbf{d}) = \text{dh}_{\rightarrow}(\mathbf{e}) = 0$.

If the analyzed program is *terminating* (i.e., there is no infinite sequence $s_0 \rightarrow s_1 \rightarrow \dots$) and all calculation steps have non-negative costs, then there is always a cost-maximal sequence $s \xrightarrow{\mathbb{R}}^* t$ such that t is a “final state”, i.e., there is no way to evaluate t any further. Formally, the concept of “final states” is captured by the notion of *normal forms*.

Definition 2.14 (Normal Form). Let \rightarrow be a weighted relation on S and let $s \in S$. If there is no $t \in S$ such that $s \rightarrow t$, then s is called a *normal form* w.r.t. \rightarrow . If we have $t \rightarrow^* s$ and s is a normal form, then s is a normal form of t . If t has a unique normal form s , then we write $t \downarrow = s$.

Example 2.15. In Example 2.11, the only normal forms are \mathbf{d} and \mathbf{e} and we have $\mathbf{b} \downarrow = \mathbf{d}$.

Besides assigning costs to calculation steps, we need two more ingredients to solve (2). First, we need to fix a set of *initial states* I . In this way, we can express that we are only interested in the cost of an evaluation $s \xrightarrow{\mathbb{R}}^* t$ if $s \in I$. If, for example, the analyzed rewrite system results from a **Java** program, this allows us to express that we are only interested in program runs starting with the main method. Second, we need to fix a *size measure* for initial states. The reason is that any statement about the complexity of a program is meaningless without specifying the underlying size measure. For example, traversing all nodes of a complete binary tree is linear in the number of nodes, but exponential in the height of the tree. *Complexity problems* combine all the ingredients to solve (1) and (2) and thus they are the foundation of our complexity analysis framework.

Definition 2.16 (Complexity Problem). Given a set S , a *complexity problem* over S is a tuple $cp = (I, \rightarrow, \|\cdot\|)$ where $I \subseteq S$, \rightarrow is a weighted relation on S , and $\|\cdot\| : I \rightarrow \mathbb{N}$. S is the *state space*, I is the set of *initial states*, \rightarrow is the *transition relation*, and $\|\cdot\|$ is the *size measure* of cp . \mathcal{CP} denotes the set of all complexity problems.

Example 2.17. Continuing Example 2.11, $cp = (\{\mathbf{a}, \mathbf{b}\}, \rightarrow, \|\cdot\|)$ with $\|\mathbf{a}\| = 3$ and $\|\mathbf{b}\| = 1$ is a complexity problem.

It remains to define the complexity of a complexity problem. While the derivation height (cf. Definition 2.12) already characterizes the complexity w.r.t. a specific start state, this notion of complexity is usually too precise to allow for concise and intuitive bounds. Thus, the use of techniques to approximate the

CHAPTER 2. PRELIMINARIES

derivation height dh is limited. Hence, we abstract over dh by defining the *runtime complexity* of a complexity problem. Its only argument is an upper bound on the size of the initial state which is mapped to the cost of the most expensive program run whose start state complies with this bound.

Definition 2.18 (Runtime Complexity). Let $cp = (I, \rightarrow, \|\cdot\|)$ be a complexity problem. The *runtime complexity* $\text{rc}_{cp} : \mathbb{N} \rightarrow \mathbb{R} \cup \{\omega\}$ is defined as

$$\text{rc}_{cp}(n) = \sup\{\text{dh}_{\rightarrow}(s) \mid s \in I, \|s\| \leq n\}.$$

Obviously, rc_{cp} is not computable if \rightarrow corresponds to the transition relation of a Turing complete model of computation. Thus, our goal is to compute bounds which are as precise as possible.

Example 2.19. For the complexity problem cp from Example 2.17, we have $\text{rc}_{cp}(1) = \text{rc}_{cp}(2) = 1$ since the only initial state whose size is bounded by 1 or 2 is \mathbf{b} and the derivation height of \mathbf{b} is 1, cf. Example 2.13. For all $n > 2$, we have $\text{rc}_{cp}(n) = 2$ since we have $\|\mathbf{a}\| \leq n$ and $\text{dh}_{\rightarrow}(\mathbf{a}) = 2$.

However, in many cases even meaningful bounds on rc are hard to express concisely. As an example, consider the HashMap implementation in the standard library of Java 8 [97]. Usually, the buckets of the hashtable are organized as lists and thus a lookup has linear worst-case complexity w.r.t. the size of the largest bucket. However, overpopulated buckets are transformed to trees, such that lookups become logarithmic. So assume that \rightarrow mimics a lookup in a hashtable, $\|\cdot\|$ measures hashtables by the size of the largest bucket, and I just contains states that comply with HashMap’s class invariant that small buckets are organized as lists, whereas large buckets are organized as trees. Then upper bounds for rc_{cp} are either (at least) linear and thus imprecise, or they explicitly distinguish the cases that the hashtable is “small” and that it is “large enough”. In contrast, *asymptotic* bounds on rc can be expressed concisely in most cases.

Definition 2.20 (Asymptotic Bounds). Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$.

We say that g is an *asymptotic lower bound* for f ($f(n) \in \Omega(g(n))$) if there is an $m > 0$ and an $n_0 \in \mathbb{N}$ such that $f(n) \geq m \cdot g(n)$ holds for all $n \geq n_0$.

We say that g is an *asymptotic upper bound* for f ($f(n) \in \mathcal{O}(g(n))$) if there is an $m > 0$ and an $n_0 \in \mathbb{N}$ such that $f(n) \leq m \cdot g(n)$ holds for all $n \geq n_0$.

We define $f(n) \in \Theta(g(n))$ if $f(n) \in \Omega(g(n))$ and $f(n) \in \mathcal{O}(g(n))$.

While we use Knuth’s definition of Ω [92], there is an alternative definition by Hardy and Littlewood [73]. According to their definition, we have $f(n) \in \Omega(g(n))$ if there is an $m > 0$ such that for each $n_0 \in \mathbb{N}$ there is an $n \geq n_0$ with $f(n) \geq m \cdot g(n)$. So it requires that for each n_0 there is some larger value n where f exceeds its lower bound. In contrast, Knuth’s variant requires

2.2. COMPLEXITY PROBLEMS

that f *always* exceeds its lower bound from some n_0 onwards. Thus, Knuth's variant is the stronger one, i.e., whenever we have $f(n) \in \Omega(g(n))$ in the sense of Definition 2.20, then we also have $f(n) \in \Omega(g(n))$ in the sense of Hardy and Littlewood. Thus, the asymptotic lower bounds computed by the techniques presented in this thesis are valid w.r.t. both definitions.

Note that Definition 2.20 is slightly non-standard, as it requires $f(n) \geq m \cdot g(n)$ resp. $f(n) \leq m \cdot g(n)$ instead of $|f(n)| \geq m \cdot |g(n)|$ resp. $|f(n)| \leq m \cdot |g(n)|$. This difference is only relevant in Chapter 4, as the considered rewrite systems disallow negative costs in all other chapters. In Chapter 4, our non-standard definition of Ω allows us to infer, e.g., the asymptotic bound $\Omega(-n)$ for an algorithm whose runtime complexity function is $rc(n) = -n$. While we think that Definition 2.20 is more meaningful than the standard definition in such cases, an asymptotic bound $\Omega(e)$ as in Definition 2.20 immediately gives rise to the asymptotic bound $\Omega(\max(0, e))$ which is also valid w.r.t. the standard definition of Ω .

Example 2.21. Continuing Example 2.19, we clearly have $rc_{cp}(n) \in \Theta(1)$ since we have $rc_{cp}(n) = 2$ for all $n \geq n_0 = 3$, i.e., the runtime complexity of cp is constant.

Now consider the complexity problem $cp' = (\mathbb{N}, \{n \xrightarrow{n} n-1 \mid n \in \mathbb{N}\}, \text{id})$ (where id denotes the identity function), i.e., we have $n \xrightarrow{n} n-1 \xrightarrow{n-1} \dots \xrightarrow{1} 0$ for each $n \in \mathbb{N}$. Thus, we have

$$rc_{cp'}(n) = \sum_{i=1}^n i = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n.$$

So we have $rc_{cp'}(n) \in \mathcal{O}(n^2)$ since $rc_{cp'}(n) \leq n^2$ for all $n \in \mathbb{N}$. Moreover, we have $rc_{cp'}(n) \in \Omega(n^2)$ since $rc_{cp'}(n) \geq \frac{1}{2} \cdot n^2$, i.e., we have $rc_{cp'}(n) \in \Theta(n^2)$.

While complexity problems satisfy our requirements (1) and (2), we still need a solution for (3), i.e., we need a “tool” to formalize complexity analysis techniques for complexity problems. To this end, we use *processors* that map a given complexity problem to a new one. Depending on the intention of a processor, it can be (asymptotically) sound for lower bounds, upper bounds, or both.

Definition 2.22 (Processor). A function $\text{proc} : \mathcal{CP} \rightarrow \mathcal{CP} \cup \{\perp\}$ is called a *processor*. We say that proc is *applicable* to a complexity problem cp if $\text{proc}(cp) \neq \perp$.

A processor is *sound for upper bounds* (resp. *lower bounds*) if $\text{proc}(cp) \neq \perp$ implies $rc_{\text{proc}(cp)} \geq rc_{cp}$ (resp. $rc_{\text{proc}(cp)} \leq rc_{cp}$). It is *equivalent* if it is sound for upper and lower bounds. It is *asymptotically sound for upper bounds* (resp. *lower bounds*) if $\text{proc}(cp) \neq \perp$ implies $rc_{cp}(n) \in \mathcal{O}(rc_{\text{proc}(cp)}(n))$ (resp. $rc_{cp}(n) \in \Omega(rc_{\text{proc}(cp)}(n))$). It is *asymptotically equivalent* if it is asymptotically sound for upper and lower bounds.

CHAPTER 2. PRELIMINARIES

Here, functions are compared point-wise, i.e., we have $\text{rc}_{\text{proc}(cp)} \geq \text{rc}_{cp}$ if $\text{rc}_{\text{proc}(cp)}(n) \geq \text{rc}_{cp}(n)$ for all $n \in \mathbb{N}$. When defining processors, we only specify their result for certain inputs and implicitly assume the result \perp for all others.

Example 2.23. We continue Example 2.21. If

$$\text{proc}(cp') = (\mathbb{N}, \{n \xrightarrow{n^2} 0 \mid n \in \mathbb{N}\}, \text{id}),$$

then proc is sound for upper bounds since we have

$$\text{rc}_{\text{proc}(cp')}(n) = n^2 \geq \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n = \text{rc}_{cp'}(n).$$

In contrast, proc is not sound for lower bounds, since we have

$$\text{rc}_{\text{proc}(cp')}(2) = 4 \not\leq 3 = \text{rc}_{cp'}(2).$$

It is, however, asymptotically sound for lower bounds, as

$$\text{rc}_{cp'}(n) \in \Omega(n^2) = \Omega(\text{rc}_{\text{proc}(cp')}(n)).$$

Thus, proc is asymptotically equivalent.

Note that one can often compute concrete bounds even if processors which are only asymptotically sound are used. To this end, one has to quantify the difference between rc_{cp} and $\text{rc}_{\text{proc}(cp)}$ and adapt the bound obtained for $\text{rc}_{\text{proc}(cp)}$ correspondingly. However, for the sake of simplicity we do not formalize such liftings from asymptotic to concrete bounds.

Now sequences of processors which are sound for upper (resp. lower) bounds can be used to simplify an initial complexity problem until we obtain a complexity problem where an upper (resp. lower) bound can be computed directly. Then the resulting bound is also valid for the initial complexity problem. Thus, all complexity analysis techniques presented in this thesis are either processors as introduced in Definition 2.22 or they directly yield a (lower or upper) bound for a complexity problem.

Related Frameworks

Our framework based on complexity problems is inspired by the framework from [131] as well as the adaptations of the *Dependency Pair Framework* [66] for complexity analysis [15, 105]. However, all of these frameworks are specific to term rewriting and use a fixed size measure. Moreover, our weighted relations allow arbitrary costs, whereas [15, 105, 131] essentially just allow costs 1 and 0 by splitting the analyzed term rewrite system into two components \mathcal{R} and \mathcal{S} such that \mathcal{S} -steps are “for free”. Regarding the analyzed transition relation, [131] is restricted to the classical (full) term rewrite relation (cf. Chapter 7), [105] is restricted to the innermost term rewrite relation (see also Chapter 7), and [15]

2.2. COMPLEXITY PROBLEMS

can handle both, innermost and full term rewriting. In contrast, we consider arbitrary transition relations. Finally, as in our framework the complexity problems from [15, 131] explicitly contain a set of start terms, whereas the start terms are implicit (and thus fixed) in the framework from [105].

On the other hand, the processors from [15] transform complexity problems into sets of new problems, which is not supported by our processors. Moreover, [15, 105, 131] also support transformation which are not complexity preserving. Then each application of a processor *proc* is annotated with additional information to obtain a bound for the original complexity problem from bounds for the complexity problem(s) resulting from *proc* (which are called *DT problems* in [105]). Thus, the notions of processors from [15, 105, 131] are more general than ours. However, the additional flexibility of these processors is not needed for the formalization of the complexity analysis techniques presented in this thesis.

Part II

Complexity Analysis of Integer Rewrite Systems

Introduction

In this part, we present two complexity analysis techniques for (subclasses of) integer rewrite systems. Both techniques have been inspired by the modular analysis implemented in the tool **KoAT** [27], which infers runtime and size bounds for program parts independently to achieve compositionality.

Chapter 4 introduces the first technique for the inference of lower bounds for integer transition systems, i.e., integer rewrite systems with limited support for recursion. Like the approach from [27], the presented analysis is completely modular. To achieve compositionality, it infers runtime bounds and applies recurrence solving techniques to program parts independently, i.e., in contrast to [27] we use recurrence solving instead of size bounds. The reason is that **KoAT**'s size bounds are an over-approximation and hence unsound for the inference of lower bounds, where under-approximations are needed. The resulting polynomial, exponential, or even infinite lower bounds complement the corresponding upper bounds computed by state-of-the-art tools like, e.g., **CoFloCo** [48, 50], **KoAT** [27], **Loopus** [114], or **PUBS** [3].

The technique presented in Chapter 5 overcomes one of the main restrictions of [27], namely its limited support for recursion. To this end, we introduce a modular bottom-up analysis, which first infers runtime and size bounds for (mutually recursive sets of) “auxiliary functions”, i.e., leaves of the call graph (resp. SCCs of the call graph without outgoing edges), and then eliminates calls to these functions from the analyzed program. In this way, inner nodes of the call graph become leaves and hence they can be analyzed in the next step. To enable modularity, the presented technique is restricted to arithmetic on natural numbers. In this way, we avoid dealing with non-monotonicity which allows us to analyze recursive programs in a compositional way (see Section 5.6 for a discussion of an extension to full integer arithmetic). As the auxiliary functions which are analyzed independently are turned into ITSs, all existing tools and techniques for the analysis of ITSs can be reused in our setting. In other words, the technique from Chapter 5 is completely independent from the underlying complexity analysis tool for ITSs and hence it can be used to lift any complexity analysis technique with little or no support for recursion like

CHAPTER 3. INTRODUCTION

[8, 27, 71, 114] to recursive programs.

Lower Bounds for Integer Transition Systems

In this chapter we consider *Integer Transition Systems* (ITSs), cf. Section 4.1. So all computations carried out by the rewrite systems considered in this chapter operate on integers, i.e., ITSs do not have any notion of “data structures”. Moreover, they disallow nesting of function symbols, i.e., the result computed by one function cannot be passed to another function.

So in the context of program verification (cf. Section 1.2), this chapter introduces the first technique for the inference of lower bounds on the complexity of integer programs with restricted support for recursion. While we also introduce the first techniques for the inference of lower bounds for programs operating on data structures in Part III, the combination of both integers and data structures as well as an extension to full recursion is left to future work.

In Sections 4.2 and 4.3, we consider a restriction of ITSs, namely *linear ITSs*. They correspond to iterative (resp. tail-recursive) integer programs and techniques to infer upper bounds on the complexity of linear ITSs or equivalent formalisms have been widely studied [8, 27, 71, 114]. However, so far there were no techniques to prove *lower* bounds on the runtime complexity of linear ITSs. The first technique to infer such lower bounds is presented in Sections 4.2 and 4.3.

In Section 4.2, we show how to use a variation of classical ranking functions which we call *metering functions* to under-estimate the number of iterations of a simple loop, i.e., a loop without nested loops or branching. Then, we present a framework to simplify linear ITSs iteratively in Section 4.3. It transforms arbitrary linear ITSs (with branching and sequences of possibly nested loops) to ITSs with only simple loops. Moreover, it eliminates simple loops by (under-)approximating their effect using a combination of metering functions and recurrence solving. In this way, linear ITSs are transformed to *simplified ITSs* without loops which directly give rise to bounds of the form

$$\varphi \implies \text{dh}(f(x_1, \dots, x_n)) \geq a, \quad (4.1)$$

i.e., they witness that for all models σ of some constraint φ , the derivation

CHAPTER 4. LOWER BOUNDS FOR ITSs

height of $f(x_1, \dots, x_n)\sigma$ is at least $a\sigma$.

Nowadays, there is also tool support for the inference of upper bounds on the runtime complexity of non-linear ITSs [3, 27, 48, 50]. Consequently, we also study an extension of our technique to non-linear ITSs in Section 4.4, i.e., Sections 4.2 to 4.4 allow us to transform arbitrary ITSs into simplified ITSs in order to obtain bounds like (4.1).

However, bounds like (4.1) are sometimes hard to grasp, as φ as well as a can be quite complex. Hence, we introduce techniques which allow us to infer *asymptotic* lower bounds from simplified ITSs in Sections 4.5 and 4.6. In contrast to bounds like (4.1), asymptotic bounds directly provide an intuitive understanding of the complexity of an ITS. Section 4.5 introduces a calculus to compute asymptotic bounds by repeatedly simplifying a *limit problem*, an abstraction of the constraint φ in (4.1) which allows us to focus on φ 's limit behavior, i.e., on the question if φ is satisfied “for large enough inputs”. Section 4.6 shows how limit problems can be encoded to quantifier free first-order formulas with integer arithmetic in many cases such that we can benefit from the power of SMT solvers instead of applying the rules of our calculus from Section 4.5 heuristically. Note that the calculus from Section 4.5 can simplify limit problems such that our SMT encoding becomes applicable and our SMT encoding can be integrated seamlessly into the calculus from Section 4.5, i.e., both techniques complement each other.

Finally, we discuss related work in Section 4.7, evaluate our implementation LoAT in Section 4.8, and conclude in Section 4.9.

4.1 Program Model

We now introduce our program model, i.e., (linear) ITSs. Such ITSs have built-in support for integer arithmetic via *arithmetic expressions*.

Definition 4.1 (Arithmetic Expressions). Let \mathcal{V} be a set of variables and let $\Sigma_{\mathbb{Z}} = \{+, -, /, \cdot, \wedge\} \cup \mathbb{Z}$. $\mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ is the set of all *arithmetic expressions*. To ease notation, we use infix notation for symbols from $\Sigma_{\mathbb{Z}} \setminus \mathbb{Z}$ and we often identify arithmetic expressions t with $\mathcal{V}(t) = \{x_1, \dots, x_n\}$ with the corresponding function $\lambda x_1, \dots, x_n. t$.

Thus, $7 + 3$ and 2^x are arithmetic expressions and if we call the latter a “function”, then we mean the function $\lambda x. 2^x$. While Definition 4.1 clarifies which terms are syntactically legal arithmetic expressions in our program model, the following definition clarifies their semantics.

Definition 4.2 (Evaluating Arithmetic Expressions). For each symbol $\circ \in \{+, -, /, \cdot, \wedge\}$, let \bullet be the usual interpretation of \circ (i.e., addition if \circ is $+$, exponentiation if \circ is \wedge , etc.). We define the function $\llbracket \cdot \rrbracket : \mathcal{T}(\Sigma_{\mathbb{Z}}) \rightarrow \mathbb{Q}$ by

$$\llbracket t \rrbracket = t \quad \text{if } t \in \mathbb{Z} \quad \text{and} \quad \llbracket t \rrbracket = \llbracket t_1 \rrbracket \bullet \llbracket t_2 \rrbracket \quad \text{if } t = t_1 \circ t_2.$$

Hence, we have, e.g., $\llbracket 7 + 3 \rrbracket = 10$, i.e., the semantics of arithmetic expressions in our program model mimics their usual semantics in mathematics. Thus, unless necessary we do not distinguish between syntactic equality and semantic equivalence of arithmetic expressions, i.e., we write “ $7 + 3 = 10$ ” instead of “ $\llbracket 7 + 3 \rrbracket = 10$ ” etc. Note that the codomain of $\llbracket \cdot \rrbracket$ is \mathbb{Q} , i.e., $\llbracket 1/2 \rrbracket = \frac{1}{2}$. However, as the variables of an ITS range over \mathbb{Z} , the integer programs considered in this chapter terminate whenever such non-integer values are computed, cf. Definition 4.4.

To restrict the control flow of ITSs, we use *constraints* which have *integer substitutions* as models.

Definition 4.3 (Integer Substitutions, Constraints). A substitution σ such that $\llbracket x\sigma \rrbracket \in \mathbb{Z}$ for each $x \in \mathcal{V}$ is called an *integer substitution*. As for (non-integer) substitutions, we sometimes denote integer substitutions by finite sets of key-value pairs $\{\mathbf{x}/\mathbf{t}\}$. Then each $x \in \mathcal{V} \setminus \mathbf{x}$ is implicitly mapped to 0. A *constraint* φ is a finite conjunction of (in-)equalities over arithmetic expressions and $\mathcal{Fml}(\mathcal{V})$ is the set of all constraints over \mathcal{V} . We lift substitutions to constraints in the obvious way and write $\mathcal{V}(\varphi)$ for the set of all variables occurring in φ . We write $\sigma \models \varphi$ if the integer substitution σ is a model of φ .

Note that σ may instantiate variables with arithmetic expressions (as opposed to just numbers). Thus, we have, e.g., $\{x/1 + 1\} \models x = 2$. This allows ITSs to evaluate program states like $f(1 + 1)$ without normalizing the subterm $1 + 1$ to

CHAPTER 4. LOWER BOUNDS FOR ITSs

its value $\llbracket 1 + 1 \rrbracket = 2$ beforehand and thus simplifies the definition of the *Integer Transition Relation*, the transition relation of ITSs (cf. Definition 4.6).

Definition 4.4 (Integer Transition System (ITS)). Let Σ be a finite signature with $\Sigma \cap \Sigma_{\mathbb{Z}} = \emptyset$. An ITS rule α over Σ is of the form $\mathbf{f}(\mathbf{x}) \xrightarrow{c} r [\varphi]$ where $\mathbf{f} \in \Sigma$, \mathbf{x} is a vector of pairwise different variables, $c \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$, $r \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{Z}}, \mathcal{V})$, $\varphi \in \mathcal{Fml}(\mathcal{V})$, and $r \supseteq \mathbf{g}(\mathbf{t})$ with $\mathbf{g} \in \Sigma$ implies $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$. An *Integer Transition System* (ITS) over Σ is a set of ITS rules over Σ .

The *left-hand side* of α is $\text{lhs}(\alpha) = \mathbf{f}(\mathbf{x})$, its *right-hand side* is $\text{rhs}(\alpha) = r$, its *root* is $\text{root}(\alpha) = \mathbf{f}$, its *cost* is $\text{cost}(\alpha) = c$, and its *guard* is $\text{guard}(\alpha) = \varphi$. Moreover, we define $\mathcal{V}(\alpha) = \mathcal{V}(\text{lhs}(\alpha)) \cup \mathcal{V}(\text{rhs}(\alpha)) \cup \mathcal{V}(\text{cost}(\alpha)) \cup \mathcal{V}(\text{guard}(\alpha))$.

The number of function symbols from Σ in r is called the *degree* of α . A rule is *linear* if it has at most degree 1. An ITS is linear if each of its rules is linear.

So while the right-hand side of an ITS rule may contain several function symbols from Σ , these function symbols must not be nested (which is enforced by the condition “ $r \supseteq \mathbf{g}(\mathbf{t})$ with $\mathbf{g} \in \Sigma$ implies $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ ”). Thus, the result computed by one function cannot be passed to another one as argument. Hence, ITSs can model tail-recursive functions (where function calls can be inlined), but non-tail recursive functions can only be modeled if the results of calls to such functions are irrelevant and thus can be discarded.

Note that the question if a rule $\ell \xrightarrow{c} r [\varphi]$ is linear is not related to linearity of ℓ or r (cf. Definition 2.1). The idea of linear ITSs is rather that they are restricted to *linear recursion*, i.e., they cannot express non-linear recursion as in the rule

$$\text{fib}(x) \xrightarrow{1} \text{fib}(x-1) + \text{fib}(x-2) [x > 1]. \quad (4.2)$$

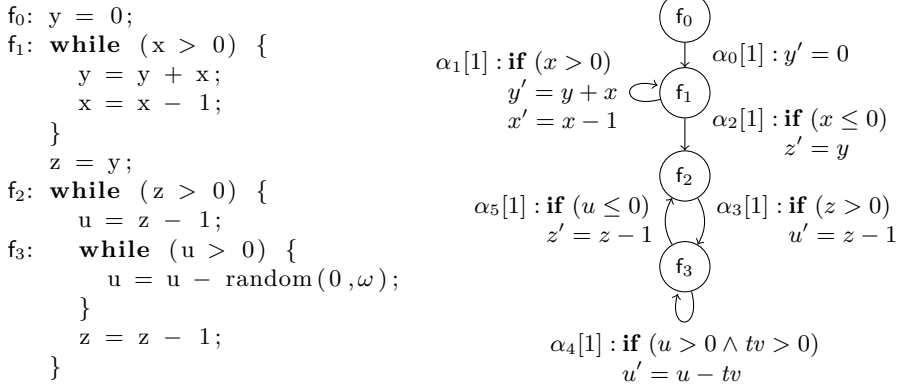
Example 4.5 (Non-Linear ITS Rules). The recursive rule to compute the Fibonacci numbers (4.2) is a valid ITS rule, but not a linear ITS rule, as the right hand side contains more than one function symbol. However, a rule like

$$\text{ack}(m, n) \xrightarrow{1} \text{ack}(m-1, \text{ack}(m, n-1)) [m > 0 \wedge n > 0]$$

is not a valid ITS rule. The reason is that the root of the right-hand side is a function symbol from Σ , but its second argument $\text{ack}(m, n-1)$ is not an arithmetic expression.

Figure 4.1c shows an example for a linear ITS, i.e., here every right-hand side just contains a single symbol from Σ . It corresponds to the pseudo-code in Figure 4.1a, where $\text{random}(x, y)$ returns a random integer m with $x < m < y$. The following definition clarifies how to evaluate ITSs.

4.1. PROGRAM MODEL



(a) Example Integer Program (b) Example ITS – Graph Representation

| | | | | |
|--------------|-------------------|-------------------|---------------------------|-------------------------|
| $\alpha_0 :$ | $f_0(x, y, z, u)$ | $\xrightarrow{1}$ | $f_1(x, 0, z, u)$ | |
| $\alpha_1 :$ | $f_1(x, y, z, u)$ | $\xrightarrow{1}$ | $f_1(x - 1, y + x, z, u)$ | $[x > 0]$ |
| $\alpha_2 :$ | $f_1(x, y, z, u)$ | $\xrightarrow{1}$ | $f_2(x, y, y, u)$ | $[x \leq 0]$ |
| $\alpha_3 :$ | $f_2(x, y, z, u)$ | $\xrightarrow{1}$ | $f_3(x, y, z, z - 1)$ | $[z > 0]$ |
| $\alpha_4 :$ | $f_3(x, y, z, u)$ | $\xrightarrow{1}$ | $f_3(x, y, z, u - tv)$ | $[u > 0 \wedge tv > 0]$ |
| $\alpha_5 :$ | $f_3(x, y, z, u)$ | $\xrightarrow{1}$ | $f_2(x, y, z - 1, u)$ | $[u \leq 0]$ |

(c) Example ITS – Rule Representation

Figure 4.1: Different Representations of Integer Programs

Definition 4.6 (Integer Transition Relation). Let \mathcal{P} be an ITS. We have $s \xrightarrow{\ell}_{\mathcal{P}} t$ if there is a context C , a rule $\ell \xrightarrow{c} r [\varphi] \in \mathcal{P}$, and an integer substitution σ such that $C[\ell\sigma] = s$, $C[r\sigma] = t$, $\sigma \models \varphi$, and $\llbracket c\sigma \rrbracket = \ell$. We write $s \xrightarrow{\ell}_{\alpha} t$ instead of $s \xrightarrow{\ell}_{\mathcal{P}} t$ if \mathcal{P} is the singleton set containing α .

So the integer transition relation is clearly a weighted relation (cf. Definition 2.10). Using the rules from Figure 4.1c, we have, e.g.,

$$\begin{aligned}
f_0(3, 2, 1, 0) &\xrightarrow{1}_{\alpha_0} f_1(3, 0, 1, 0) \\
&\xrightarrow{1}_{\alpha_1} f_1(3 - 1, 0 + 3, 1, 0) \\
&\xrightarrow{1}_{\alpha_1} f_1(3 - 1 - 1, 0 + 3 + 3 - 1, 1, 0) \\
&\xrightarrow{1}_{\alpha_1} \dots
\end{aligned}$$

So as mentioned before, the integer transition relation does not evaluate arithmetic expressions. Intuitively, the result of evaluating a term t with an ITS is obtained by first computing a normal form s of t and then evaluating all arithmetic expressions in s using $\llbracket \cdot \rrbracket$. The reason for not applying $\llbracket \cdot \rrbracket$ eagerly after every $\rightarrow_{\mathcal{P}}$ -step is that in this way $t \rightarrow_{\alpha} s$ and $\tau \in \text{pos}(\text{rhs}(\alpha))$ implies $\pi.\tau \in \text{pos}(s)$ where π is the position where the rewrite step $t \rightarrow_{\alpha} s$ takes place. This property greatly simplifies many proofs which argue about both positions and rewrite sequence. In examples, however, we sometimes simplify arithmetic

CHAPTER 4. LOWER BOUNDS FOR ITSs

expressions for the sake of readability.

Note that the derivation height of a term w.r.t. an ITS can be unbounded even if the ITS terminates.

Example 4.7 (Unbounded Costs without Non-Termination). Consider the ITS rule $f(x) \xrightarrow{y} f(x-1) [x > 0]$. By Definition 4.6, *every* integer substitution σ with $\sigma \models x > 0$ can be used to rewrite $f(x\sigma)$ to $f(x\sigma - 1)$. Thus, for any $n \in \mathbb{N}$ we have $f(1) \xrightarrow{n} f(0)$ using the integer substitution $\sigma_n = \{x/1, y/n\}$. Therefore we have $\text{dh}(f(1)) = \omega$.

Definition 4.4 allows ITSs which produce non-integer intermediate values like $f(x) \rightarrow f(x/2)$. While such evaluations get stuck immediately (e.g., we cannot rewrite $f(1/2)$ to $f(1/4)$ as $\{x/(1/2)\}$ is not an integer substitution), some of the presented techniques assume that the analyzed ITS just computes integer values. Hence, throughout this chapter we restrict ourselves to *well-formed* ITSs.

Definition 4.8 (Well-Formed ITS). An ITS rule α is *well formed* if $\text{rhs}(\alpha) \geq f(\mathbf{t})$ with $f \in \Sigma$ implies $\llbracket \mathbf{t}\sigma \rrbracket \subseteq \mathbb{Z}$ for all integer substitutions σ with $\sigma \models \text{guard}(\alpha)$. An ITS is well formed if each of its rules is well formed.

To ensure that the analyzed ITS \mathcal{P}_0 is initially well formed, we just allow addition, subtraction, and multiplication in \mathcal{P}_0 , but we disallow division and exponentiation. While our approach relies on several program transformations, i.e., the initial ITS \mathcal{P}_0 is usually transformed to several other ITSs $\mathcal{P}_1, \mathcal{P}_2, \dots$ which may contain division and exponentiation, these transformations preserve well-formedness.

When analyzing the complexity of an ITS, then one is usually interested in the cost of evaluating a term of the form $f(\mathbf{n})$ where $f \in \Sigma$ and $\mathbf{n} \subseteq \mathbb{Z}$. We call such terms *int-basic*.

Definition 4.9 (Int-Basic Terms). Let Σ be a signature and let $f \in \Sigma$. We define

$$\mathcal{T}_{\text{basic}}(f) = \{f(\mathbf{n}) \mid \mathbf{n} \subseteq \mathbb{Z}, \text{ar}_{\Sigma}(f) = \text{len}(\mathbf{n})\}.$$

Then

$$\mathcal{T}_{\text{basic}}(\Sigma) = \bigcup_{f \in \Sigma} \mathcal{T}_{\text{basic}}(f)$$

is the set of all *int-basic terms* over Σ .

As standard for ITSs, we always assume that Σ contains a *canonical start symbol* f_0 throughout this chapter and we are only interested in program runs whose start term is from $\mathcal{T}_{\text{basic}}(f_0)$. Note that this is not a restriction, as we can simulate several start symbols f_1, \dots, f_n by adding corresponding rules from f_0 to f_1, \dots, f_n .

4.1. PROGRAM MODEL

Thus, we now fixed the transition relation (Definition 4.6) and the start terms we are concerned with. So the last missing piece in order to arrive at complexity problems for ITSs is a suitable size measure.

Definition 4.10 ($\|\cdot\|_i$). Let $f(\mathbf{n})$ be an int-basic term. We define

$$\|f(\mathbf{n})\|_i = \sum |\mathbf{n}|.$$

Now all the ingredients to define the *canonical complexity problem* of an ITS are at hand.

Definition 4.11 (Canonical Complexity Problem). Let \mathcal{P} be an ITS over Σ . The *canonical complexity problem* of \mathcal{P} is $cp(\mathcal{P}) = (\mathcal{T}_{\text{basic}}(f_0), \rightarrow_{\mathcal{P}}, \|\cdot\|_i)$.

W.l.o.g., we assume that the start symbol f_0 does not occur on right-hand sides. Otherwise, one can add a fresh start symbol f'_0 and connect it with f_0 .

We also assume that all function symbols in Σ have the same arity. Otherwise, one can construct a variant of \mathcal{P} where additional unused arguments are added to each function symbol whose arity is not maximal. Moreover, we assume that the left-hand sides of \mathcal{P} only differ in their root symbols, i.e., the argument lists are equal (e.g., in Figure 4.1c, the variables on the left-hand sides are consistently named x, y, z, u). Otherwise, one can rename variables accordingly without affecting the relation $\rightarrow_{\mathcal{P}}$. Those variables which occur on left-hand sides are called *program variables* and all other variables are called *temporary*. \mathcal{PV} resp. \mathcal{TV} is the set of all program variables resp. temporary variables. So in Figure 4.1c, we have $\mathcal{PV} = \{x, y, z, u\}$ and $\mathcal{TV} = \{tv\}$.

Furthermore, we assume that every right-hand side is of the form

$$\sum_{i=1}^m f_i(\mathbf{t}_i) \text{ where } m > 0 \text{ and } f_1, \dots, f_m \in \Sigma. \quad (4.3)$$

Clearly, each ITS rule can be transformed to this form without affecting the runtime complexity of the overall ITS. If a right-hand side does not contain any function symbols (i.e., it is an arithmetic expression), then it can be replaced by $\text{sink}(0, \dots, 0)$ where sink is a fresh function symbol.

We now focus on linear ITSs until we consider non-linear ITSs in Section 4.4. For each linear rule

$$\alpha = f(\mathbf{x}) \rightarrow g(\mathbf{t}) \ [\varphi],$$

its *update* is $\text{update}(\alpha) = \{\mathbf{x}/\mathbf{t}\}$ and its *target* is $\text{target}(\alpha) = g$.

Linear ITSs can be represented as directed graphs where each node corresponds to a function symbol and each edge corresponds to a rule. Figure 4.1b shows the graphical representation of Figure 4.1c, where we write the costs of a rule in $[]$ next to its name and represent the updates by imperative commands. We use x to refer to the value of the variable x before the update and x' to refer

CHAPTER 4. LOWER BOUNDS FOR ITSs

to x 's value after the update. As it nicely exposes the control flow, we often use the graphical representation for ITSs with nested loops and branching in examples.

In Figure 4.1b, the loop at f_1 sets y to a value that is quadratic in x . Thus, the loop at f_2 is executed quadratically often where in each iteration, the inner loop at f_3 may also be repeated quadratically often. Thus, the ITS's runtime complexity is a polynomial of degree 4 in x . Our technique can infer such lower bounds automatically.

Our goal is to find a lower bound on the runtime complexity of an ITS \mathcal{P} which is as precise as possible (i.e., a lower bound which is, e.g., unbounded, exponential, or a polynomial of a degree as high as possible). For all terms $f_0(x, y, z, u)$ with $x > 1$, our method will detect that the derivation height of the ITS in Figure 4.1 is at least $\frac{1}{8}x^4 + \frac{1}{4}x^3 + \frac{7}{8}x^2 + \frac{7}{4}x$. From this concrete lower bound, our approach will infer that the asymptotic runtime complexity of the ITS is in $\Omega(n^4)$.

4.2 Estimating the Number of Iterations

We now show how to under-estimate the number of possible loop iterations for linear ITSs. We only consider *simple loops* of the form

$$\alpha = f(\mathbf{x}) \rightarrow f(\mathbf{x})\eta \ [\varphi]$$

where η is the update of α and show how to transform more complex loops into simple loops in Section 4.3. So our goal is to infer an arithmetic expression b such that for all integer substitutions σ with $\sigma \models \varphi$, there is an integer substitution σ' with

$$f(\mathbf{x})\sigma \rightarrow_{\alpha}^{\lceil b\sigma \rceil} f(\mathbf{x})\sigma'.$$

Here, as usual, $\lceil x \rceil$ is the smallest integer n with $n \geq x$ (and, similarly, $\lfloor x \rfloor$ is the largest integer n with $n \leq x$).

To find such estimations, we use an adaptation of ranking functions [8, 19, 25, 109] which we call *metering functions*.

Definition 4.12 (Ranking Function). We say that $b \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ is a *ranking function* for a simple loop α with $\text{update}(\alpha) = \eta$ if the following conditions hold:

$$\text{guard}(\alpha) \implies b > 0 \tag{4.4}$$

$$\text{guard}(\alpha) \implies b\eta \leq b - 1 \tag{4.5}$$

So e.g., x is a ranking function for α_1 in Figure 4.1. If $\mathcal{TV}(\alpha) = \emptyset$, then for any integer substitution σ , $b\sigma$ *over-estimates* the number of repetitions of the loop α .¹ (4.5) ensures that $b\sigma$ decreases at least by 1 in each loop iteration, and (4.4) requires that $b\sigma$ is positive whenever the loop can be executed.

In contrast, metering functions are *under-estimations* for the maximal number of repetitions of a simple loop.

Definition 4.13 (Metering Function). We call $b \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ a *metering function* for a simple loop α with $\text{update}(\alpha) = \eta$ if the following conditions hold:

$$\neg \text{guard}(\alpha) \implies b \leq 0 \tag{4.6}$$

$$\text{guard}(\alpha) \implies b\eta \geq b - 1 \tag{4.7}$$

Here, (4.7) ensures that $b\sigma$ decreases at most by 1 in each loop iteration, and (4.6) requires that $b\sigma$ is non-positive if the loop cannot be executed. Thus, the loop can be executed *at least* $b\sigma$ times (i.e., $b\sigma$ is an under-estimation).

For the loop α_1 in Figure 4.1, x is also a valid metering function: (4.6) requires

¹Without requiring $\mathcal{TV}(\alpha) = \emptyset$, $x - tv$ were a ranking function for the non-terminating simple loop $f(x) \rightarrow f(x-1) \ [x > tv]$.

CHAPTER 4. LOWER BOUNDS FOR ITSS

$\neg x > 0 \implies x \leq 0$ and (4.7) requires $x > 0 \implies x - 1 \geq x - 1$. While x is a metering *and* a ranking function, $\frac{1}{2} \cdot x$ is a metering, but not a ranking function for α_1 . Similarly, x^2 is a ranking, but not a metering function for α_1 . Theorem 4.14 states that a simple loop α can be executed at least $\lceil b\sigma \rceil$ times when starting with $\text{lhs}(\alpha)\sigma$ if b is a metering function for α .

Theorem 4.14 (Metering Functions are Under-Estimations). *Let b be a metering function for a well-formed simple loop α with $\text{update}(\alpha) = \eta$. Then b under-estimates α , i.e., for all integer substitutions σ with $\sigma \models \text{guard}(\alpha)$*

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^{\lceil b\sigma \rceil}\sigma$$

is a rewrite sequence which σ -preserves \mathcal{TV} , i.e., the temporary variables are instantiated according to σ in every step.

Proof. For any integer substitution σ , let $m_{\sigma} \in \mathbb{N} \cup \{\omega\}$ be the maximal number such that $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha}^{m_{\sigma}} \text{lhs}(\alpha)\eta^{m_{\sigma}}\sigma$ is a rewrite sequence that σ -preserves \mathcal{TV} . So the loop α can be executed m_{σ} times without changing the temporary variables when starting with $\text{lhs}(\alpha)\sigma$.

If $m_{\sigma} = \omega$, then

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^2\sigma \rightarrow_{\alpha} \dots$$

is an infinite rewrite sequence that σ -preserves \mathcal{TV} and thus the claim is trivial. For the case $m_{\sigma} \neq \omega$, we use induction on m_{σ} . The base case $m_{\sigma} = 0$ is trivial. For the induction step, note that $\sigma \models \text{guard}(\alpha)$ implies that

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \quad \sigma\text{-preserves } \mathcal{TV}. \quad (4.8)$$

Case 1. $\eta \diamond \sigma \models \text{guard}(\alpha)$

Note that $m_{\sigma} > m_{\eta \diamond \sigma}$. Thus, by the induction hypothesis

$$\text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^2\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^{\lceil b\eta\sigma \rceil}\eta\sigma \quad (4.9)$$

$\eta \diamond \sigma$ -preserves \mathcal{TV} . Since $(\eta \diamond \sigma)|_{\mathcal{TV}} = \sigma|_{\mathcal{TV}}$, this implies that (4.9) σ -preserves \mathcal{TV} . With (4.8) we obtain that

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^{\lceil b\eta\sigma \rceil + 1}\sigma$$

σ -preserves \mathcal{TV} . This proves the theorem, since (4.7) implies $\lceil b\eta\sigma \rceil + 1 \geq \lceil b\sigma \rceil$.

Case 2. $\eta \diamond \sigma \not\models \text{guard}(\alpha)$

Then (4.6) implies $b\eta\sigma \leq 0$ and thus (4.7) implies $\lceil b\sigma \rceil \leq 1$. Hence, (4.8) proves the theorem. \square

4.2. ESTIMATING THE NUMBER OF ITERATIONS

Our implementation builds upon a well-known transformation based on Farkas' Lemma [25, 109] to find *linear* metering functions. The basic idea is to search for coefficients of a linear template polynomial b such that (4.6) and (4.7) hold for all possible instantiations of the variables $\mathcal{V}(\alpha)$. In addition to (4.6) and (4.7), we also require (4.4) to avoid trivial solutions like $b = 0$. Here, the coefficients of b are existentially quantified, while the variables from $\mathcal{V}(\alpha)$ are universally quantified. As in [25, 109], eliminating the universal quantifiers using Farkas' Lemma allows us to use standard SMT solvers to search for b 's coefficients.

When searching for a metering function for a simple loop α with $\text{update}(\alpha) = \eta$, one can omit constraints from $\text{guard}(\alpha)$ that are irrelevant for α 's termination. So if $\text{guard}(\alpha)$ is $\varphi \wedge \psi$ and $\text{guard}(\alpha) \implies \psi\eta$, then it suffices to find a metering function b for $\alpha' = \text{lhs}(\alpha) \rightarrow \text{rhs}(\alpha)$ $[\varphi]$.

Lemma 4.15 (Irrelevant Constraints). *Let α be a simple loop such that $\text{update}(\alpha) = \eta$ and $\text{guard}(\alpha) = \varphi \wedge \psi$ where $\text{guard}(\alpha)$ implies $\psi\eta$. Moreover, let b be an arithmetic expression which under-estimates α' where α' is like α , but $\text{guard}(\alpha') = \varphi$. Then b under-estimates α .*

Proof. Let σ be an integer substitution such that $\sigma \models \text{guard}(\alpha)$. For all $n \in \mathbb{N}$, we prove: If

$$\text{lhs}(\alpha')\sigma \rightarrow_{\alpha'} \text{lhs}(\alpha')\eta\sigma \rightarrow_{\alpha'} \dots \rightarrow_{\alpha'} \text{lhs}(\alpha')\eta^n\sigma$$

is a rewrite sequence that σ -preserves \mathcal{TV} , then

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^n\sigma$$

is a rewrite sequence that σ -preserves \mathcal{TV} . Then the claim follows immediately. We use induction on n . The cases $n = 0$ and $n = 1$ are trivial. If $n > 1$, then by the induction hypothesis

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^{n-1}\sigma$$

is a rewrite sequence that σ -preserves \mathcal{TV} . Hence, $\eta^{n-2} \diamond \sigma \models \text{guard}(\alpha)$. Since $\text{guard}(\alpha) \implies \psi\eta$, we obtain $\eta^{n-2} \diamond \sigma \models \psi\eta$ and thus $\eta^{n-1} \diamond \sigma \models \psi$. Since $\text{lhs}(\alpha')\eta^{n-1}\sigma \rightarrow_{\alpha'} \text{lhs}(\alpha')\eta^n\sigma$ implies $\eta^{n-1} \diamond \sigma \models \varphi$, we get $\eta^{n-1} \diamond \sigma \models \text{guard}(\alpha)$. Thus $\text{lhs}(\alpha)\eta^{n-1}\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^n\sigma$ is a rewrite step which $\eta^{n-1} \diamond \sigma$ -preserves \mathcal{TV} . Since $(\eta^{n-1} \diamond \sigma)|_{\mathcal{TV}} = \sigma|_{\mathcal{TV}}$, it is also a rewrite step that σ -preserves \mathcal{TV} . Thus,

$$\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha} \dots \rightarrow_{\alpha} \text{lhs}(\alpha)\eta^n\sigma$$

is a rewrite sequence that σ -preserves \mathcal{TV} . □

So if

$$\alpha = f(x, y) \rightarrow f(x + 1, y) \ [x < y \wedge 0 < y],$$

CHAPTER 4. LOWER BOUNDS FOR ITSS

we can consider

$$\alpha' = f(x, y) \rightarrow f(x + 1, y) \ [x < y]$$

instead. While α only has complex metering functions like $\min(y - x, y)$, α' has the linear metering function $y - x$.

Example 4.16 (Unbounded Loops). Let α be a simple loop whose update is η . If $\text{guard}(\alpha) \implies \text{guard}(\alpha)\eta$ and hence the *whole* guard can be omitted, then α does not terminate. So for

$$\mathcal{P} = \{f_0(x, y) \xrightarrow{1} f(x, y), \alpha\}$$

with

$$\alpha = f(x, y) \xrightarrow{y} f(x + 1, y) \ [0 < x],$$

we can omit $0 < x$ since $0 < x \implies 0 < x + 1$. Hence, a fresh temporary variable tv under-estimates the resulting loop $f(x, y) \xrightarrow{y} f(x + 1, y)$ and thus, tv also under-estimates α .

4.3 Simplifying ITSs

We now define processors to simplify complexity problems over linear ITSs. They are applied repeatedly until extraction of a (concrete) lower bound is straightforward. In Section 4.3.1, we show how to *accelerate* a simple loop α to a rule which is equivalent to applying α multiple times (according to a metering function for α). The resulting ITS can be simplified by *chaining* subsequent rules which may result in new simple loops, cf. Section 4.3.2. We describe a simplification strategy which alternates these steps repeatedly. In this way, we eventually obtain a *simplified* ITS without loops which directly gives rise to a concrete lower bound.

4.3.1 Accelerating Simple Loops

Consider a simple loop α with $\text{update}(\alpha) = \eta$ and $\text{cost}(\alpha) = c$. To accelerate α , we compute its *iterated* update and costs, i.e., a closed form η_{it} of η^{tv} and an under-approximation $c_{\text{it}} \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ of $\sum_{i=0}^{tv-1} c\eta^i$ for a fresh temporary variable tv . If b under-estimates α , then we add the rule

$$\text{lhs}(\alpha) \xrightarrow{c_{\text{it}}} \text{lhs}(\alpha)\eta_{\text{it}} [\text{guard}(\alpha) \wedge 0 < tv < b + 1]$$

to the ITS. It summarizes tv iterations of α , where tv is positive and bounded by $\lceil b \rceil$. It does not cover the case that α is not applied at all. Otherwise, given a loop with $x\eta = 0$ we would get

$$x\eta_{\text{it}} = \begin{cases} 0 & \text{if } tv > 0 \\ x & \text{otherwise} \end{cases}$$

i.e., the iterated update could not be expressed using arithmetic expressions from $\mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ even in quite simple cases.

Note that η_{it} and c_{it} may also contain division and exponentiation (i.e., we can also infer exponential bounds).

For $\mathcal{PV} = \{x_1, \dots, x_n\}$, the iterated update is computed by solving the recurrence equations $x^{(1)} = x\eta$ and $x^{(tv+1)} = x\eta\{x_1/x_1^{(tv)}, \dots, x_n/x_n^{(tv)}\}$ for all $x \in \mathcal{PV}$. So for the rule α_1 from Figure 4.1 we get the recurrence equations $x^{(1)} = x - 1$, $x^{(tv_1+1)} = x^{(tv_1)} - 1$, $y^{(1)} = y + x$, and $y^{(tv_1+1)} = y^{(tv_1)} + x^{(tv_1)}$. Usually, they can easily be solved using state-of-the-art recurrence solvers [18]. In our example, we obtain the closed forms $x\eta_{\text{it}} = x^{(tv_1)} = x - tv_1$ and $y\eta_{\text{it}} = y^{(tv_1)} = y + tv_1 \cdot x - \frac{1}{2} \cdot tv_1^2 + \frac{1}{2} \cdot tv_1$. While $y\eta_{\text{it}}$ contains rational coefficients, our approach ensures that η_{it} always maps integers to integers. Thus, our technique to accelerate loops preserves well-formedness. We proceed similarly for the iterated cost of a rule, where we may under-approximate the solution of the recurrence equations $c^{(1)} = c$ and $c^{(tv+1)} = c^{(tv)} + c\{x_1/x_1^{(tv)}, \dots, x_n/x_n^{(tv)}\}$. For α_1 in Figure 4.1, we get $c^{(1)} = 1$ and $c^{(tv_1+1)} = c^{(tv_1)} + 1$ which leads to the closed form $c_{\text{it}} = c^{(tv_1)} = tv_1$.

CHAPTER 4. LOWER BOUNDS FOR ITSS

Theorem 4.17 (Loop Acceleration). *Let \mathcal{P} be a well-formed ITS, let $\alpha \in \mathcal{P}$ be a simple loop with $\text{update}(\alpha) = \eta$ and $\text{cost}(\alpha) = c$, let tv be a fresh temporary variable, and let b be an arithmetic expression which under-estimates α . Moreover, let $x\eta_{it} = x\eta^{tv}$ for all $x \in \mathcal{PV}$, let $c_{it} \leq \sum_{i=0}^{tv-1} c\eta^i$, and let*

$$\mathcal{P}' = \mathcal{P} \cup \{\text{lhs}(\alpha) \xrightarrow{c_{it}} \text{lhs}(\alpha)\eta_{it} \mid \text{guard}(\alpha) \wedge 0 < tv < b + 1\}.$$

Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is equivalent.

Proof. Soundness for upper bounds is trivial. It remains to show that \mathcal{P}' is well formed and that the processor is sound for lower bounds. Let

$$\alpha_{it} = \text{lhs}(\alpha) \xrightarrow{c_{it}} \text{lhs}(\alpha)\eta_{it} \mid \text{guard}(\alpha) \wedge 0 < tv < b + 1$$

and let σ be an integer substitution such that $\sigma \models \text{guard}(\alpha_{it})$, i.e., we have $\text{lhs}(\alpha)\sigma \xrightarrow{c_{it}\sigma}_{\alpha_{it}} \text{lhs}(\alpha)\eta_{it}\sigma$. Note that $\sigma \models 0 < tv < b + 1$ implies $\sigma \models 0 < tv \leq [b]$. Moreover, we have $\sigma \models \text{guard}(\alpha)$, and thus by Theorem 4.14,

$$\text{lhs}(\alpha)\sigma \xrightarrow{c\sigma}_{\alpha} \text{lhs}(\alpha)\eta\sigma \xrightarrow{c\eta\sigma}_{\alpha} \dots \xrightarrow{c\eta^{(tv-1)}\sigma}_{\alpha} \text{lhs}(\alpha)\eta^{(tv)}\sigma \quad (4.10)$$

is a rewrite sequence that σ -preserves \mathcal{TV} .

Claim 1. \mathcal{P}' is well formed.

We have to show $\llbracket x\eta_{it}\sigma \rrbracket \in \mathbb{Z}$ for all $x \in \mathcal{PV}$. We have $\eta^{(tv-1)}\sigma \diamond \sigma \models \text{guard}(\alpha)$ by (4.10) and thus $\llbracket x\eta^{(tv)}\sigma \rrbracket \in \mathbb{Z}$ since \mathcal{P} is well formed. By definition of η_{it} , we have $x\eta_{it} = x\eta^{tv}$ and thus $\llbracket x\eta_{it}\sigma \rrbracket \in \mathbb{Z}$, as desired.

Claim 2. The processor is sound for lower bounds.

It suffices to show that every evaluation step with α_{it} can be simulated using a sequence of evaluation steps with α with at least the same costs. By definition of η_{it} and c_{it} , we have $x\eta_{it} = x\eta^{tv}$ for all $x \in \mathcal{PV}$ and $c_{it} \leq \sum_{i=0}^{tv-1} c\eta^i$, as desired. Thus, the claim follows from (4.10). \square

While the fresh variable tv that represents the number of loop iterations which are summarized by an accelerated loop ranges over the integers, its upper bound $b + 1$ can be rational, as the following example shows.

Example 4.18 (Non-Integer Metering Functions). Theorem 4.17 also allows metering functions that do not map to the integers. Let

$$\mathcal{P} = \{f_0(x) \xrightarrow{1} f(x), \alpha\} \text{ with } \alpha = f(x) \xrightarrow{1} f(x - 2) \mid [0 < x].$$

4.3. SIMPLIFYING ITSs

Accelerating α with the metering function $\frac{1}{2} \cdot x$ yields

$$f(x) \xrightarrow{tv} f(x - 2 \cdot tv) \left[0 < tv < \frac{1}{2} \cdot x + 1 \right].$$

Note that $0 < tv < \frac{1}{2} \cdot x + 1$ implies $0 < x$ as tv and x range over \mathbb{Z} . Hence, $0 < x$ can be omitted in the resulting guard.

If a simple loop is under-estimated by a fresh temporary variable tv (i.e., if it is non-terminating), then the upper bound $b + 1 = tv + 1$ on the number of summarized loop iterations can take arbitrary values.

Example 4.19 (Unbounded Loops Continued). In Example 4.16, the fresh temporary variable tv under-estimates $\alpha = f(x, y) \xrightarrow{y} f(x + 1, y) [0 < x]$. The resulting accelerated loop is

$$\alpha_{it} = f(x, y) \xrightarrow{tv_1 \cdot y} f(x + tv_1, y) [0 < x \wedge 0 < tv_1 < tv + 1].$$

Since tv does not have any upper bound, the value of tv_1 is not bounded by the values of the program variables x and y .

If we cannot find a metering function or fail to obtain the closed form η_{it} or c_{it} for a simple loop α , then we can simplify α by eliminating temporary variables. To do so, we fix their values by adding suitable constraints to $\text{guard}(\alpha)$. As we are interested in witnesses for maximal computations, we use a heuristic that adds constraints $tv = a$ for temporary variables tv , where the arithmetic expression a is a suitable upper or lower bound on tv 's values, i.e., $\text{guard}(\alpha)$ implies $tv \leq a$ or $tv \geq a$, but not $tv \leq a - 1$ or $tv \geq a + 1$. This is repeated until we find constraints which allow us to apply loop acceleration. Note that adding additional constraints to $\text{guard}(\alpha)$ is *always* sound in our setting.

Theorem 4.20 (Strengthening). *Let \mathcal{P} be a well-formed ITS, let $\alpha \in \mathcal{P}$, let φ be a constraint, and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha'\}$ where α' is like α , but $\text{guard}(\alpha')$ is $\text{guard}(\alpha) \wedge \varphi$. Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is equivalent.*

Proof. Soundness for upper bounds is trivial. For lower bounds, the processor is sound since $t \xrightarrow{\mathbb{K}}_{\alpha'} t'$ implies $t \xrightarrow{\mathbb{K}}_{\alpha} t'$ as $\text{guard}(\alpha')$ implies $\text{guard}(\alpha)$. Moreover, \mathcal{P}' is trivially well-formed, since $\text{update}(\alpha') = \text{update}(\alpha)$. \square

In α_4 from Figure 4.1, $\text{guard}(\alpha_4)$ contains the constraint $tv > 0$. So $\text{guard}(\alpha_4)$ implies the bound $tv \geq 1$ since tv must be instantiated by integers. Hence, we strengthen it with the constraint $tv = 1$. Now the update $\{u/u - tv\}$ of the strengthened rule α'_4 is equivalent to $\{u/u - 1\}$, and thus, u is a metering function. So by fixing $tv = 1$, α'_4 can be accelerated similarly to α_1 .

CHAPTER 4. LOWER BOUNDS FOR ITSs

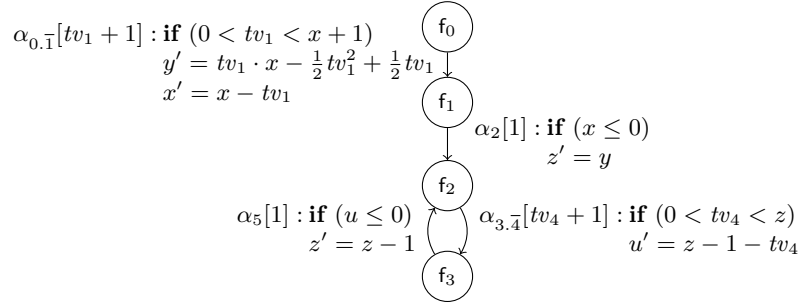
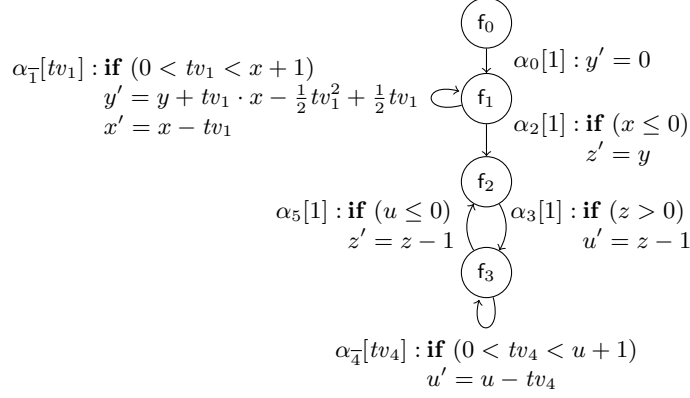


Figure 4.2: Loop Elimination

To simplify the ITS, we delete the original rules after strengthening or accelerating them, i.e., we just keep accelerated loops. For our example, we obtain the ITS in Figure 4.2a with the accelerated rules $\alpha_{\bar{1}}$ and $\alpha_{\bar{4}}$.

Theorem 4.21 (Deletion). *Let \mathcal{P} be a well-formed ITS, let $\alpha \in \mathcal{P}$, and let $\mathcal{P}' = \mathcal{P} \setminus \{\alpha\}$. Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for lower bounds.*

Proof. Since \mathcal{P} is well formed, \mathcal{P}' is trivially well formed, too. The processor is sound for lower bounds since every evaluation sequence with $\mathcal{P} \setminus \{\alpha\}$ is also an evaluation sequence with \mathcal{P} . \square

4.3.2 Chaining Rules

After trying to accelerate all simple loops of an ITS, we can *chain* subsequent rules α_1, α_2 by adding a new rule $\alpha_{1,2}$ that simulates their combination. Afterwards, the rules α_1 and α_2 can (but need not) be deleted with Theorem 4.21.

4.3. SIMPLIFYING ITSs

Theorem 4.22 (Chaining). *Let \mathcal{P} be a well formed ITS and let $\alpha_1, \alpha_2 \in \mathcal{P}$ where*

$$\begin{aligned} \alpha_1 &= f_1(\mathbf{x}) \xrightarrow{c_1} f_2(\mathbf{x})\eta_1 \quad [\varphi_1] \text{ and} \\ \alpha_2 &= f_2(\mathbf{x}) \xrightarrow{c_2} f_3(\mathbf{x})\eta_2 \quad [\varphi_2]. \end{aligned}$$

W.l.o.g., let $\mathcal{TV}(\alpha_1) \cap \mathcal{TV}(\alpha_2) = \emptyset$ (otherwise, the temporary variables in α_2 can be renamed accordingly). Moreover, let

$$\alpha_{1,2} = f_1(\mathbf{x}) \xrightarrow{c_1+c_2\eta_1} f_3(\mathbf{x})\eta_2\eta_1 \quad [\varphi_1 \wedge \varphi_2\eta_1]$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha_{1,2}\}$. Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is equivalent.

Proof. We prove the more general Theorem 4.31 in Section 4.4. □

Algorithm 1 Program Simplification

While there is a rule α with $\text{root}(\alpha) \neq f_0$:

1. Apply *Deletion* to rules whose guard is proved unsatisfiable or whose root symbol is unreachable from f_0 .
 2. While there is a non-accelerated simple loop α :
 - 2.1. Try to apply *Loop Acceleration* to α .
 - 2.2. If 2.1 failed and α uses temporary variables:
Apply *Strengthening* to α to eliminate a temporary variable.
 - 2.3. Apply *Deletion* to α .
 3. Let $S = \emptyset$.
 4. While there is an accelerated rule α :
 - 4.1. For each α' with $\text{root}(\alpha') \neq \text{target}(\alpha') = \text{root}(\alpha)$:
Apply *Chaining* to α' and α .
Add α' to S .
 - 4.2. Apply *Deletion* to α .
 5. Apply *Deletion* to each rule in S .
 6. While there is a function symbol f without simple loops but with incoming and outgoing rules (starting with symbols f with just one incoming rule):
 - 6.1. Apply *Chaining* to each pair α', α where $\text{target}(\alpha') = \text{root}(\alpha) = f$.
 - 6.2. Apply *Deletion* to each α where $\text{root}(\alpha) = f$ or $\text{target}(\alpha) = f$.
-

One goal of chaining is to eliminate all accelerated simple loops. To this end, we chain all subsequent rules α', α where α is a simple loop and α' is no simple loop. Afterwards, we delete α . Moreover, once α' has been chained with all subsequent simple loops, then we also remove α' , since its effect is now covered by the newly introduced (chained) rule. So in our example from Figure 4.2a, we chain α_0 with $\bar{\alpha}_1$ and α_3 with $\bar{\alpha}_4$. The resulting ITS is depicted in Figure 4.2b. Chaining also allows to eliminate function symbols by chaining all pairs of rules α, α' where $\text{target}(\alpha) = \text{root}(\alpha')$ and removing them afterwards. It is

CHAPTER 4. LOWER BOUNDS FOR ITSs

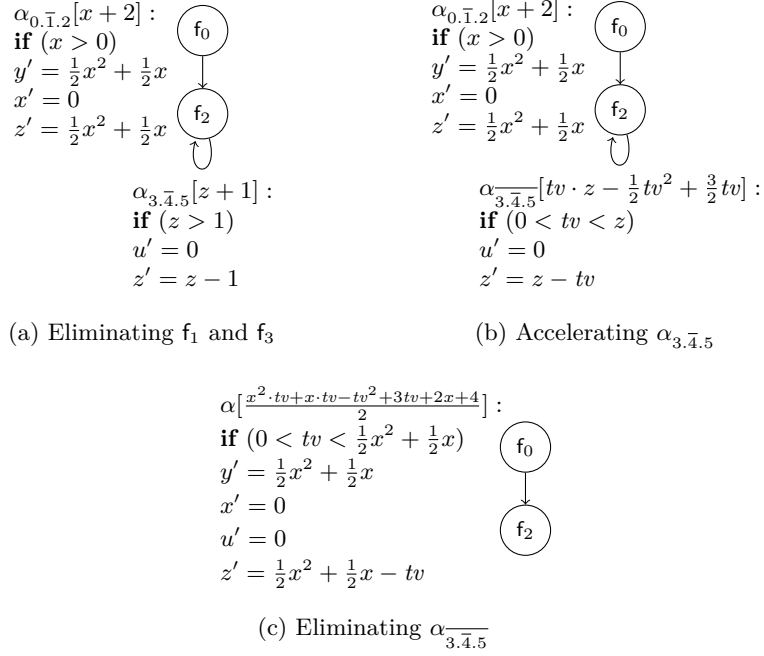


Figure 4.3: Finishing the Simplification

advantageous to eliminate symbols which are the target of just one single rule first. This heuristic takes into account which rules were the entry points of loops. So for the example in Figure 4.2b, it would avoid chaining α_5 and $\alpha_{3,\bar{4}}$ in order to eliminate f_2 . In this way, we avoid constructing chained rules that correspond to a run from the “middle” of a loop to the “middle” of the next loop iteration.

So instead of eliminating f_2 , we chain $\alpha_{0,\bar{1}}$ and α_2 as well as $\alpha_{3,\bar{4}}$ and α_5 to eliminate the function symbols f_1 and f_3 , leading to the ITS in Figure 4.3a. Here, the temporary variables tv_1 and tv_4 vanish since, before applying arithmetic simplifications, the guards of $\alpha_{0,\bar{1},2}$ resp. $\alpha_{3,\bar{4},5}$ imply $tv_1 = x$ resp. $tv_4 = z - 1$. Our overall approach for ITS simplification is shown in Algorithm 1. Of course, this algorithm is a heuristic and other strategies for the application of the processors would also be possible. The set S in the Steps 3 – 5 is needed to handle function symbols f with multiple simple loops. The reason is that each rule α' with $\text{target}(\alpha') = f$ should be chained with *each* of f 's simple loops before removing α' .

Algorithm 1 terminates: The Loop 2 terminates since each iteration either decreases the number of temporary variables in α or reduces the number of non-accelerated simple loops. In Loop 4, the number of simple loops is decreasing and for Loop 6, the number of function symbols decreases. The overall loop terminates as it reduces the number of function symbols. The reason is that the ITS does not have simple loops anymore when the algorithm reaches Step 6

4.3. SIMPLIFYING ITSs

(as simple loops where acceleration fails are deleted in Step 2.3 and accelerated loops are eliminated in Step 4). Thus, at this point there is either a function symbol f which can be eliminated or the ITS does not have a path of length 2, i.e., all rules have root f_0 .

According to Algorithm 1, in our example we go back to Step 1 and 2 and apply *Loop Acceleration* to the rule $\alpha_{3.\bar{4}.5}$. This rule has the metering function $z - 1$ and its iterated update sets u to 0 and z to $z - tv$ for a fresh temporary variable tv . To compute $\alpha_{3.\bar{4}.5}$'s iterated costs, we have to find an under-approximation for the solution of the recurrence equations $c^{(1)} = z + 1$ and $c^{(tv+1)} = c^{(tv)} + z^{(tv)} + 1$. After computing the closed form $z - tv$ of $z^{(tv)}$, the second equation simplifies to $c^{(tv+1)} = c^{(tv)} + z - tv + 1$, which results in the closed form $c_{it} = c^{(tv)} = tv \cdot z - \frac{1}{2} \cdot tv^2 + \frac{3}{2} \cdot tv$. By adding the corresponding accelerated rule and removing $\alpha_{3.\bar{4}.5}$ in Step 2.3, we obtain the ITS in Figure 4.3b. A final chaining step and deletion of $\alpha_{0.\bar{1}.2}$ and $\alpha_{\bar{3}.\bar{4}.5}$ yields the ITS in Figure 4.3c.

4.4 Non-Linear ITSs

So far, we only considered linear ITSs, i.e., ITSs where all rules have the form $f(\mathbf{x}) \rightarrow g(\mathbf{t}) [\varphi]$. We now extend our technique to non-linear ITSs, i.e., we now also consider rules where the right-hand side contains several function symbols.

Theorems 4.20 and 4.21 are trivially applicable to non-linear ITSs, i.e., we can still add additional constraints to the guard of a rule and we can still remove rules. However, Theorems 4.17 and 4.22 have to be adapted. In particular, we have to extend our notion of metering functions to non-linear rules in order to adapt Theorem 4.17. As in the case of linear ITSs, we first focus on *simple non-linear loops*, i.e., loops $f(\mathbf{x}) \xrightarrow{c} r [\varphi]$ where r contains at least two occurrences of f , but no other function symbols. Afterwards, we show how to transform more complicated loop structures to simple (possibly non-linear) loops via chaining and *partial deletion*, a new technique which is specific to non-linear ITSs.

To understand the idea of metering functions for non-linear loops, let $s_0 \rightarrow_{\mathcal{P}} \dots \rightarrow_{\mathcal{P}} s_m$ be a rewrite sequence and let π_i be the position of the rewrite step $s_i \rightarrow_{\mathcal{P}} s_{i+1}$. Then we say that the rewrite step $s_j \rightarrow_{\mathcal{P}} s_{j+1}$ is the *predecessor* of $s_i \rightarrow_{\mathcal{P}} s_{i+1}$ if j is the maximal index such that $j < i$ and $\pi_j \leq \pi_i$.

If \mathcal{P} is a linear ITS, then this notion of “predecessor” yields a string of rewrite steps: Every rewrite step except for the first one has one and only one predecessor and every rewrite step except for the last one has one and only one successor. Metering functions under-estimate the length of such strings. If \mathcal{P} is a non-linear ITSs, then this notion of “predecessor” yields a tree of rewrite steps: Every rewrite step except for the first one has one and only one predecessor, but every rewrite step may have *several* successors. The idea of the extension of metering functions to non-linear loops is to under-estimate the length of the shortest paths in such trees instead of the length of rewrite sequences. Hence, if b is a metering function for a simple non-linear loop α of degree d , then the maximal number of consecutive applications of α is in $\Omega(d^b)$.

Definition 4.23 (Metering Function for Possibly Non-Linear Loops). Let

$$\alpha = f(\mathbf{x}) \xrightarrow{c} \sum_{i=1}^d f(\mathbf{x})\eta_i [\varphi]$$

be a simple (possibly non-linear) loop. We call $b \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ a metering function for α if the following conditions hold:

$$\neg \text{guard}(\alpha) \implies b \leq 0 \tag{4.11}$$

$$\text{guard}(\alpha) \implies b\eta_i \geq b - 1 \text{ for all } i \in \{1, \dots, d\} \tag{4.12}$$

Note that Definition 4.23 is a generalization of Definition 4.13, i.e., if α is linear, then Definition 4.23 and Definition 4.13 coincide.

4.4. NON-LINEAR ITSS

Example 4.24 (Metering Function for Fibonacci). According to Definition 4.23, $\frac{1}{2} \cdot x - 1$ is a metering function for the recursive Fibonacci rule from Example 4.5. It satisfies (4.11), as we have $x \leq 1 \implies \frac{1}{2} \cdot x - 1 \leq 0$. The recursive call $\text{fib}(x - 1)$ satisfies (4.12), since we have

$$x > 1 \implies \frac{1}{2} \cdot (x - 1) - 1 = \frac{1}{2} \cdot x - \frac{3}{2} \geq \frac{1}{2} \cdot x - 2.$$

Finally, the recursive call $\text{fib}(x - 2)$ also satisfies (4.12), since we have

$$x > 1 \implies \frac{1}{2} \cdot (x - 2) - 1 = \frac{1}{2} \cdot x - 2 \geq \frac{1}{2} \cdot x - 2.$$

While a metering function for a linear loop α immediately gives rise to a lower bound on the number of consecutive applications of α , this is not the case for non-linear loops. The following theorem clarifies the relation between metering functions for non-linear loops and the length of rewrite sequences.

Theorem 4.25 (From Metering Functions to Rewrite Sequences). *Let b be a metering function for a well-formed simple non-linear loop α with degree d . Then for all integer substitutions σ with $\sigma \models \text{guard}(\alpha)$ there is a rewrite sequence $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha}^n t$ with $n = \lceil \frac{d^{b\sigma}-1}{d-1} \rceil$.*

Proof. First note that we have $d > 1$ since α is non linear and hence n is well defined. As in the proof of Theorem 4.14, let m_{σ} be the length of the longest rewrite sequence which starts with $\text{lhs}(\alpha)\sigma$ and σ -preserves \mathcal{TV} . The case $m_{\sigma} = \omega$ is trivial. For the case $m_{\sigma} \neq \omega$, we use induction on m_{σ} .

The case $m_{\sigma} = 0$ is trivial. For the case $m_{\sigma} > 0$, note that $\sigma \models \text{guard}(\alpha)$ implies $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{rhs}(\alpha)\sigma = \sum_{i=1}^d \text{lhs}(\alpha)\eta_i\sigma$. (Recall that we have $\text{rhs}(\alpha) = \sum_{i=1}^d \text{lhs}(\alpha)\eta_i$ due to the assumption (4.3) in Section 4.1.) Let $\eta = \eta_i$ for some $i \in \{1, \dots, d\}$. Clearly, we have $m_{\eta \diamond \sigma} < m_{\sigma}$.

Case 1. $\eta \diamond \sigma \models \text{guard}(\alpha)$

In this case, the induction hypothesis implies that there is a rewrite sequence $\text{lhs}(\alpha)\eta\sigma \rightarrow_{\alpha}^{n'_{\eta}} t'$ with $n'_{\eta} = \lceil \frac{d^{b\eta\sigma}-1}{d-1} \rceil$. Since (4.12) implies $b\eta\sigma \geq b\sigma - 1$, we get $n'_{\eta} \geq \lceil \frac{d^{b\sigma-1}-1}{d-1} \rceil$. Thus we get

$$\text{rhs}(\alpha)\sigma = \sum_{i=1}^d \text{lhs}(\alpha)\eta_i\sigma \rightarrow_{\alpha}^{n''} t''$$

with

$$\begin{aligned} n'' &= \sum_{i=1}^d n'_{\eta_i} \geq d \cdot \lceil \frac{d^{b\sigma-1}-1}{d-1} \rceil \geq \lceil \frac{d^{b\sigma}-d}{d-1} \rceil \\ &= \lceil \frac{d^{b\sigma}-1-(d-1)}{d-1} \rceil = \lceil \frac{d^{b\sigma}-1}{d-1} \rceil - 1 = n - 1. \end{aligned}$$

CHAPTER 4. LOWER BOUNDS FOR ITSS

Thus we have $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha}^{1+n''} t''$ with $1 + n'' \geq n$ and hence $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha}^n t$ for some term t .

Case 2. $\eta \diamond \sigma \not\models \text{guard}(\alpha)$

Then (4.11) implies $b\eta\sigma \leq 0$ and hence we get $b\sigma \leq 1$ due to (4.12). Hence we have $n = \lceil \frac{d^b\sigma - 1}{d-1} \rceil \leq \lceil \frac{d-1}{d-1} \rceil = 1$. Thus, the rewrite sequence $\text{lhs}(\alpha)\sigma \rightarrow_{\alpha} \text{rhs}(\alpha)\sigma$ has at least length n . \square

Example 4.26 (Under-Estimating Fibonacci). Since $\frac{1}{2} \cdot x - 1$ is a metering function for the recursive Fibonacci rule from Example 4.5, each term $\text{fib}(x)\sigma$ where $\sigma \models x > 1$ admits a rewrite sequence of length $\lceil 2^{\frac{1}{2} \cdot x\sigma - 1} - 1 \rceil$.

Using Definition 4.23 and Theorem 4.25, we can finally accelerate non-linear loops.

Theorem 4.27 (Accelerating Non-Linear Loops). *Let \mathcal{P} be a well-formed ITS, let $\alpha \in \mathcal{P}$ be a simple non-linear loop such that*

$$\text{guard}(\alpha) \implies \text{cost}(\alpha) \geq 1, \quad (4.13)$$

and let b be a metering function for α . Moreover, let sink be a fresh function symbol, let

$$\alpha' = \text{lhs}(\alpha) \xrightarrow{c} \text{sink}(0, \dots, 0) [\text{guard}(\alpha)] \text{ where } c = \frac{d^b - 1}{d - 1},$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha'\}$. Then \mathcal{P}' is well-formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is equivalent.

Proof. Soundness for upper bounds is trivial and \mathcal{P}' is trivially well formed. To prove soundness for lower bounds, note that by Theorem 4.25 $\sigma \models \text{guard}(\alpha)$ implies $\text{lhs}(\alpha)\sigma \xrightarrow{\ell}_{\alpha}^{n\sigma} t$ with $n = \lceil \frac{d^b - 1}{d-1} \rceil$ for some costs ℓ and some term t . Since $\text{guard}(\alpha) \implies \text{cost}(\alpha) \geq 1$, we get $\ell \geq n\sigma$ and thus $\ell \geq c\sigma$, as we clearly have $n\sigma \geq c\sigma$. Thus, for every rewrite step with α' , there is a rewrite sequence with α which has at least the same costs. Since $\text{sink}(0, \dots, 0)$ is a normal form, this implies soundness for lower bounds. \square

Example 4.28 (Accelerating Fibonacci). Since the recursive Fibonacci rule from Example 4.5 has cost 1, (4.13) is trivially satisfied. Thus, accelerating the recursive Fibonacci rule yields

$$\text{fib}(x) \xrightarrow{2^{\frac{1}{2} \cdot x - 1} - 1} \text{sink}(0) [x > 1].$$

4.4. NON-LINEAR ITSs

Theorem 4.27 is useful to handle programs with non-linear recursion like Fibonacci. Apart from non-linear recursion, non-linear ITSs can also be used to modularize programs by extracting auxiliary functions.

Example 4.29 (Modular ITSs). In the following ITS, the auxiliary function `fac` computes the factorial of its argument, i.e., `fac(x)` computes $x!$. Using `fac`, `facSum(x)` computes $0! + \dots + x!$.

$$\begin{array}{lll}
f_0(x) & \xrightarrow{1} & \text{facSum}(x) \\
\text{facSum}(x) & \xrightarrow{1} & \text{facSum}(x-1) + \text{fac}(x) \quad [x > 0] \\
\text{facSum}(x) & \xrightarrow{1} & 1 \quad [x = 0] \\
\text{fac}(x) & \xrightarrow{1} & x \cdot \text{fac}(x-1) \quad [x > 1] \\
\text{fac}(x) & \xrightarrow{1} & 1 \quad [0 \leq x \leq 1]
\end{array}$$

To establish our assumptions from Section 4.1, we can transform it to the following ITS $\mathcal{P}_{\text{facSum}}$ without affecting its runtime complexity:

$$\begin{array}{lll}
f_0(x) & \xrightarrow{1} & \text{facSum}(x) \\
\text{facSum}(x) & \xrightarrow{1} & \text{facSum}(x-1) + \text{fac}(x) \quad [x > 0] \\
\text{facSum}(x) & \xrightarrow{1} & \text{sink}(0) \quad [x = 0] \\
\text{fac}(x) & \xrightarrow{1} & \text{fac}(x-1) \quad [x > 1] \\
\text{fac}(x) & \xrightarrow{1} & \text{sink}(0) \quad [0 \leq x \leq 1]
\end{array}$$

To analyze $\mathcal{P}_{\text{facSum}}$, we first accelerate and chain the recursive `fac` rule as in Theorem 4.17 and Theorem 4.22.

Example 4.30 (Accelerating `fac`). Clearly, $x-1$ is a metering function for the recursive `fac` rule from $\mathcal{P}_{\text{facSum}}$. Accelerating it yields

$$\text{fac}(x) \xrightarrow{tv} \text{fac}(x-tv) \quad [x > 1 \wedge 0 < tv < x].$$

Chaining this rule with the non-recursive `fac` rule yields

$$\text{fac}(x) \xrightarrow{tv+1} \text{sink}(0) \quad [x > 1 \wedge 0 < tv < x \wedge 0 \leq x-tv \leq 1]$$

which simplifies to

$$\text{fac}(x) \xrightarrow{x} \text{sink}(0) \quad [x > 1].$$

By deleting the original `fac` rules, we obtain the following ITS:

$$\begin{array}{lll}
f_0(x) & \xrightarrow{1} & \text{facSum}(x) \\
\text{facSum}(x) & \xrightarrow{1} & \text{facSum}(x-1) + \text{fac}(x) \quad [x > 0] \\
\text{facSum}(x) & \xrightarrow{1} & \text{sink}(0) \quad [x = 0] \\
\text{fac}(x) & \xrightarrow{x} & \text{sink}(0) \quad [x > 1]
\end{array}$$

At this point, we would like to chain the recursive `facSum` rule with the `fac` rule in order to inline the call to `fac`. However, Theorem 4.22 is only applicable

CHAPTER 4. LOWER BOUNDS FOR ITSS

to linear rules, but the recursive **facSum** rule is non-linear. Hence, we now generalize Theorem 4.22 to non-linear rules.

Theorem 4.31 (Chaining). *Let \mathcal{P} be a well formed ITS and let $\alpha_1, \alpha_2 \in \mathcal{P}$ where*

$$\begin{array}{llll} \alpha_1 & = & f_1(\mathbf{x}) & \xrightarrow{c_1} C[f_2(\mathbf{x})\eta_1] \quad [\varphi_1] \text{ and} \\ \alpha_2 & = & f_2(\mathbf{x}) & \xrightarrow{c_2} r \quad [\varphi_2]. \end{array}$$

W.l.o.g., let $\mathcal{TV}(\alpha_1) \cap \mathcal{TV}(\alpha_2) = \emptyset$ (otherwise, the temporary variables in α_2 can be renamed accordingly). Moreover, let

$$\alpha_{1.2} = f_1(\mathbf{x}) \xrightarrow{c_1 + c_2\eta_1} C[r\eta_1] \quad [\varphi_1 \wedge \varphi_2\eta_1]$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha_{1.2}\}$. Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is equivalent.

Proof. Soundness for upper bounds is trivial. It remains to show that \mathcal{P}' is well formed and that the processor is sound for lower bounds. To this end, we show that every evaluation step with $\alpha_{1.2}$ can be simulated by two evaluation steps with the rules α_1, α_2 of the same cost. Thus, let σ be an integer substitution with $\sigma \models \text{guard}(\alpha_{1.2})$, i.e., we have

$$f_1(\mathbf{x})\sigma \xrightarrow{c_1\sigma + c_2\eta_1\sigma}_{\alpha_{1.2}} C[r\eta_1]\sigma.$$

Since $\sigma \models \varphi_1$, we have

$$f_1(\mathbf{x})\sigma \xrightarrow{c_1\sigma}_{\alpha_1} C[f_2(\mathbf{x})\eta_1]\sigma.$$

Since $\sigma \models \varphi_2\eta_1$ implies $\eta_1 \diamond \sigma \models \varphi_2$ we have

$$f_2(\mathbf{x})\eta_1\sigma \xrightarrow{c_2\eta_1\sigma}_{\alpha_2} r\eta_1\sigma$$

and hence

$$C[f_2(\mathbf{x})\eta_1]\sigma \xrightarrow{c_2\eta_1\sigma}_{\alpha_2} C[r\eta_1]\sigma.$$

Thus, we have $f_1(\mathbf{x})\sigma \xrightarrow{c_1\sigma + c_2\eta_1\sigma}_{\mathcal{P}} C[r\eta_1]\sigma$ as desired. \square

Theorem 4.31 allows us to continue Example 4.30.

Example 4.32 (Chaining **facSum** and **fac**). Chaining the recursive **facSum** rule from Example 4.30 with the **fac** rule yields

$$\text{facSum}(x) \xrightarrow{x+1} \text{facSum}(x-1) + \text{sink}(0) \quad [x > 1].$$

At this point, we would like to accelerate the recursive **facSum** rule. However, it is neither a simple loop (since its right-hand side is not linear) nor a simple non-linear loop (since its right-hand side contains two different function symbols).

4.4. NON-LINEAR ITSs

The following theorem allows us to transform such rules to simple (non-linear) rules by deleting subterms of the right-hand side.

Theorem 4.33 (Partial Deletion). *Let \mathcal{P} be a well-formed ITS and let $\alpha \in \mathcal{P}$ be a rule with $\text{rhs}(\alpha) = \sum_{i=1}^m f_i(\mathbf{x})\eta_i$ where $f_1, \dots, f_m \in \Sigma$. Moreover, let α' be like α , but $\text{rhs}(\alpha') = \sum_{i \in \{1, \dots, m\} \setminus \{j\}} f_i(\mathbf{x})\eta_i$ for some $j \in \{1, \dots, m\}$, and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha'\}$. Then \mathcal{P}' is well formed and the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for lower bounds.*

Proof. Since \mathcal{P} is well formed, \mathcal{P}' is trivially well formed, too. In the following, let \mathcal{X}, \mathcal{Y} denote multisets of int-basic terms. Moreover, for convenience we identify the multiset \mathcal{X} with the term $\sum_{t \in \mathcal{X}} t$. To prove soundness for lower bounds, we define $\mathcal{X} \xrightarrow{\ell}_{\mathcal{P} \circ \supseteq} \mathcal{Y}$ if $\mathcal{X} \xrightarrow{\ell}_{\mathcal{P}} \circ \supseteq \mathcal{Y}$. Then we clearly have $\text{dh} \rightarrow_{\mathcal{P}'} \leq \text{dh} \rightarrow_{\mathcal{P} \circ \supseteq}$ as each $\rightarrow_{\mathcal{P}'}$ -sequence is also a valid $\rightarrow_{\mathcal{P} \circ \supseteq}$ -sequence. To finish the proof, we show $\text{dh} \rightarrow_{\mathcal{P} \circ \supseteq} \leq \text{dh} \rightarrow_{\mathcal{P}}$. This clearly implies $\text{rc}_{cp(\mathcal{P})} \leq \text{rc}_{cp(\mathcal{P}')}$, i.e., it implies that the processor is sound for lower bounds.

Consider an evaluation $\mathcal{X}_0 \xrightarrow{\ell_1}_{\mathcal{P} \circ \supseteq} \dots \xrightarrow{\ell_n}_{\mathcal{P} \circ \supseteq} \mathcal{X}_n$. We prove

$$\mathcal{X}_0 \xrightarrow{\ell_1}_{\mathcal{P}} \dots \xrightarrow{\ell_n}_{\mathcal{P}} \mathcal{X}'_n \supseteq \mathcal{X}_n \text{ for some } \mathcal{X}'_n$$

by induction on n . The case $n = 0$ is trivial. If $n > 0$, the induction hypothesis implies

$$\mathcal{X}_0 \xrightarrow{\ell_1}_{\mathcal{P}} \dots \xrightarrow{\ell_{n-1}}_{\mathcal{P}} \mathcal{X}'_{n-1} \supseteq \mathcal{X}_{n-1} \text{ for some } \mathcal{X}'_{n-1}.$$

Moreover,

$$\mathcal{X}_{n-1} \xrightarrow{\ell_n}_{\mathcal{P} \circ \supseteq} \mathcal{X}_n \text{ implies } \mathcal{X}_{n-1} \xrightarrow{\ell_n}_{\mathcal{P}} \overline{\mathcal{X}}_n \supseteq \mathcal{X}_n,$$

i.e., we have $t \xrightarrow{\ell_n}_{\mathcal{P}} \mathcal{Y}$ and $\overline{\mathcal{X}}_n = (\mathcal{X}_{n-1} \setminus \{t\}) \cup \mathcal{Y}$ for some $t \in \mathcal{X}_{n-1}$. Since $\mathcal{X}_{n-1} \subseteq \mathcal{X}'_{n-1}$, we get

$$\mathcal{X}'_{n-1} \xrightarrow{\ell_n}_{\mathcal{P}} (\mathcal{X}'_{n-1} \setminus \{t\}) \cup \mathcal{Y} = \mathcal{X}'_n$$

and thus $\mathcal{X}_0 \xrightarrow{\ell_1}_{\mathcal{P}} \dots \xrightarrow{\ell_n}_{\mathcal{P}} \mathcal{X}'_n \supseteq \mathcal{X}_n$ with $\mathcal{X}'_n \supseteq \overline{\mathcal{X}}_n \supseteq \mathcal{X}_n$, as desired. \square

Example 4.34 (Applying Partial Deletion to `facSum`). Applying Theorem 4.33 to the call to `sink` in the recursive `facSum` rule from Example 4.32 yields

$$\text{facSum}(x) \xrightarrow{x+1} \text{facSum}(x-1) [x > 1].$$

Accelerating this rule via Theorem 4.17 with the metering function $x - 1$ yields

$$\text{facSum}(x) \xrightarrow{x \cdot tv - \frac{1}{2} \cdot tv^2 + \frac{3}{2} \cdot tv} \text{facSum}(x - tv) [x > 1 \wedge 0 < tv < x] \quad (4.14)$$

CHAPTER 4. LOWER BOUNDS FOR ITSs

since $x\eta^{tv} = x - tv$ and

$$\sum_{i=0}^{tv-1} x\eta^i + 1 = \sum_{i=0}^{tv-1} x - i + 1 = x \cdot tv - \frac{1}{2} \cdot tv^2 + \frac{3}{2} \cdot tv.$$

By chaining (4.14) with the f_0 rule, we obtain

$$f_0(x) \xrightarrow{1+x \cdot tv - \frac{1}{2} \cdot tv^2 + \frac{3}{2} \cdot tv} \text{facSum}(x - tv) \ [x > 1 \wedge 0 < tv < x].$$

Finally, deleting all other rules results in a *simplified ITS*. Section 4.5 shows how to analyze the complexity of such simplified ITSs.

Algorithm 2 shows how Algorithm 1 can be adapted in order to handle non-linear ITSs. The first additional step is Step 2, which deletes sinks from right-hand sides as in Example 4.34. The second change is that we apply Partial Deletion if we failed to accelerate a non-linear loop in Step 3.2. In this way, the degree of the loop is reduced, which may simplify its acceleration. In this step, it is not clear which subterm of the right-hand side should be deleted. Here one may for example try all possibilities until the loop can be accelerated successfully.

4.4. NON-LINEAR ITSS

Algorithm 2 Program Simplification for Non Linear ITSS

While there is a rule α with $\text{root}(\alpha) \neq f_0$:

1. Apply *Deletion* to rules whose guard is proved unsatisfiable or whose root symbol is unreachable from f_0 .
 2. While there is a rule α whose right-hand side contains a symbol f without outgoing rules:
 - 2.1. Apply *Partial Deletion* to an occurrence of f in $\text{rhs}(\alpha)$.
 - 2.2. Apply *Deletion* to α .
 3. While there is a non-accelerated simple loop α :
 - 3.1. Try to apply *Loop Acceleration* to α .
 - 3.2. If 3.1 failed:
 - If $\mathcal{TV}(\alpha) \neq \emptyset$, then apply *Strengthening* to α to eliminate a temporary variable.
 - Otherwise, if α is non-linear, apply *Partial Deletion* to α .
 - 3.3. Apply *Deletion* to α .
 4. Let $S = \emptyset$.
 5. While there is an accelerated rule α :
 - 5.1. For each α' where $\text{rhs}(\alpha')$ contains $\text{root}(\alpha)$ but not $\text{root}(\alpha')$:
 - Apply *Chaining* to α' and α .
 - Add α' to S .
 - 5.2. Apply *Deletion* to α .
 6. Apply *Deletion* to each rule in S .
 7. While there is a function symbol f without simple loops but with incoming and outgoing rules (starting with symbols f with just one incoming rule):
 - 7.1. Apply *Chaining* to each pair α', α where $\text{root}(\alpha) = f$ and f occurs on the right-hand side of α' .
 - 7.2. Apply *Deletion* to each α where $\text{root}(\alpha) = f$ or $\text{target}(\alpha) = f$.
-

4.5 Asymptotic Lower Bounds

After Algorithm 2, all program paths have length 1. We call such programs *simplified* and throughout this section we assume that \mathcal{P} is a simplified program. Now for any integer substitution σ ,

$$\max\{\text{cost}(\alpha)\sigma \mid \alpha \in \mathcal{P}, \sigma \models \text{guard}(\alpha)\}, \quad (4.15)$$

is a lower bound on $\text{dh} \rightarrow_{\mathcal{P}} (f_0(\mathbf{x})\sigma)$, i.e., (4.15) is the maximal cost of those rules whose guard is satisfied by σ . So for the program in Figure 4.3c, we obtain the bound

$$\frac{x^2 \cdot tv + x \cdot tv - tv^2 + 3 \cdot tv + 2 \cdot x + 4}{2} \quad (4.16)$$

for all integer substitutions with $\sigma \models 0 < tv < \frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x$. However, such bounds do not provide an intuitive understanding of the program's complexity and are also not suitable to detect possible attacks. The reason is that both $\text{cost}(\alpha)$ and $\text{guard}(\alpha)$ may be complex and, even more importantly, they may be interdependent. For example, the bound (4.16) is cubic, but it even witnesses that the complexity of the program from Figure 4.3c is at least a polynomial of degree 4. The reason is that the value of the temporary variable tv may be quadratic in the value of the program variable x according to the condition $tv < \frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x$.

Hence, we now show how to derive *asymptotic* lower bounds for simplified programs. These asymptotic bounds can easily be understood (i.e., a high lower bound can help programmers to improve their program to make it more efficient) and they identify potential attacks.

To derive asymptotic bounds, we use so-called *limit problems*, cf. Section 4.5.1. A limit problem is an abstraction of the guard φ of a rule which allows us to analyze how to satisfy φ , presuming that all variables are instantiated with “large enough” values. More precisely, a solution of a limit problem is a *family* of substitutions σ_m which is parameterized by a single variable m . This family of substitutions satisfies φ for large enough m and can be found using the calculus presented in Section 4.5.2. Thus, applying σ_m to the cost of a rule yields a univariate bound, even if the rule has a multivariate cost function and hence allows us to deduce an asymptotic bound.

4.5.1 Limit Problems

While $\text{dh} \rightarrow_{\mathcal{P}}$ is defined on terms, asymptotic bounds are usually defined for functions on \mathbb{N} . Thus, our goal is to derive an asymptotic lower bound for $\text{rc}_{cp(\mathcal{P})}$ from a concrete bound on $\text{dh} \rightarrow_{\mathcal{P}}$ of the form (4.15). So for the program \mathcal{P} in Figure 4.3c, we would like to derive $\text{rc}_{cp(\mathcal{P})}(n) \in \Omega(n^4)$. However, as discussed above, in general the costs of a rule do not directly give rise to the desired asymptotic lower bound.

To infer an asymptotic lower bound from a rule $\alpha \in \mathcal{P}$, we try to find an

4.5. ASYMPTOTIC LOWER BOUNDS

infinite family of integer substitutions σ_m (parameterized by $m \in \mathbb{N}$) such that there is an $m_0 \in \mathbb{N}$ with $\sigma_m \models \text{guard}(\alpha)$ for all $m \geq m_0$. This implies $\text{rc}_{cp(\mathcal{P})}(\|f_0(\mathbf{x})\sigma_m\|_i) \in \Omega(\text{cost}(\alpha)\sigma_m)$, since for all $m \geq m_0$ we have

$$\text{rc}_{cp(\mathcal{P})}(\|f_0(\mathbf{x})\sigma_m\|_i) \geq \text{dh}_{\rightarrow \mathcal{P}}(f_0(\mathbf{x})\sigma_m) \geq \text{cost}(\alpha)\sigma_m.$$

To find such a family of substitutions, we first normalize all constraints in $\text{guard}(\alpha)$ such that they have the form $a > 0$ or $a \geq 0$. Now our goal is to find infinitely many models σ_m for a formula of the form “ $\bigwedge_{i=1}^k a_i \circ 0$ ” where $\circ \in \{>, \geq\}$. Obviously, such a formula is satisfied for large enough m if all terms $a_i\sigma_m$ are positive constants or increase infinitely towards ω . Thus, we introduce a technique which tries to find out whether fixing the valuations of some variables and increasing or decreasing the valuations of others results in positive resp. increasing valuations of a_1, \dots, a_k . Our technique operates on *limit problems* of the form $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$ where a_i is an arithmetic expression and $\bullet_i \in \{+, -, +!, -!\}$ for all $i \in \{1, \dots, k\}$. Here, a^+ (resp. a^-) means that a has to grow towards ω (resp. $-\omega$) and $a^{+!}$ (resp. $a^{-!}$) means that a has to be a positive (resp. negative) constant. So we represent $\text{guard}(\alpha)$ by an *initial limit problem* $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$ where $\bullet_i \in \{+, +!\}$ for all $i \in \{1, \dots, k\}$. To solve a limit problem S , we search for a *solution* σ_m of S , which is defined in terms of *limits* of functions.

Definition 4.35 (Limit). For each $f : \mathbb{N} \rightarrow \mathbb{R}$ we have $\lim_{n \rightarrow \omega} f(n) = \omega$ (resp. $\lim_{n \rightarrow \omega} f(n) = -\omega$) if for every $m \in \mathbb{R}$ there is an $n_0 \in \mathbb{N}$ such that $f(n) \geq m$ (resp. $f(n) \leq m$) holds for all $n \geq n_0$. Similarly, we have $\lim_{n \rightarrow \omega} f(n) = m$ if there is an n_0 such that $f(n) = m$ holds for all $n \geq n_0$.

Now a family of substitutions σ_m is a solution for a limit problem $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$ if $\lim_{m \rightarrow \omega} a_i\sigma_m$ complies with \bullet_i for each $i \in \{1, \dots, k\}$.

Definition 4.36 (Solutions of Limit Problems). For any function $f : \mathbb{N} \rightarrow \mathbb{R}$ and any $\bullet \in \{+, -, +!, -!\}$, we say that f *satisfies* \bullet if:

$$\begin{aligned} \lim_{m \rightarrow \omega} f(m) &= \omega, \text{ if } \bullet = + & \exists c \in \mathbb{R}. \lim_{m \rightarrow \omega} f(m) = c > 0, \text{ if } \bullet = +! \\ \lim_{m \rightarrow \omega} f(m) &= -\omega, \text{ if } \bullet = - & \exists c \in \mathbb{R}. \lim_{m \rightarrow \omega} f(m) = c < 0, \text{ if } \bullet = -! \end{aligned}$$

A family σ_m of integer substitutions is a *solution* of a limit problem S if for every $a^{\bullet} \in S$, the function $\lambda m. a\sigma_m$ satisfies \bullet .

Example 4.37 (Bound for Figure 4.3c). Consider the initial limit problem $S = \{tv^+, (\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv)^{+!}\}$ for Figure 4.3c. It is solved by σ_m with $(tv)\sigma_m = \frac{1}{2} \cdot m^2 + \frac{1}{2} \cdot m - 1$, $x\sigma_m = m$ and $y\sigma_m = z\sigma_m = u\sigma_m = 0$. The reason is that $\lim_{m \rightarrow \omega} \lambda m. (tv)\sigma_m = \omega$, i.e., $\lim_{m \rightarrow \omega} (tv)\sigma_m$ satisfies $+$. Similarly, $\lambda m. (\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv)\sigma_m = \lambda m. 1$ satisfies $+!$.

CHAPTER 4. LOWER BOUNDS FOR ITSS

Section 4.5.2 will show how to infer such solutions of limit problems automatically. The following theorem clarifies how to deduce an asymptotic lower bound from a solution of a limit problem.

Theorem 4.38 (Asymptotic Bounds for Simplified Programs). *Given a rule α of a simplified program \mathcal{P} with $\text{guard}(\alpha) = a_1 \circ_1 0 \wedge \dots \wedge a_k \circ_k 0$ where $\circ_1, \dots, \circ_k \in \{>, \geq\}$, let the family σ_m be a solution of an initial limit problem $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$ with $\bullet_1, \dots, \bullet_k \in \{+, +!\}$. Then $\text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i) \in \Omega(\text{cost}(\alpha)\sigma_m)$.*

Proof. Since σ_m is a solution of $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$, there is an $m_0 \in \mathbb{N}$ such that for all $m \geq m_0$, we have $\sigma_m \models a_1 > 0 \wedge \dots \wedge a_k > 0$, i.e., $\sigma_m \models \text{guard}(\alpha)$. Hence, for all $m \geq m_0$, we obtain:

$$\begin{aligned} \text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i) &\geq \text{dh}_{\rightarrow \mathcal{P}}(\text{lhs}(\alpha)\sigma_m) \\ &\geq \text{cost}(\alpha)\sigma_m \quad \text{as } \sigma_m \models \text{guard}(\alpha) \end{aligned}$$

This implies $\text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i) \in \Omega(\text{cost}(\alpha)\sigma_m)$. □

Of course, if \mathcal{P} has several rules, then we try to take the one which results in the highest lower bound. Moreover, one should extend the initial limit problem $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}\}$ by $\text{cost}(\alpha)^+$. In this way, one searches for families of substitutions σ_m where $\text{cost}(\alpha)\sigma_m$ depends on m , i.e., where the costs are not constant.

Example 4.39 (Asymptotic Bound for Figure 4.3c). We continue Example 4.37. According to Theorem 4.38, we get the asymptotic lower bound

$$\begin{aligned} \text{rc}_{cp(\mathcal{P})}(\|f_0(x, y, z, u)\sigma_m\|_i) &\in \Omega(\text{cost}(\alpha)\sigma_m) \\ &= \Omega\left(\frac{1}{8} \cdot m^4 + \frac{1}{4} \cdot m^3 + \frac{7}{8} \cdot m^2 + \frac{7}{4} \cdot m\right) \\ &= \Omega(m^4). \end{aligned}$$

The costs are *unbounded* (i.e., they only depend on temporary variables) if the initial limit problem $\{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}, \text{cost}(\alpha)^+\}$ has a solution σ_m where $x\sigma_m$ is constant for all $x \in \mathcal{PV}$. Then we can even infer $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(\omega)$.

Example 4.40 (Unbounded Loops Continued). By chaining the rule α_{it} from Example 4.19 with the initial rule $f_0(x, y) \xrightarrow{1} f(x, y)$ (see Example 4.16), we obtain

$$f_0(x, y) \xrightarrow{tv_1 \cdot y + 1} f(x + tv_1, y) \quad [0 < x \wedge 0 < tv_1 < tv + 1].$$

The resulting initial limit problem $\{x^{+!}, (tv_1)^+, (tv + 1 - tv_1)^{+!}, (tv_1 \cdot y + 1)^+\}$ has the solution σ_m with $x\sigma_m = y\sigma_m = 1$ and $(tv)\sigma_m = (tv_1)\sigma_m = m$, which implies $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(\omega)$.

4.5. ASYMPTOTIC LOWER BOUNDS

Theorem 4.38 results in bounds “ $\text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i) \in \Omega(\text{cost}(\alpha)\sigma_m)$ ” which depend on the sizes $\|\text{lhs}(\alpha)\sigma_m\|_i$. Let $f(m) = \text{rc}_{cp(\mathcal{P})}(m)$, $g(m) = \|\text{lhs}(\alpha)\sigma_m\|_i$, and let $\Omega(\text{cost}(\alpha)\sigma_m)$ have the form $\Omega(m^k)$ or $\Omega(k^m)$ for some $k \in \mathbb{N}$. Moreover for all $x \in \mathcal{PV}$, let $x\sigma_m$ be a polynomial of at most degree d , i.e., let $g(m) \in \mathcal{O}(m^d)$. Then the following lemma allows us to infer a bound for $\text{rc}_{cp(\mathcal{P})}(m)$ instead of $\text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i)$. Here, we use the notation $\mathbb{R}_{\circ c} = \{x \in \mathbb{R} \mid x \circ c\}$ for $\circ \in \{\geq, >\}$ (and $\mathbb{N}_{\circ c}$ is defined analogously).

Lemma 4.41 (Bounds for Function Composition). *Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ where $g(m) \in \mathcal{O}(m^d)$ for some $d \in \mathbb{N}$ with $d > 0$. Moreover, let $f(m)$ be weakly and let $g(m)$ be strictly monotonically increasing for large enough m .*

- If $f(g(m)) \in \Omega(m^k)$ with $k \in \mathbb{N}$, then $f(m) \in \Omega(m^{\frac{k}{d}})$.
- If $f(g(m)) \in \Omega(k^m)$ with $k > 1$, then $f(m) \in \Omega(b^{\sqrt[d]{m}})$ for some $b > 1$.

Proof. For any (total) function $h : M \rightarrow \mathbb{N}_{\geq m_0}$ with $M \subseteq \mathbb{N}$ where M is infinite, we define $\lfloor h \rfloor(m) : \mathbb{N}_{\geq \min(M)} \rightarrow \mathbb{N}_{\geq m_0}$ and $\lceil h \rceil(m) : \mathbb{N} \rightarrow \mathbb{N}_{\geq m_0}$ by:

$$\begin{aligned} \lfloor h \rfloor(m) &= h(\max\{m' \in M \mid m' \leq m\}) \\ \lceil h \rceil(m) &= h(\min\{m' \in M \mid m' \geq m\}) \end{aligned}$$

Note that infinity of h 's domain of definition M ensures that there is always an $m' \in M$ with $m' \geq m$.

To prove the lemma, we first show that if $h : M \rightarrow \mathbb{N}_{\geq m_0}$ is strictly monotonically increasing and surjective, then

$$\lfloor h \rfloor(m) \in \{\lceil h \rceil(m), \lceil h \rceil(m) - 1\} \quad \text{for all } m \in \mathbb{N}_{\geq \min(M)} \quad (4.17)$$

Then we prove the case $f(g(m)) \in \Omega(m^k)$. The proof for the case $f(g(m)) \in \Omega(k^m)$ is analogous and thus it is only presented in the appendix of this thesis, cf. Appendix A.

Claim 1. $\lfloor h \rfloor(m) \in \{\lceil h \rceil(m), \lceil h \rceil(m) - 1\}$

To prove (4.17), let $m \in \mathbb{N}_{\geq \min(M)}$. If $m \in M$, then clearly $\lfloor h \rfloor(m) = \lceil h \rceil(m)$. If $m \notin M$, then let $\check{m} = \max\{x \in M \mid x < m\}$ and $\hat{m} = \min\{x \in M \mid x > m\}$. Thus, $\check{m} < m < \hat{m}$. Strict monotonicity of h implies $h(\check{m}) < h(\hat{m})$. Assume that $h(\hat{m}) - h(\check{m}) > 1$. Then by surjectivity of h , there is an $\overline{m} \in M$ with $h(\overline{m}) = h(\check{m}) + 1$ and thus $h(\check{m}) < h(\overline{m}) < h(\hat{m})$. By strict monotonicity of h , we obtain $\check{m} < \overline{m} < \hat{m}$. Since $m \notin M$ and $\overline{m} \in M$ implies $m \neq \overline{m}$, we either have $\overline{m} < m$ which contradicts $\check{m} = \max\{\check{m} \in M \mid \check{m} < m\}$ or $\overline{m} > m$ which contradicts $\hat{m} = \min\{\hat{m} \in M \mid \hat{m} > m\}$. Hence, $\lfloor h \rfloor(m) = h(\check{m}) = h(\hat{m}) - 1 = \lceil h \rceil(m) - 1$, which proves (4.17).

CHAPTER 4. LOWER BOUNDS FOR ITSS

Claim 2. $f(g(m)) \in \Omega(m^k)$ implies $f(m) \in \Omega(m^{\frac{k}{d}})$

Note that $g(m) \in \mathcal{O}(m^d)$ and $f(g(m)) \in \Omega(m^k)$ imply

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq m_0}. g(m) \leq c \cdot m^d \wedge c' \cdot m^k \leq f(g(m)).$$

We can choose m_0 large enough such that $f|_{\mathbb{N}_{\geq m_0}}$ is weakly and $g|_{\mathbb{N}_{\geq m_0}}$ is strictly monotonically increasing. Let $M = \{g(m) \mid m \geq m_0\}$ and let $g^{-1} : M \rightarrow \mathbb{N}_{\geq m_0}$ be the function such that $g(g^{-1}(m)) = m$. Note that g^{-1} exists, since strict monotonicity of g implies injectivity of g . By instantiating m with $g^{-1}(m)$, we obtain

$$\begin{aligned} \exists m_0, c, c' > 0. \forall m \in M. \\ g(g^{-1}(m)) \leq c \cdot (g^{-1}(m))^d \wedge c' \cdot (g^{-1}(m))^k \leq f(g(g^{-1}(m))) \end{aligned}$$

which simplifies to

$$\exists m_0, c, c' > 0. \forall m \in M. m \leq c \cdot (g^{-1}(m))^d \wedge c' \cdot (g^{-1}(m))^k \leq f(m).$$

When dividing by c and building the d^{th} root on both sides of the first inequality, we get

$$\exists m_0, c, c' > 0. \forall m \in M. \sqrt[d]{\frac{m}{c}} \leq g^{-1}(m) \wedge c' \cdot (g^{-1}(m))^k \leq f(m).$$

By monotonicity of $\sqrt[d]{\frac{m}{c}}$ and $f(m)$ in m , this implies

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. \sqrt[d]{\frac{m}{c}} \leq \lceil g^{-1} \rceil(m) \wedge c' \cdot (\lfloor g^{-1} \rfloor(m))^k \leq f(m).$$

Note that $g|_{\mathbb{N}_{\geq m_0}}$ is total and hence, $g^{-1} : M \rightarrow \mathbb{N}_{\geq m_0}$ is surjective. Moreover, by strict monotonicity of $g|_{\mathbb{N}_{\geq m_0}}$, M is infinite and g^{-1} is also strictly monotonically increasing. Hence, by (4.17) we get $\lceil g^{-1} \rceil(m) \leq \lfloor g^{-1} \rfloor(m) + 1$ for all $m \in \mathbb{N}_{\geq g(m_0)}$. Thus,

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. \sqrt[d]{\frac{m}{c}} - 1 \leq \lfloor g^{-1} \rfloor(m) \wedge c' \cdot (\lfloor g^{-1} \rfloor(m))^k \leq f(m)$$

which implies

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c' \cdot \left(\sqrt[d]{\frac{m}{c}} - 1 \right)^k \leq f(m).$$

Therefore, $\exists c > 0. f(m) \in \Omega\left(\left(\sqrt[d]{\frac{m}{c}} - 1\right)^k\right)$ and thus, $f(m) \in \Omega\left(m^{\frac{k}{d}}\right)$. \square

4.5. ASYMPTOTIC LOWER BOUNDS

Example 4.42 (Bound for Figure 4.3c Continued). In Example 4.39, we inferred $\text{rc}_{cp(\mathcal{P})}(\|f_0(x, y, z, u)\sigma_m\|_i) \in \Omega(m^4)$ where $x\sigma_m = m$ and $y\sigma_m = z\sigma_m = u\sigma_m = 0$. Hence, we have $\|f_0(x, y, z, u)\sigma_m\|_i = m \in \mathcal{O}(m^1)$. By Lemma 4.41, we obtain $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(m^{\frac{4}{1}}) = \Omega(m^4)$.

In some cases, Lemma 4.41 even allows us to infer sub-linear bounds.

Example 4.43 (Sub-Linear Bounds). Let

$$\mathcal{P} = \{f_0(x, y) \xrightarrow{y} f(x, y) \ [x > y^2]\}.$$

By Definition 4.36, the family σ_m with $x\sigma_m = m^2 + 1$ and $y\sigma_m = m$ is a solution of the initial limit problem $\{(x - y^2)^+, y^+\}$. Due to Theorem 4.38, this proves $\text{rc}_{cp(\mathcal{P})}(\|f_0(x, y)\sigma_m\|_i) \in \Omega(m)$. As $\|f_0(x, y)\sigma_m\|_i = m^2 + 1 + m \in \mathcal{O}(m^2)$, Lemma 4.41 results in $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(m^{\frac{1}{2}}) = \Omega(\sqrt{m})$.

4.5.2 Transforming Limit Problems

A limit problem S is *trivial* if all terms in S are variables and there is no variable x with $x^{\bullet_1}, x^{\bullet_2} \in S$ and $\bullet_1 \neq \bullet_2$. For trivial limit problems S we can immediately obtain a particular solution σ_m^S which instantiates variables “according to S ”.

Lemma 4.44 (Solving Trivial Limit Problems). *Let S be a trivial limit problem. Then σ_m^S is a solution of S where for all $m \in \mathbb{N}$, σ_m^S is defined as follows:*

$$x\sigma_m^S = \begin{cases} m & \text{if } x^+ \in S \\ -m & \text{if } x^- \in S \\ 1 & \text{if } x^{+!} \in S \\ -1 & \text{if } x^{-!} \in S \\ 0 & \text{otherwise} \end{cases}$$

Proof. If $x^+ \in S$ (resp. $x^- \in S$), then $x\sigma_m^S = m$ (resp. $x\sigma_m^S = -m$) and thus, $\lim_{m \mapsto \omega} x\sigma_m = \lim_{m \mapsto \omega} m = \omega$ (resp. $\lim_{m \mapsto \omega} x\sigma_m = \lim_{m \mapsto \omega} -m = -\omega$), i.e., $\lambda m. x\sigma_m$ satisfies $+$ (resp. $-$). If $x^{+!} \in S$ (resp. $x^{-!} \in S$), then $x\sigma_m^S = 1$ (resp. $x\sigma_m^S = -1$). Thus, $\lim_{m \mapsto \omega} x\sigma_m = 1$ (resp. $\lim_{m \mapsto \omega} x\sigma_m = -1$), i.e., $\lambda m. x\sigma_m$ satisfies $+!$ (resp. $-!$). Hence, σ_m^S is a solution of S . \square

For instance, if $\mathcal{V}(\alpha) = \{x, y, tv\}$ and $S = \{x^+, y^{-!}\}$, then S is a trivial limit problem and σ_m^S with $x\sigma_m^S = m$, $x\sigma_m^S = -1$, and $(tv)\sigma_m^S = 0$ is a solution for S . However, in general the initial limit problem $S = \{a_1^{\bullet_1}, \dots, a_k^{\bullet_k}, \text{cost}(\alpha)^+\}$ is not trivial. Therefore, we now define a transformation \rightsquigarrow to simplify limit problems

CHAPTER 4. LOWER BOUNDS FOR ITSS

until one reaches a trivial problem. With our transformation, $S \rightsquigarrow S'$ ensures that each solution of S' also gives rise to a solution of S .

If S contains $f(a_1, a_2)^\bullet$ for some standard arithmetic operation f like addition, subtraction, multiplication, division, and exponentiation, we use a so-called *limit vector* (\bullet_1, \bullet_2) with $\bullet_i \in \{+, -, +!, -!\}$ to characterize for which kinds of arguments the operation f is increasing (if $\bullet = +$) resp. decreasing (if $\bullet = -$) resp. a positive or negative constant (if $\bullet = +!$ or $\bullet = -!$).² Then S can be transformed into the new limit problem $S \setminus \{f(a_1, a_2)^\bullet\} \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$.

For example, $(+, +!)$ is an increasing limit vector for subtraction. The reason is that $a_1 - a_2$ is increasing if a_1 is increasing and a_2 is a positive constant. Hence, our transformation \rightsquigarrow allows us to replace $(a_1 - a_2)^+$ by a_1^+ and $a_2^{+!}$.

To define limit vectors formally, we say that (\bullet_1, \bullet_2) is an *increasing* (resp. *decreasing*) *limit vector* for f if the function $\lambda m. f(g(m), h(m))$ satisfies $+$ (resp. $-$) for any functions g and h that satisfy \bullet_1 and \bullet_2 , respectively. Similarly, (\bullet_1, \bullet_2) is a *positive* (resp. *negative*) *limit vector* for f if $\lambda m. f(g(m), h(m))$ satisfies $+!$ (resp. $-!$) for any functions g and h that satisfy \bullet_1 and \bullet_2 , respectively.

With this definition, $(+, +!)$ is indeed an increasing limit vector for subtraction, since $\lim_{m \rightarrow \omega} g(m) = \omega$ and $\lim_{m \rightarrow \omega} h(m) = k$ with $k > 0$ implies $\lim_{m \rightarrow \omega} (g(m) - h(m)) = \omega$. In other words, if $g(m)$ satisfies $+$ and $h(m)$ satisfies $+!$, then $g(m) - h(m)$ satisfies $+$ as well. In contrast, $(+, +)$ is not an increasing limit vector for subtraction. To see this, consider the functions $g(m) = h(m) = m$. Both $g(m)$ and $h(m)$ satisfy $+$, whereas $g(m) - h(m) = 0$ does not satisfy $+$. Similarly, $(+!, +!)$ is not a positive limit vector for subtraction, since for $g(m) = 1$ and $h(m) = 2$, both $g(m)$ and $h(m)$ satisfy $+!$, but $g(m) - h(m) = -1$ does not satisfy $+!$.

Limit vectors can be used to simplify limit problems, cf. (A) in the following definition. Moreover, for numbers $k \in \mathbb{Z}$, one can easily simplify constraints of the form $k^{+!}$ and $k^{-!}$ (e.g., $2^{+!}$ is obviously satisfied since $2 > 0$), cf. (B).

Definition 4.45 (\rightsquigarrow). Let S be a limit problem. We have:

- (A) $S \cup \{f(a_1, a_2)^\bullet\} \rightsquigarrow S \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$ if \bullet is $+$ (resp. $-$, $+!$, $-!$) and (\bullet_1, \bullet_2) is an increasing (resp. decreasing, positive, negative) limit vector for f
- (B) $S \cup \{k^{+!}\} \rightsquigarrow S$ if $k \in \mathbb{Z}$ and $k > 0$, $S \cup \{k^{-!}\} \rightsquigarrow S$ if $k \in \mathbb{Z}$ and $k < 0$

However, transforming a limit problem with \rightsquigarrow may also result in *contradictory* limit problems that contain x^{\bullet_1} and x^{\bullet_2} where $\bullet_1 \neq \bullet_2$, as the following example illustrates.

²To ease the presentation, we restrict ourselves to binary operations f . For operations of arity n , one would need limit vectors of the form $(\bullet_1, \dots, \bullet_n)$.

4.5. ASYMPTOTIC LOWER BOUNDS

Example 4.46 (Bound for Figure 4.3c Continued). For the initial limit problem from Example 4.37, we have

$$\begin{aligned} \{tv^+, (\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x - tv)^{+!}\} &\rightsquigarrow \{tv^+, (\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x)^{+!}, tv^{-!}\} \\ &\rightsquigarrow \{tv^+, (\tfrac{1}{2} \cdot x^2)^{+!}, (\tfrac{1}{2} \cdot x)^{+!}, tv^{-!}\} \\ &\rightsquigarrow^* \{tv^+, x^{+!}, tv^{-!}\} \end{aligned}$$

using the positive limit vector $(+!, -!)$ for subtraction and the positive limit vector $(+!, +!)$ for addition.

The resulting problem in Example 4.46 is not trivial as it contains tv^+ and $tv^{-!}$, i.e., we failed to compute an asymptotic lower bound. However, if we substitute tv with its upper bound $\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1$, then we can reduce the initial limit problem to a trivial one. Hence, we now extend \rightsquigarrow by allowing to apply substitutions.

Definition 4.47 (\rightsquigarrow Continued). Let S be a limit problem and let θ be a substitution such that $x \notin \mathcal{V}(x\theta)$ for all $x \in \text{dom}(\theta)$ and $\theta \diamond \sigma$ is an integer substitution for each integer substitution σ . Then we have:³

$$(C) \quad S \xrightarrow{\theta} S\theta$$

Example 4.48 (Bound for Figure 4.3c Continued). For the initial limit problem from Example 4.37, we now have

$$\begin{aligned} \{tv^+, (\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x - tv)^{+!}\} &\xrightarrow{\{tv/\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x - 1\}} \{(\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x - 1)^{+!}, 1^{+!}\} \\ &\rightsquigarrow \{(\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x - 1)^{+!}\} \\ &\rightsquigarrow \{(\tfrac{1}{2} \cdot x^2 + \tfrac{1}{2} \cdot x)^{+!}, 1^{+!}\} \\ &\rightsquigarrow^* \{x^{+!}\} \end{aligned}$$

i.e., we obtain the trivial limit problem $\{x^{+!}\}$. Note that, given an integer substitution σ , $\{tv/\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1\} \diamond \sigma$ is indeed an integer substitution, since $\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1 \in \mathbb{Z}$ for each $x \in \mathbb{Z}$.

While Definition 4.47 requires that variables may only be instantiated by integer terms, it is also useful to handle limit problems that contain non-integer terms.

Example 4.49 (Non-Integer Metering Functions Continued). By chaining the accelerated rule from Example 4.18 with the only initial rule, we obtain the rule

$$f_0(x) \xrightarrow{tv+1} f(x - 2 \cdot tv) \quad [0 < tv < \tfrac{1}{2} \cdot x + 1].$$

³The other rules for \rightsquigarrow are implicitly labeled with the identical substitution id .

CHAPTER 4. LOWER BOUNDS FOR ITSS

For the initial limit problem $\{tv^+, (\frac{1}{2} \cdot x - tv + 1)^{+!}, (tv + 1)^+\}$ we get

$$\begin{array}{ccc} \{tv^+, (\frac{1}{2} \cdot x - tv + 1)^{+!}, (tv + 1)^+\} & \rightsquigarrow^2 & \{tv^+, (\frac{1}{2} \cdot x - tv + 1)^{+!}\} \\ & \xrightarrow[\text{wavy line}]{\{x/2, tv-1\}} & \{tv^+, \frac{1}{2}^{+!}\} \\ & \rightsquigarrow & \{tv^+, 1^{+!}, 2^{+!}\} \\ & \rightsquigarrow^2 & \{tv^+\} \end{array}$$

using the positive limit vector $(+!, +!)$ for division. This allows us to infer $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(m)$.

So far, it is unclear how to check the side-condition that $\theta \diamond \sigma$ has to be an integer substitution for every integer substitution σ automatically. If the range of θ consists of univariate polynomials, then we can exploit the following observation from [33].

Lemma 4.50 (Polynomials Mapping to \mathbb{Z}). *Let $f : \mathbb{Z} \rightarrow \mathbb{R}$ be a polynomial of degree d such that $f(i), f(i+1), \dots, f(i+d+1) \in \mathbb{Z}$ for some $i \in \mathbb{Z}$. Then $\text{img}(f) \subseteq \mathbb{Z}$.*

Proof. We use induction on d . If $d = 0$, then f is a constant and thus the claim is trivial. If $d = n$, then $g(x) = f(x+1) - f(x)$ is a polynomial of degree $d-1$. Moreover, we have $g(i), \dots, g(i+d) \in \mathbb{Z}$. Thus, by the induction hypothesis, we have $\text{img}(g) \subseteq \mathbb{Z}$. This means that we have $f(x+1) - f(x) \in \mathbb{Z}$ for all $x \in \mathbb{Z}$. Since we also have $f(i) \in \mathbb{Z}$, this proves $\text{img}(f) \subseteq \mathbb{Z}$. \square

Thus, if $x\theta$ is a univariate polynomial of degree d , then it suffices to check if instantiating $x\theta$ with $0, \dots, d+1$ results in an integer. Moreover, $x\theta$ clearly maps to \mathbb{Z} if it neither contains division nor exponentiation. Thus, one can implement Definition 4.47 by restricting θ to these cases (but, of course, one may also incorporate further sufficient criteria).

However, up to now we cannot prove that, e.g., a rule α with $\text{guard}(\alpha) = x^2 - x > 0$ and $\text{cost}(\alpha) = x$ has a linear lower bound, since $(+, +)$ is not an increasing limit vector for subtraction. To handle such cases, the following rules allow us to neglect polynomial sub-expressions if they are “dominated” by other polynomials of higher degree or by exponential sub-expressions.

Definition 4.51 (\rightsquigarrow Continued). Let S be a limit problem, let $\pm \in \{+, -\}$, and let a, b, e be univariate polynomials. Then we have:

- (D) $S \cup \{(a \pm b)^\bullet\} \rightsquigarrow S \cup \{a^\bullet\}$ if $\bullet \in \{+, -\}$ and the degree of a is greater than the degree of b
- (E) $S \cup \{(a^e \pm b)^+\} \rightsquigarrow S \cup \{(a-1)^\bullet, e^+\}$ if $\bullet \in \{+, +!\}$

4.5. ASYMPTOTIC LOWER BOUNDS

Thus,

$$\{(x^2 - x)^+\} \rightsquigarrow \{(x^2)^+\} = \{(x \cdot x)^+\} \rightsquigarrow \{x^+\}$$

by the increasing limit vector $(+, +)$ for multiplication. Similarly,

$$\{(2^x - x^3)^+\} \rightsquigarrow \{(2 - 1)^{+1}, x^+\} \rightsquigarrow \{x^+\}.$$

Rule (E) can also be used to handle problems like $(a^e)^+$ (by choosing $b = 0$).

Example 4.52. We continue Example 4.28, where we obtain the initial limit problem $\{(2^{\frac{1}{2} \cdot x - 1} - 1)^+, (x - 1)^+\}$. We get:

$$\begin{aligned} \{(2^{\frac{1}{2} \cdot x - 1} - 1)^+, (x - 1)^+\} &\rightsquigarrow^2 \{(2^{\frac{1}{2} \cdot x - 1} - 1)^+, x^+\} \\ &\rightsquigarrow \{1^{+1}, (\tfrac{1}{2} \cdot x - 1)^+, x^+\} \\ &\rightsquigarrow^* \{x^+\} \end{aligned}$$

Thus, the accelerated Fibonacci rule from Example 4.28 gives rise to a lower bound in $\Omega(2^{\frac{1}{2} \cdot m - 1} - 1) = \Omega(2^{\frac{1}{2} \cdot m}) = \Omega(\sqrt{2}^m) \subset \Omega(1.4^m)$.

Theorem 4.53 states that \rightsquigarrow is indeed correct. When constructing the solution from the resulting trivial limit problem, one has to take the substitutions into account which were used in the derivation.

Theorem 4.53 (Correctness of \rightsquigarrow). *If $S \xrightarrow{\theta} S'$ and the family σ_m is a solution of S' , then $\theta \diamond \sigma_m$ is a solution of S .*

Proof. To prove the theorem, we consider Definition 4.45 (A) – (E) separately.

Claim 1. Definition 4.45 (A) is sound.

Assume that the step from S to S' was done by Definition 4.45 (A). Since σ_m is a solution for S' , it is a solution for $a_1^{\bullet_1}$ and $a_2^{\bullet_2}$, where (\bullet_1, \bullet_2) is an increasing (resp. decreasing, positive, or negative) limit vector for f . As σ_m is a solution for both $a_i^{\bullet_i}$, the function $\lambda m. a_i \sigma_m$ satisfies \bullet_i . By the definition of limit vectors, this implies that $\lambda m. f(a_1 \sigma_m, a_2 \sigma_m) = \lambda m. f(a_1, a_2) \sigma_m$ satisfies \bullet . Thus, σ_m is a solution for $f(a_1, a_2)^{\bullet}$.

Claim 2. Definition 4.45 (B) is sound.

If the step from S to S' was done by Definition 4.45 (B), then every solution σ_m for S' is also a solution for S , since $k \sigma_m = k$ holds for any $k \in \mathbb{Z}$.

Claim 3. Definition 4.45 (C) is sound.

If the step from S to S' was done by Definition 4.47 (C), then let σ_m be a solution for $S' = S\theta$. Then for every $(a\theta)^{\bullet} \in S\theta$, $\lambda m. a\theta \sigma_m$ satisfies \bullet and hence $\theta \diamond \sigma_m$ is a solution for a^{\bullet} . Thus, $\theta \diamond \sigma_m$ is a solution for S .

Claim 4. Definition 4.45 (D) is sound.

If the step from S to S' was done by Definition 4.51 (D), then let σ_m be a

CHAPTER 4. LOWER BOUNDS FOR ITSS

solution for a^\bullet . Since the only variable of the polynomial a is x , we must have $\lim_{m \mapsto \omega} x\sigma_m = \omega$ or $\lim_{m \mapsto \omega} x\sigma_m = -\omega$. W.l.o.g, let $\lim_{m \mapsto \omega} x\sigma_m = \omega$ and $\bullet = +$ (the other cases work analogously). Then $\lim_{m \mapsto \omega} a\sigma_m = \omega$ implies $\lim_{x \mapsto \omega} a = \omega$. Since the degree of a is greater than the degree of b , this means $\lim_{x \mapsto \omega} a \pm b = \omega$ and hence $\lim_{m \mapsto \omega} (a \pm b)\sigma_m = \omega$.

Claim 5. Definition 4.45 (E) is sound.

For Definition 4.51 (E), the proof is analogous. Here for large enough m , $a^e\sigma_m$ is an exponential function with a base > 1 . Since σ_m is a solution for e^+ , we again have $\lim_{m \mapsto \omega} x\sigma_m = \omega$ or $\lim_{m \mapsto \omega} x\sigma_m = -\omega$. Thus $a^e\sigma_m$ is an exponential function which grows faster than $b\sigma_m$ for $m \mapsto \omega$. Hence, we obtain $\lim_{m \mapsto \omega} (a^e \pm b)\sigma_m = \omega$. \square

Example 4.54 (Bound for Figure 4.3c Continued). Example 4.48 leads to the solution $\theta \diamond \sigma'_m$ of the initial limit problem for the program from Figure 4.3c where $\theta = \{tv/\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1\}$, $x\sigma'_m = m$, and $(tv)\sigma'_m = y\sigma'_m = z\sigma'_m = u\sigma'_m = 0$. Hence, $\theta \diamond \sigma'_m = \sigma_m$ where σ_m is as in Example 4.37. As explained in Example 4.42, this proves $\text{rc}_{cp(\mathcal{P})}(m) \in \Omega(m^4)$.

So we start with an initial limit problem $S = \{a_1^{\bullet 1}, \dots, a_k^{\bullet k}, \text{cost}(\alpha)^+\}$ that represents $\text{guard}(\alpha)$ and requires non-constant costs, and transform S with \rightsquigarrow into a trivial limit problem S' , i.e., $S \xrightarrow{\theta_1} \dots \xrightarrow{\theta_k} S'$. For automation, one should leave the \bullet_i in the initial problem S open, and only instantiate them by a value from $\{+, +_1\}$ when this is needed to apply a particular rule for the transformation \rightsquigarrow . Then the resulting family $\sigma_m^{S'}$ of valuations gives rise to a solution $\sigma_m = \theta_1 \diamond \theta_2 \diamond \dots \diamond \sigma_m^{S'}$ of S . Thus, we have $\text{rc}_{cp(\mathcal{P})}(\|\text{lhs}(\alpha)\sigma_m\|_i) \in \Omega(\text{cost}(\alpha)\sigma_m)$, which leads to a lower bound for $\text{rc}_{cp(\mathcal{P})}(m)$ with Lemma 4.41.

Our implementation uses the following strategy to apply the rules from Definitions 4.45, 4.47, and 4.51 for \rightsquigarrow . Initially, we reduce the number of variables by propagating bounds implied by the guard, i.e., if $\gamma \implies x \geq a$ resp. $\gamma \implies x \leq a$ (but $\gamma \not\Rightarrow x \geq a + 1$ resp. $\gamma \not\Rightarrow x \geq a - 1$) for some arithmetic expression a with $x \notin \mathcal{V}(a)$, then we apply the substitution $\{x/a\}$ to the initial limit problem by rule (C). For example, we simplify the limit problem from Example 4.43 by instantiating x with $y^2 + 1$, as the guard of the corresponding rule implies $x > y^2$. So here, we get

$$\{(x - y^2)^{+_1}, y^+\} \xrightarrow{\{x/y^2+1\}} \{1^{+_1}, y^+\} \rightsquigarrow \{y^+\}.$$

Afterwards, we use (B) and (D) with highest and (E) with second highest priority. The third priority is trying to apply (A) to univariate terms (since processing univariate terms helps to guide the search). As fourth priority, we apply (C) with a substitution $\{x/k\}$ if x^{+_1} or x^{-_1} in S , where we use SMT solving to find a suitable $k \in \mathbb{Z}$. Otherwise, we apply (A) to multivariate terms. Since \rightsquigarrow is well founded and, except for (C), finitely branching, one may also backtrack and explore alternative applications of \rightsquigarrow . In particular, we backtrack

4.5. ASYMPTOTIC LOWER BOUNDS

if we obtain a contradictory limit problem. Moreover, if we obtain a trivial S' where $\text{cost}(\alpha)\sigma_m$ is a polynomial, but $\text{cost}(\alpha)$ is a polynomial of higher degree or an exponential function, then we backtrack to search for other solutions which might lead to a higher lower bound. However, our implementation can of course fail, since solvability of limit problems is undecidable (due to Hilbert's Tenth Problem).

4.6 Solving Limit Problems via SMT

While the calculus presented in Section 4.5.2 enables a precise analysis of simplified ITSs, it is also quite expensive in practice. The reason is that the next \rightsquigarrow -step is rarely unique and thus backtracking is often unavoidable in order to find a good solution. We now show how limit problems can be encoded as conjunctions of polynomial inequalities in many cases. This allows us to use SMT solvers to solve limit problems more efficiently.

Essentially, the idea is to replace each variable $x \in \mathcal{V}$ with a linear template polynomial $c_x \cdot m + k_x$ where m is the parameter of the desired family of integer substitutions σ_m and c_x and k_x are abstract coefficients. Here, m is implicitly universally quantified over \mathbb{N} and c_x and k_x are existentially quantified over \mathbb{Z} . Thus, if a is a polynomial over \mathcal{V} , then $a_m = a\{x/c_x \cdot m + k_x \mid x \in \mathcal{V}\}$ is a univariate polynomial over m with abstract coefficients. Moreover, if a is of degree d , then a_m can be rearranged to the form $c_d \cdot m^d + \dots + c_0 \cdot m^0$ where we have $\bigcup_{i=0}^d \mathcal{V}(c_i) \subseteq \{c_x, k_x \mid x \in \mathcal{V}\}$.

Clearly, we have $\lim_{m \rightarrow \omega} a_m = \omega$ (resp. $-\omega$) if and only if $c_i > 0$ (resp. $c_i < 0$) for some $i > 0$ and $a_j = 0$ for all $j \in \{i+1, \dots, d\}$. Similarly, $\lim_{m \rightarrow \omega} a_m$ is a positive (resp. negative) constant if and only if $c_i = 0$ for all $i \in \{1, \dots, d\}$ and $c_0 > 0$ (resp. $c_0 < 0$).

Definition 4.55 (SMT Encoding of Limit Problems). Let a be a polynomial of degree d and let $a_m = c_d \cdot m^d + \dots + c_0 \cdot m^0$. We define

$$\text{smt}(a^\bullet) = \begin{cases} \bigvee_{i=1}^d \left(c_i > 0 \wedge \bigwedge_{j=i+1}^d c_j = 0 \right) & \text{if } \bullet = + \\ \bigvee_{i=1}^d \left(c_i < 0 \wedge \bigwedge_{j=i+1}^d c_j = 0 \right) & \text{if } \bullet = - \\ \bigwedge_{j=1}^d c_j = 0 \wedge c_0 > 0 & \text{if } \bullet = +! \\ \bigwedge_{j=1}^d c_j = 0 \wedge c_0 < 0 & \text{if } \bullet = -! \end{cases}$$

We lift smt to limit problems S where a is a polynomial for each $a^\bullet \in S$ by defining $\text{smt}(S) = \bigwedge_{a^\bullet \in S} \text{smt}(a^\bullet)$. Furthermore, given polynomial costs c of degree d with

$$c_m = c\{x/c_x \cdot m + k_x \mid x \in \mathcal{V}\} = c_d \cdot m^d + \dots + c_0 \cdot m^0,$$

we define $\text{smt}_{c,i}(S) = \text{smt}(S) \wedge c_i > 0$ for each $i \in \{1, \dots, d\}$.

To solve a limit problem S , it suffices to find a solution for $\text{smt}(S)$. However, to maximize the costs c , one should try to find a solution for $\text{smt}_{c,i}(S)$ where i is as large as possible. The reason is that the resulting solution for S allows us to prove a polynomial lower bound of degree i via Theorem 4.38 and Lemma 4.41.

4.6. SOLVING LIMIT PROBLEMS VIA SMT

Example 4.56 (Encoding the Initial Limit Problem for Figure 4.3c). We show how to encode the initial limit problem

$$\left\{ tv^+, \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv \right)^{+!} \right\}$$

from Example 4.37.⁴ We have $tv_m = c_{tv} \cdot m + k_{tv}$ and $\text{smt}(tv^+) = c_{tv} > 0$. Moreover, we have

$$\begin{aligned} & \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv \right)_m \\ &= \frac{1}{2} \cdot (c_x \cdot m + k_x)^2 + \frac{1}{2} \cdot (c_x \cdot m + k_x) - (c_{tv} \cdot m + k_{tv}) \\ &= c_2 \cdot m^2 + c_1 \cdot m + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= \frac{1}{2} \cdot c_x^2, \\ c_1 &= c_x \cdot k_x + \frac{1}{2} \cdot c_x - c_{tv}, \text{ and} \\ c_0 &= k_x^2 + k_x - k_{tv} \end{aligned}$$

and thus

$$\text{smt} \left(\left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv \right)^{+!} \right) = (c_2 = c_1 = 0 \wedge c_0 > 0).$$

Hence, we have

$$\text{smt} \left(\left\{ tv^+, \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv \right)^{+!} \right\} \right) = (c_{tv} > 0 \wedge c_2 = c_1 = 0 \wedge c_0 > 0).$$

State-of-the-art SMT solvers can prove unsatisfiability of

$$\text{smt} \left(\left\{ tv^+, \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - tv \right)^{+!} \right\} \right)$$

within milliseconds. This is not surprising, since we instantiated tv with a non-linear expression in Example 4.48 in order to find a solution, but Definition 4.55 replaces tv with a *linear* template polynomial. Thus, even if all arithmetic expressions in the analyzed limit problem are polynomials, \rightsquigarrow is still required, i.e., our SMT based technique does not subsume the calculus presented in

⁴For reasons of simplicity, we do not take the costs from Figure 4.3c into account.

CHAPTER 4. LOWER BOUNDS FOR ITSS

Section 4.5.2.⁵

Example 4.57 (Encoding the Simplified Limit Problem for Figure 4.3c). After instantiating tv in Example 4.48, we obtain the limit problem

$$\left\{ \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1 \right)^+ \right\}.$$

We have

$$\begin{aligned} & \left(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1 \right)_m \\ &= \frac{1}{2} \cdot (c_x \cdot m + k_x)^2 + \frac{1}{2} \cdot (c_x \cdot m + k_x) - 1 \\ &= c_2 \cdot m^2 + c_1 \cdot m + c_0 \end{aligned}$$

where $c_2 = \frac{1}{2} \cdot c_x^2$ and $c_1 = c_x \cdot k_x + \frac{1}{2} \cdot c_x$.

Thus, we have $\text{smt}((\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x)^+) = ((c_2 > 0) \vee (c_2 = 0 \wedge c_1 > 0))$. State-of-the-art SMT solvers can easily find an appropriate solution like, e.g., $\{c_x/1, k_x/0\}$.

The following theorem shows how a solution for $\text{smt}(S)$ can be used to obtain a solution for S .

Theorem 4.58 (Solving Limit Problems via SMT). *Let S be a limit problem such that a is a polynomial for each $a^\bullet \in S$ and let σ be an integer substitution such that $\sigma \models \text{smt}(S)$. Then*

$$\sigma_m = \{x/c_x \sigma \cdot m + k_x \sigma \mid x \in \mathcal{V}\}$$

is a solution for S .

Proof. First note that σ_m is clearly an integer substitution for each $m \in \mathbb{N}$. Thus, to prove the theorem, it suffices to prove that $\lambda m. a \sigma_m$ satisfies $a^\bullet \in S$ for an arbitrary but fixed $a^\bullet \in S$. Let d be the degree of a . Then we have $a_m = c_d \cdot m^d + \dots + c_0 \cdot m^0$.

Case 1. $\bullet = +$

Then $\sigma \models \bigvee_{i=1}^d (c_i > 0 \wedge \bigwedge_{j=i+1}^d c_j = 0)$. Hence, we have

$$a_m \sigma = c_i \sigma \cdot m^i + \dots + c_0 \sigma \cdot m^0$$

where $c_i \sigma > 0$ for some $i \in \{1, \dots, d\}$. Thus, we have $\lim_{m \rightarrow \omega} a_m \sigma = \omega$. By

⁵We could also use non-linear templates, but in any case the degree of the template has to be fixed in advance and thus may be insufficient for the problem at hand.

4.6. SOLVING LIMIT PROBLEMS VIA SMT

construction, we have

$$a_m \sigma = a\{x/c_x \sigma \cdot m + k_x \sigma \mid x \in \mathcal{V}\} = a\sigma_m. \quad (4.18)$$

Thus, we have $\lim_{m \mapsto \omega} a\sigma_m = \omega$, i.e., $\lambda m. a\sigma_m$ satisfies a^+ .

Case 2. $\bullet = -$

Analogous to *Case 1*.

Case 3. $\bullet = +!$

Then $\sigma \models \bigwedge_{j=1}^d c_j = 0 \wedge c_0 > 0$. Hence, we have

$$\lim_{m \mapsto \omega} a_m \sigma = \lim_{m \mapsto \omega} c_0 \sigma = c_0 \sigma > 0.$$

With (4.18), we get $\lim_{m \mapsto \omega} a\sigma_m = c_0 \sigma > 0$, i.e., $\lambda m. a\sigma_m$ satisfies $a^{+!}$.

Case 4. $\bullet = -!$

Analogous to *Case 3*.

□

Note that Theorem 4.58 can be integrated into the calculus from Section 4.5.2 seamlessly. Whenever the current limit problem S satisfies the prerequisites of Theorem 4.58, one tries to find a solution for $\text{smt}_{c,i}(S)$ where i is initially set to the degree of c and decremented until a solution is found. As soon as a solution σ_m is found, one can either return σ_m or keep searching for a better solution. To this end, one can either backtrack or keep simplifying S via \rightsquigarrow . Similarly, if the SMT solver does not find a solution one can either backtrack or keep simplifying S via \rightsquigarrow .

However, note that the intention of Theorem 4.58 and its integration into \rightsquigarrow is not to add additional power to \rightsquigarrow . Instead, the goal is to delegate the search for a solution to existing tools instead of relying on heuristics as often as possible.

Example 4.59 (Solving the Simplified Limit Problem for Figure 4.3c). In Example 4.57, we saw that $\{c_x/1, k_x/0\} \models \text{smt}((\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1)^+)$. Hence, according to Theorem 4.58, $\sigma_m = \{x/m\}$ is a solution for $\{(\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x - 1)^+\}$.

4.7 Related Work

As discussed in Section 1.4, there are many techniques to infer worst-case upper bounds and some techniques to infer best-case lower bounds on the complexity of integer programs, but the presented technique is the first to infer worst-case lower bounds. From these techniques, here we just discuss [3], as it has several similarities to our analysis.

Like our approach, it uses recurrence solving to compute a closed form for the costs of several consecutive applications of a cost equation with direct recursion (which corresponds to a simple – potentially non-linear – loop in our setting). As the analysis from [3] is over-approximating in the sense that it has to reason about all program runs, there the handling of non-determinism is challenging. To deal with this issue, they show how to over- and under-approximate (for the inference of worst-case upper and best-case lower bounds, respectively) the costs of several consecutive applications of cost equations with linear or geometric progression behavior via standard recurrence equations. In contrast, we treat temporary variables, which we use to model non-determinism, as constants when computing the iterated update and costs. Thus, our iterated update and costs are only valid for consecutive applications of simple loops where temporary variables are instantiated with the same values in each iteration. This restriction is sound in our setting, as it is under-approximating in the sense that it suffices to prove the existence of a certain family of program runs, i.e., we do not have to reason about all program runs. To reason about evaluations where the valuation of the temporary variables changes, we can instantiate them with expressions containing program variables via *Strengthening* (Theorem 4.20).

In contrast to our approach from Theorem 4.27, the techniques for cost equations with multiple recursive calls and non-constant costs from [3] are more sophisticated. The reason is that their approach to deduce that a cost equation has linear or geometric progression behavior naturally applies to cost equations with multiple recursive calls. In contrast, our computation of the iterated update relies on the existence of a single, deterministic update and hence is not applicable to simple non-linear loops, which also prevents us from computing the iterated costs for such loops. Thus, our handling of simple non-linear loops may be improved by incorporating ideas from [3].

Finally, [3] also uses ranking functions to over-estimate the height of evaluation trees to analyze cost equations with multiple recursive calls. This approach is closely related to our adaption of metering functions to simple non-linear loops, as we use metering functions to under-estimate the height up to which evaluation trees are complete.

Apart from techniques for the computation of symbolic bounds, [32] presents a technique to generate test-cases that trigger the worst-case execution time of programs. The idea is to execute the program for small inputs, observe the required runtime, and then generalize those inputs that lead to expensive runs. In this way, one obtains *generators* which can be used to construct larger

4.7. RELATED WORK

inputs that presumably result in expensive runs. In contrast to the technique presented in the current chapter, [32] operates on **Java**, i.e., it also supports data-structures. However, [32] does not try to infer symbolic bounds, which is the main purpose of our technique. In fact, ideas from [32] could be incorporated into our framework. For example, a similar approach could be used in order to apply *Strengthening* (cf. Theorem 4.20) in a way that leads to expensive runs.

To simplify programs, we use a variant of *loop acceleration* to summarize the effect of applying a loop repeatedly. Acceleration is mostly used in over-approximating settings (e.g., [46, 69, 91, 99]), where handling non-determinism is challenging, as loop summaries have to cover *all* possible non-deterministic choices. However, our technique is under-approximating, i.e., we can instantiate non-deterministic values arbitrarily.

In contrast to the under-approximating acceleration technique in [95], instead of quantifier elimination we use an adaptation of ranking functions to underestimate the number of loop iterations symbolically.

Another approach which is related to our acceleration technique is [24], which identifies cases where the transitive closure of relations can be computed precisely. This is similar to our computation of the iterated update, cf. Section 4.3.1. There, we compute a closed form of $x\eta^n$ (i.e., the n -fold application of the update to x) for every program variable x . Thus, we know the value of $x\eta^n$ for each $n \in \mathbb{N}$, which we exploit later on by instantiating n with a fresh variable that represents the number of loop iterations. In contrast, the transitive closure $\bigcup_{n \in \mathbb{N}} x\eta^n$ is *not* sufficient for our use-case, since it does not provide any information about the value of x after a given (symbolic) number of iterations.

The paper [4] presents a technique to infer asymptotic bounds from concrete bounds with a so-called context constraint φ , i.e., bounds of the form $\varphi \implies rt \leq e$ or $\varphi \implies rt \geq e$. Here, rt is the runtime of the program and e is a *cost expression*, i.e., an expression built from linear expressions, addition, multiplication, maximum, logarithm, and exponentiation. To avoid negative values, all occurrences of linear expressions ℓ in cost expressions have to be of the form $\max(\ell, 0)$. Moreover, exponentiation is only allowed with positive natural numbers as base. Thus, the expressions which are supported by [4] and our technique from Sections 4.5 and 4.6 are orthogonal. Moreover, [4] infers multi-variate asymptotic bounds, whereas our technique infers univariate bounds which are only parameterized in the size of the input. Finally, [4] does not aim to eliminate the context constraint, i.e., the resulting asymptotic bounds are of the form $\varphi \implies rt \in \mathcal{O}(e)$ or $\varphi \implies rt \in \Omega(e)$. In contrast, eliminating such context constraints is one of the main motivations for our technique to deduce asymptotic bounds from concrete bounds.

4.8 Experiments

Our implementation **LoAT** (“**L**ower **B**ounds **A**nalysis **T**ool”) is freely available at [102]. We evaluated it on the benchmarks [20] from the evaluation of [27]. We omitted 50 non-linear programs, since the extension of our technique to non-linear ITSs from Section 4.4 is not yet implemented. Moreover, we omitted 15 duplicates. As we know of no other tool to compute worst-case lower bounds for integer programs, we compared our results with the asymptotically smallest results of leading tools for upper bounds: **KoAT**, **CoFloCo** [48, 50], **Loopus** [113], **PUBS** [3], and **RanK** [8]. The results are presented in Table 4.1, where the results of the tools for upper bounds were taken from the evaluation of [27]. We did not compare our results with the best-case lower bounds computed by **CoFloCo** and **PUBS**, as such a comparison would be meaningless since the worst-case lower bounds computed by **LoAT** are no valid best-case lower bounds. We used a timeout of 60 seconds. In the following, we disregard 132 examples where $\text{rc}_{cp(\mathcal{P})}(n) \in \mathcal{O}(1)$ was proved since there is no non-trivial lower bound in these cases.

LoAT infers non-trivial lower bounds for 393 (80%) of the remaining 494 examples. *Tight* bounds (i.e., the lower and the upper bound coincide) are proved in 345 cases (70%). Whenever an exponential upper bound is proved, **LoAT** also proves an exponential lower bound (i.e., $\text{rc}_{cp(\mathcal{P})}(n) \in \Omega(k^n)$ for some $k > 1$). In 176 cases, **LoAT** infers unbounded runtime complexity. In some cases, this is due to non-termination, but for this particular goal, specialized tools are more powerful (e.g., whenever **LoAT** proves unbounded runtime complexity due to non-termination, the termination analyzer **T2** [26] shows non-termination as well). The average runtime of **LoAT** was 5.3 seconds per successfully analyzed example. These results could be improved further by supplementing **LoAT** with invariant inference as implemented in tools like **APRON** [90].

Without the SMT encoding for limit problems from Section 4.6, the results for the examples from [20] only differ marginally, cf. Table 4.2. However, we used a preliminary implementation of the SMT encoding in our experiments which only applies to limit problems where all expressions are *linear* polynomials. While the latest **LoAT** version (Git revision 2a41dff) uses the full SMT encoding

| | | LoAT | | | | | | |
|------------------|-----------------------|-------------|-------------|---------------|---------------|---------------|-------|------------------|
| Best Upper Bound | $\text{rc}(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | (132) | – | – | – | – | – | – |
| | $\mathcal{O}(n)$ | 37 | 126 | – | – | – | – | – |
| | $\mathcal{O}(n^2)$ | 8 | 14 | 35 | – | – | – | – |
| | $\mathcal{O}(n^3)$ | 2 | – | 2 | 1 | – | – | – |
| | $\mathcal{O}(n^4)$ | 1 | – | – | – | 2 | – | – |
| | EXP | – | – | – | – | – | 5 | – |
| | $\mathcal{O}(\omega)$ | 53 | 31 | 1 | – | – | – | 176 |

Table 4.1: Best Upper Bound vs. **LoAT**

4.8. EXPERIMENTS

| LoAT SMT | | | | | | | | |
|----------|------------------|-------------|-------------|---------------|---------------|---------------|-------|------------------|
| LoAT | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | EXP | $\Omega(\omega)$ |
| | $\Omega(1)$ | 149 | 4 | — | — | — | — | 2 |
| | $\Omega(n)$ | — | 166 | — | — | — | — | 1 |
| | $\Omega(n^2)$ | — | — | 38 | — | — | — | — |
| | $\Omega(n^3)$ | — | — | — | 1 | — | — | — |
| | $\Omega(n^4)$ | — | — | — | — | 2 | — | — |
| | EXP | — | — | — | — | — | 5 | — |
| | $\Omega(\omega)$ | — | 1 | — | — | — | — | 173 |

Table 4.2: LoAT vs. LoAT SMT

as presented in Section 4.6, the experiments could not be repeated for reasons of time. Moreover, as mentioned in Section 1.3.1, LoAT’s SMT encoding was designed with the particular use-case of finding denial of service vulnerabilities in Java programs in mind. Thus, we used LoAT as backend for AProVE’s transformation from Java Bytecode (JBC) programs to ITSs presented in [51] to see if the SMT encoding for limit problems improves LoAT’s performance when analyzing Java programs. Note, however, that the transformation from [51] is sound for upper bounds, but in general it is unsound for lower bounds. Thus, coupling the transformation from [51] with LoAT is only a heuristic: If LoAT identifies an expensive family of ITS-runs, then one still has to check if these ITS-runs are spurious or if they correspond to runs of the original Java program.

To test the combination of [51] and LoAT, we analyzed the 272 non-recursive Java programs from the *Termination Problems Data Base* [122]. We didn’t include

| LoAT JBC SMT | | | | | |
|--------------|------------------|-------------|-------------|---------------|------------------|
| LoAT JBC | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(\omega)$ |
| | $\Omega(1)$ | 130 | 35 | 1 | — |
| | $\Omega(n)$ | — | 29 | — | — |
| | $\Omega(n^2)$ | — | — | 3 | — |
| | $\Omega(\omega)$ | 1 | 1 | — | 72 |

Table 4.3: LoAT JBC vs. LoAT JBC SMT

| LoAT JBC SMT | | | | | | | |
|--------------|-----------------------|-------------|-------------|---------------|---------------|---------------|------------------|
| AProVE | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^4)$ | $\Omega(n^8)$ | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 28 | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 54 | 42 | — | — | — | — |
| | $\mathcal{O}(n^2)$ | 7 | 7 | 3 | — | — | — |
| | $\mathcal{O}(n^4)$ | 2 | 2 | — | — | — | — |
| | $\mathcal{O}(n^8)$ | 1 | — | — | — | — | — |
| | $\mathcal{O}(\omega)$ | 67 | 14 | 1 | — | — | 72 |

Table 4.4: AProVE vs. LoAT JBC SMT

CHAPTER 4. LOWER BOUNDS FOR ITSs

any recursive **Java** programs, since the transformation from [51] is restricted to non-recursive programs. Moreover, we excluded 28 examples where **AProVE** can infer a constant upper bound. We used an overall timeout of 120 seconds and a timeout of 60 seconds for **LoAT** (i.e., **LoAT** had 60 seconds to analyze each resulting ITS, unless the transformation from [51] took more than 60 seconds, which rarely happens in practice). The results are presented in Table 4.3 and Table 4.4. Table 4.3 compares versions of **LoAT** with (**LoAT JBC SMT**) and without (**LoAT JBC**) the SMT encoding from Section 4.6. Table 4.4 compares the lower bounds inferred by **LoAT JBC SMT** with the upper bounds proven by **AProVE**. **LoAT JBC** proved unbounded runtime complexity in 74 cases and it inferred 3 quadratic and 29 linear lower bounds, i.e., **LoAT** inferred non-trivial bounds for 39% of the analyzed ITSs. **LoAT JBC SMT** proved unbounded runtime complexity in 72 cases and it inferred 4 quadratic and 65 linear lower bounds, i.e., non-trivial bounds were inferred in 52% of all cases. These bounds were tight in 117 cases (43%), i.e., in these cases, the lower bound inferred by **LoAT** coincides with the upper bound inferred by **AProVE**. Moreover, enabling the SMT encoding decreased the average runtime from 17.0 to 12.7 seconds per successfully analyzed example. This clearly shows that the technique from Section 4.6 significantly improves **LoAT**'s performance for certain classes of ITSs.

LoAT uses the recurrence solver **PURRS** [18] and the SMT solver **Z3** [39]. The version of **LoAT** that we used as well as a version of **AProVE** to generate ITSs from **Java** programs are available at [52].

4.9 Conclusion and Future Work

We presented the first technique to infer lower bounds on the worst-case runtime complexity of integer transition systems, based on a modular program simplification framework. The main simplification technique is *loop acceleration*, which relies on *recurrence solving* and *metering functions*, an adaptation of classical ranking functions. By eliminating loops and locations via *chaining*, we eventually obtain *simplified programs*. We presented a technique to infer *asymptotic lower bounds* from simplified programs, which can also be used to find vulnerabilities. In comparison to the preliminary version from [56], we extended our program simplification framework to non-linear ITSs and we extended our technique to infer asymptotic lower bounds for simplified programs by an SMT encoding.

Our implementation **LoAT** is freely available at [102]. It was inspired by the tool **KoAT** [27], which alternates runtime and size analysis to infer *upper* bounds in a modular way. Similarly, **LoAT** alternates runtime analysis and recurrence solving to transform loops to non-looping rules independently. An experimental evaluation (Section 4.8) demonstrates the applicability of our technique in practice.

There are several interesting directions for future work. First of all, as mentioned in Section 4.8, one should couple **LoAT** with invariant inference techniques to improve its power. Furthermore, **LoAT**'s heuristics to apply *Strengthening* are very basic and should be improved, e.g., by incorporating ideas from [32]. Another interesting question is to what extent **LoAT** can benefit from more sophisticated techniques to infer metering functions. Possibilities include the inference of logarithmic or super-linear polynomial metering functions, but one could also adapt the *quasi-ranking functions* from [96] to our setting. Moreover, as mentioned in Section 4.7, ideas from [3] could be adapted to under-approximate the costs of repeatedly applying non-linear simple loops more precisely when accelerating them. Finally, lifting **LoAT** to integer rewrite systems, i.e., adding support for arbitrary recursion, would improve its applicability in practice.

Upper Bounds for Recursive Natural Transition Systems

In this chapter, we study another flavor of integer rewrite systems, namely *Recursive Natural Transition Systems* (RNTSs). While Chapter 4 was concerned with lower bounds, the goal of this chapter is to infer upper bounds on the runtime complexity of RNTSs. In contrast to ITSs, RNTSs just support arithmetic on natural numbers (instead of integers), but they allow nesting of function symbols on right-hand sides. Thus, RNTSs allow to pass the result of one function as a parameter to another function. Hence, the additional difficulty in contrast to the analysis of ITSs is to estimate the results of functions.

Consequently, we discuss how to obtain bounds for the result computed by a function using standard complexity analysis tools in Section 5.2 after introducing the necessary preliminaries in Section 5.1.

Then we show how RNTSs can be analyzed in a bottom-up fashion in Section 5.3. To this end, we repeatedly analyze the runtime and the result of program parts \mathcal{P} which, considered individually, are ITSs. Afterwards, we use the obtained bounds to eliminate calls to \mathcal{P} from the RNTS. By eliminating calls to \mathcal{P} , program parts which had nested function symbols on right-hand sides are transformed to ITSs and thus can be analyzed later on. The presented approach is completely modular, as it repeatedly finds bounds for parts of the RNTS and combines them.

Finally, we discuss related work in Section 5.4, we evaluate our implementation in AProVE in Section 5.5, and we conclude in Section 5.6.

In the context of program verification, our technique allows us to overcome the restrictions of various existing techniques w.r.t. recursion, cf. Section 1.2. Thereby, we limit ourselves to natural numbers, as the combination of integers and full recursion is particularly challenging. See Section 5.6 for a more detailed discussion of ideas and obstacles regarding an extension of our technique to integers.

5.1 Program Model

We start with the definition of the program model which we consider throughout this chapter.

Definition 5.1 (Recursive Natural Transition System (RNTS)). Let $\Sigma_{\mathbb{N}} = \{+, \cdot\} \cup \mathbb{N}$ and let Σ be a finite signature. An RNTS rule over Σ is of the form $f(\mathbf{x}) \xrightarrow{c} r[\varphi]$ where $f \in \Sigma$, \mathbf{x} is a vector of pairwise different variables, $c \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$, $r \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}}, \mathcal{V})$, and $\varphi \in \mathcal{Fml}(\mathcal{V})$. A *Recursive Natural Transition System* (RTNS) over Σ is a set of RNTS rules over Σ .

Now the transition relation of RNTSs can be defined analogously to ITSs.

Definition 5.2 (Natural Transition Relation). Let \mathcal{P} be an RNTS. We have $s \xrightarrow{\ell}_{\mathcal{P}} t$ if there is a context C , a rule $\ell \xrightarrow{c} r[\varphi] \in \mathcal{P}$, and a substitution $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma_{\mathbb{N}})$ such that $C[\ell\sigma] = s$, $C[r\sigma] = t$, $\sigma \models \varphi$, and $\llbracket c\sigma \rrbracket = \#$.

Note that the rewrite relation for RNTSs is innermost by construction. The reason is that the left-hand sides of RNTSs do not have nested function symbols and RNTS rules are only applicable if all arguments are arithmetic expressions.

Example 5.3. The program from Figure 4.1a can also be modeled by implementing each loop as a separate recursive function, resulting in an RNTS.

$$\begin{array}{llll}
 \beta_0 : & f_0(x) & \xrightarrow{1} & f_2(f_1(x), u) \\
 \beta_1 : & f_1(x) & \xrightarrow{1} & f_1(x') + x \quad [x > 0 \wedge x' = x - 1] \\
 \beta_2 : & f_1(x) & \xrightarrow{1} & 0 \quad [x = 0] \\
 \beta_3 : & f_2(z, u) & \xrightarrow{1} & f_2(z', f_3(z')) \quad [z > 0 \wedge z' = z - 1] \\
 \beta_4 : & f_3(u) & \xrightarrow{1} & f_3(u') \quad [u > 0 \wedge tv > 0 \wedge u' = u - tv] \\
 \beta_5 : & f_3(u) & \xrightarrow{1} & 0 \quad [u = 0]
 \end{array}$$

Note that y is just used as an accumulator in the first loop from Figure 4.1a. Hence, in the RNTS above, y is not required since we are no longer restricted to tail recursion and thus do not need an accumulator to store the result of f_1 , i.e., f_1 is now a unary function. The assignment $z' = y$ is modeled by passing the result of f_1 to f_2 in rule β_0 . Nesting f_1 below f_2 in rule β_0 also enforces the condition from Figure 4.1a that we must not leave the first loop until $x = 0$ is satisfied. The reason is that f_2 cannot be evaluated until the inner call to f_1 was evaluated to an arithmetic expression, which is only possible when $x = 0$ holds. Similarly, nesting f_3 below f_2 in rule β_3 corresponds to the condition that we cannot leave the inner loop until $u = 0$ holds in Figure 4.1a.

While Example 5.3 could easily be transformed to an ITS like the one from Figure 4.1c automatically, this is not true in general.

5.1. PROGRAM MODEL

Example 5.4. Consider the following RNTS, where `times` multiplies two natural numbers.

$$\begin{array}{llll}
 \gamma_0 : & \text{times}(x, y) & \xrightarrow{1} & \text{plus}(\text{times}(x', y), y) & [x > 0 \wedge x' = x - 1] \\
 \gamma_1 : & \text{times}(x, y) & \xrightarrow{1} & 0 & [x = 0] \\
 \gamma_2 : & \text{plus}(x, y) & \xrightarrow{1} & 1 + \text{plus}(x', y) & [x > 0 \wedge x' = x - 1] \\
 \gamma_3 : & \text{plus}(x, y) & \xrightarrow{1} & y & [x = 0]
 \end{array}$$

Transforming it to an ITS is non-trivial, since evaluating `times`(x, y) results in a term of the form $\underbrace{\text{plus}(\text{plus}(\dots 0, y), y)}_{x \times} \dots$ before a `plus` rule can be applied for the first time. Such stacks of function calls cannot be modeled with ITSs.

So in contrast to Example 5.3, the additional difficulty in Example 5.4 is that the call to the auxiliary function `plus` is *above* the recursive call to `times` in rule γ_0 (whereas the call to f_3 is *below* the recursive call to f_2 in rule β_3 from Example 5.3).

In this chapter, we focus on complexity problems where the start terms can be arbitrary int-basic terms (i.e., in contrast to Chapter 4 we do not fix a start symbol).¹ The reason is that, as mentioned in Section 1.3.2, the technique presented in this chapter was initially developed as backend for an abstraction from term rewrite systems and the established notion of complexity for term rewrite systems also does not require a start symbol. However, as our technique infers an upper bound for each function individually, this is not a restriction (i.e., it could also be used to analyze object-oriented or functional programs with a specific entry point via a transformation to RNTSs).

Definition 5.5 (Canonical Complexity Problem). Let \mathcal{P} be an RNTS over Σ . Its *canonical complexity problem* is $cp(\mathcal{P}) = (\mathcal{T}_{\text{basic}}(\Sigma), \rightarrow_{\mathcal{P}}, \|\cdot\|_i)$.

Our approach builds upon the idea of alternating between *runtime* and *size* analysis [27]. The key insight is to *summarize* functions by approximating their runtime and their result, and then to eliminate calls to them from the RNTS. In this way, our analysis decomposes the call graph of the RNTS into “blocks” of mutually recursive functions and exports each of these blocks into a separate ITS. Thus, in each analysis step it suffices to analyze just an ITS instead of an RNTS.

To see the motivation to approximate the result of functions (i.e., to infer *size bounds*), reconsider Example 5.4. Its runtime is cubic, as `plus` is linear in its first argument, which is instantiated with `times`(x', y), i.e., a value of size $(x - 1) \cdot y$ in rule γ_0 . In other words, to infer a cubic bound on `times`’s complexity, it is crucial to know that the result of `times` is quadratic, i.e., a quadratic size bound for `times` is required.

¹Note that it does not harm to allow start terms $f(\mathbf{n})$ where \mathbf{n} contains negative integers, since such terms are trivially normal forms.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

We use weakly monotonic runtime and size bounds to compose them easily when analyzing nested terms. To see why we need monotonicity, assume that the right-hand side of an RNTS rule has the form $f(x, g(z))$ where the result of g is linear in its only argument, but we obtained a (correct but imprecise) quadratic bound for g 's result. To estimate the cost of evaluating $f(x, g(z))$, our analysis essentially substitutes the size bound for g into the runtime bound for f . Thus, if we would allow non-monotonic runtime bounds like $x - y$ for $f(x, y)$, then our approximation of the cost of evaluating $f(x, g(z))$ would get *smaller* (and hence potentially incorrect) due to the imprecise bound for g 's result.

Definition 5.6 (Weakly Monotonic Functions). We say that $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is *weakly monotonically increasing* if $f(n_1, \dots, n_k) \leq f(n_1, \dots, n_i + 1, \dots, n_k)$ holds for all $n_1, \dots, n_k \in \mathbb{N}$ and all $i \in \{1, \dots, k\}$.

Then runtime resp. size bounds map each function symbol f to a weakly monotonic function which is an upper bound for the cost resp. the result of f .

Definition 5.7 (Runtime and Size Bounds). A function

$$\text{rt} : \Sigma \rightarrow \mathbb{N}^* \rightarrow \mathbb{N} \cup \{\omega\}$$

is a *runtime bound* for an RNTS \mathcal{P} if $\text{rt}(f)$ is weakly monotonically increasing and

$$\text{dh}_{\rightarrow_{\mathcal{P}}}(\mathbf{f}(\mathbf{n})) \leq \text{rt}(f)(\mathbf{n})$$

for all $k \in \mathbb{N}$, $f \in \Sigma^k$, and $\mathbf{n} \in \mathbb{N}^k$. Similarly,

$$\text{sz} : \Sigma \rightarrow \mathbb{N}^* \rightarrow \mathbb{N} \cup \{\omega\}$$

is a *size bound* for \mathcal{P} if $\text{sz}(f)$ is weakly monotonically increasing and

$$\mathbf{f}(\mathbf{n}) \rightarrow_{\mathcal{P}}^* n \text{ implies } n \leq \text{sz}(f)(\mathbf{n})$$

for all $k \in \mathbb{N}$, $f \in \Sigma^k$, $\mathbf{n} \in \mathbb{N}^k$, and $n \in \mathcal{T}(\Sigma_{\mathbb{N}})$.

To ensure monotonicity, we only use runtime and size bounds which are built from variables, $\Sigma_{\mathbb{N}}$, and the special constant ω .

Example 5.8 (Size and Runtime Bounds – Example 5.3 continued). For the RNTS from Example 5.3, any function rt with

$$\begin{aligned} \text{rt}(f_0)(x) &\geq x^4 + 2 \cdot x + 2 \\ \text{rt}(f_1)(x) &\geq x + 1 \\ \text{rt}(f_2)(z, u) &\geq z^2 + z \\ \text{rt}(f_3)(u) &\geq u + 1 \end{aligned}$$

5.1. PROGRAM MODEL

is a runtime bound. Similarly, any sz with

$$\begin{aligned} \text{sz}(\mathbf{f}_0)(x) &\geq 0 \\ \text{sz}(\mathbf{f}_1)(x) &\geq x^2 \\ \text{sz}(\mathbf{f}_2)(z, u) &\geq 0 \\ \text{sz}(\mathbf{f}_3)(u) &\geq 0 \end{aligned}$$

is a size bound.

A runtime bound clearly gives rise to an upper bound on the runtime complexity.

Theorem 5.9 (rt and rc). *Let \mathcal{P} be an RNTS over Σ and let rt be a runtime bound for \mathcal{P} . Then for all $n \in \mathbb{N}$, we have*

$$\text{rc}_{cp(\mathcal{P})}(n) \leq \sup \left\{ \text{rt}(\mathbf{f})(\mathbf{n}) \mid k \in \mathbb{N}, \mathbf{f} \in \Sigma^k, \mathbf{n} \in \mathbb{N}^k, \sum |\mathbf{n}| \leq n \right\}.$$

So in particular, $\text{rc}_{cp(\mathcal{P})}(n) \in \mathcal{O}(\sum_{\mathbf{f} \in \Sigma} \text{rt}(\mathbf{f})(n, \dots, n))$.

Proof. For any $n \in \mathbb{N}$ we have

$$\begin{aligned} &\text{rc}_{cp(\mathcal{P})}(n) \\ &= \sup \{ \text{dh} \rightarrow_{\mathcal{P}}(t) \mid t \in \mathcal{T}_{\text{basic}}(\Sigma), \|t\|_i \leq n \} && \text{by Definition 2.18} \\ &= \sup \{ \text{dh} \rightarrow_{\mathcal{P}}(\mathbf{f}(\mathbf{n})) \mid k \in \mathbb{N}, \mathbf{f} \in \Sigma^k, \mathbf{n} \in \mathbb{N}^k, \sum |\mathbf{n}| \leq n \} && \text{by Definition 4.10} \\ &\leq \sup \{ \text{rt}(\mathbf{f})(\mathbf{n}) \mid k \in \mathbb{N}, \mathbf{f} \in \Sigma^k, \mathbf{n} \in \mathbb{N}^k, \sum |\mathbf{n}| \leq n \} && \text{by Definition 5.7} \end{aligned}$$

The second statement of the theorem follows by weak monotonicity of rt . \square

Thus, a suitable runtime bound rt for the RNTS from Example 5.3 yields $\text{rc}_{cp(\mathcal{P})}(n) \in \mathcal{O}(n^4)$, cf. Example 5.8. However, as mentioned above, besides runtime bounds the technique presented in this chapter also relies on size bounds. To formalize our techniques for the inference of size bounds in a modular way, we use the following notion of *sound processors for size*.

Definition 5.10 (Soundness for Size). Let \mathcal{P} be an RNTS over Σ and let proc be a processor such that $\text{proc}(cp(\mathcal{P})) = cp(\mathcal{P}')$. We say that proc is *sound for size* if $t \rightarrow_{\mathcal{P}}^* n$ with $n \in \mathcal{T}(\Sigma_{\mathbb{N}})$ implies $t \rightarrow_{\mathcal{P}'}^* n'$ with $n' \geq n$ for all $t \in \mathcal{T}_{\text{basic}}(\Sigma)$.

Thus, as for the inference of lower and upper bounds we can simplify an RNTS \mathcal{P} via processors which are sound for size until we can extract a size bound directly. Then this size bound is also valid for \mathcal{P} .

5.2 Size Bounds as Runtime Bounds

We first present a transformation for a large class of ITSs that lets us obtain size bounds from any method that can infer runtime bounds. The transformation extends each function symbol from Σ by an additional accumulator argument. Then expressions that are multiplied with the result of a function are collected in the accumulator. Expressions that are added to the result are moved to the cost of the rule.

Theorem 5.11 (Size Bounds for ITSs). *Let \mathcal{P} be an ITS whose rules are of the form*

$$f(\mathbf{x}) \rightarrow u + v \cdot g(\mathbf{t}) \ [\varphi]$$

or

$$f(\mathbf{x}) \rightarrow u \ [\varphi]$$

with $u, v \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ and $f, g \in \Sigma$. Let²

$$\mathcal{P}_{\uparrow} = \{f'(\mathbf{x}, tv) \xrightarrow{u \cdot tv} g'(\mathbf{t}, v \cdot tv) \ [\varphi] \mid f(\mathbf{x}) \rightarrow u + v \cdot g(\mathbf{t}) \ [\varphi] \in \mathcal{P}\} \cup \{f'(\mathbf{x}, tv) \xrightarrow{u \cdot tv} 0 \ [\varphi] \mid f(\mathbf{x}) \rightarrow u \ [\varphi] \in \mathcal{P}\}$$

for a fresh variable $tv \in \mathcal{V}$ and let rt be a runtime bound for \mathcal{P}_{\uparrow} . Then sz with $sz(f)(\mathbf{x}) = rt(f')(\mathbf{x}, 1)$ for all $f \in \Sigma$ is a size bound for \mathcal{P} .

Theorem 5.11 can be generalized to right-hand sides like $f(\mathbf{t}) + 2 \cdot g(\mathbf{s})$ with $f, g \in \Sigma$, cf. Theorem A.1.³ However, it is not applicable if the results of function calls are multiplied on right-hand sides (e.g., $f(\mathbf{t}) \cdot g(\mathbf{s})$) and our technique fails in such cases.

Example 5.12 (Size Bounds as Runtime Bounds – Example 5.8 continued). To get a size bound for $\mathcal{P}^{f_1} = \{\beta_1, \beta_2\}$ (cf. Example 5.3), we construct $\mathcal{P}_{\uparrow}^{f_1}$:

$$\begin{array}{lll} f'_1(x, tv) & \xrightarrow{x \cdot tv} & f'_1(x', tv) \quad [x > 0 \wedge x' = x - 1] \\ f'_1(x, tv) & \xrightarrow{0} & 0 \quad [x = 0] \end{array}$$

Existing ITS tools can compute a runtime bound like $rt(f'_1)(x, tv) = x^2 \cdot tv$ for $\mathcal{P}_{\uparrow}^{f_1}$. Hence, by Theorem 5.11 we obtain the size bound sz with $sz(f_1)(x) = rt(f'_1)(x, 1) = x^2$ for \mathcal{P}^{f_1} .

²Here, $f(\mathbf{v}, tv)$ where \mathbf{v} is a vector of length k stands for $f(v_1, \dots, v_k, tv)$.

³As the generalized Theorem A.1 is less intuitive than Theorem 5.11, which is sufficient to illustrate the underlying idea, Theorem A.1 is only presented in the appendix of this thesis.

5.3 Complexity Bounds for RNTSs

Now we show how complexity tools for ITSs can be used to infer runtime and size bounds for RNTSs in a bottom-up fashion. To formally capture the idea of a bottom-up analysis, we introduce the *call graph relation* for RNTSs.

Definition 5.13 (Call Graph). Let \mathcal{P} be an RNTS over Σ and let

$$\Sigma(\mathcal{P}) = \{\text{root}(\alpha) \mid \alpha \in \mathcal{P}\} \cup \bigcup_{\alpha \in \mathcal{P}} \Sigma(\text{rhs}(\alpha)).$$

Then $\Sigma(\mathcal{P})$ is the node set of the *call graph* of \mathcal{P} and its edges are

$$\{(\text{root}(\alpha), \mathbf{g}) \mid \alpha \in \mathcal{P}, \mathbf{g} \in \Sigma(\text{rhs}(\alpha))\}.$$

We write $\mathbf{f} \sqsupset \mathbf{g}$ if there is a non-empty path from \mathbf{f} to \mathbf{g} in the call graph of \mathcal{P} and $\mathbf{f} \sqsupseteq \mathbf{g}$ if $\mathbf{f} \sqsupset \mathbf{g}$ or $\mathbf{f} = \mathbf{g}$.

So the call graph of \mathcal{P} has an edge from \mathbf{f} to \mathbf{g} if and only if there is a rule where \mathbf{f} occurs on the left-hand and \mathbf{g} occurs on the right-hand side. Our approach is only applicable to RNTSs without *nested recursion*. For such RNTSs, we analyze *induced* sub-RNTSs independently

Definition 5.14 (Nested Recursion, Induced RNTSs). An RNTS \mathcal{P} has *nested recursion* if it has a rule $\ell \rightarrow r \ [\varphi]$ with $\text{root}(r|_{\pi}) \sqsupset \text{root}(\ell)$ and $\text{root}(r|_{\tau}) \sqsupset \text{root}(\ell)$ for positions $\pi < \tau$. The sub-RNTS of \mathcal{P} *induced* by \mathbf{f} is $\mathcal{P}^{\mathbf{f}} = \{\alpha \in \mathcal{P} \mid \mathbf{f} \sqsupseteq \text{root}(\alpha)\}$.

Thus, our approach is not applicable to RNTSs with rules of the form $\mathbf{f}(\dots) \rightarrow \mathbf{f}(\dots \mathbf{f}(\dots) \dots)$. However, such rules rarely occur in practice. The most prominent example for nested recursion is the recursive rule for the Ackermann function:

$$\text{ack}(m, n) \rightarrow \text{ack}(m', \text{ack}(m, n')) \ [m' = m - 1 \wedge n' = n - 1]$$

We have

$$\text{root}(\text{ack}(m', \text{ack}(m, n'))|_{\epsilon}) = \text{root}(\text{ack}(m', \text{ack}(m, n'))|_2) \sqsupset \text{ack}$$

and $\epsilon < 2$. Thus, this rule has nested recursion and hence cannot be handled by the technique presented in this chapter.

The call graph relation \sqsupset induces a partial order on Σ where $\mathbf{f} \geq \mathbf{g}$ if and only if $\mathbf{f} = \mathbf{g}$ or $\mathbf{f} \sqsupset \mathbf{g}$ and $\mathbf{g} \not\sqsupset \mathbf{f}$. In the remainder of this section, when we say that “ \mathbf{f} is smaller than \mathbf{g} ”, “ \mathbf{f} is minimal”, etc., then we refer to this partial order.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

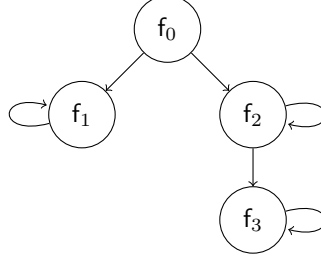


Figure 5.1: Call Graph of Example 5.3

Example 5.15 (Call Graph – Example 5.12 continued). The RNTS from Example 5.3 does not have nested recursion. Its call graph is depicted in Figure 5.1. Its function symbols induce the RNTSSs $\mathcal{P}^{f_0} = \{\beta_1, \dots, \beta_5\}$, $\mathcal{P}^{f_1} = \{\beta_1, \beta_2\}$, $\mathcal{P}^{f_2} = \{\beta_3, \beta_4, \beta_5\}$, and $\mathcal{P}^{f_3} = \{\beta_4, \beta_5\}$.

To compute bounds for an RNTS \mathcal{P} without nested recursion, we start with the trivial bounds $\text{rt}(f)(\mathbf{x}) = \text{sz}(f)(\mathbf{x}) = \omega$ for all $f \in \Sigma$. In each step, we analyze the sub-RNTS \mathcal{P}^f induced by one of the smallest symbols f with $\text{rt}(f)(\mathbf{x}) = \omega$ and refine rt and sz for all function symbols of \mathcal{P}^f . Afterwards we replace the rules from \mathcal{P}^f with trivial rules, i.e., rules whose right-hand sides are arithmetic expressions, and eliminate calls to \mathcal{P}^f via *chaining*. In this way, sub-RNTSSs induced by greater symbols are transformed into ITSs and thus they can be analyzed in the next step.

When computing bounds for a function symbol f , we already know (weakly monotonic) size and runtime bounds for all smaller symbols. To handle right-hand sides like $f(\dots g(\dots) \dots)$ where $f \in \Sigma$ is the function symbol we are analyzing and we have an *inner* call to a smaller symbol g , we replace $g(\dots)$ by a fresh variable tv . The size bound of the replaced call $g(\dots)$ serves as upper bound for the value of tv , but tv can also take smaller values. This replacement is implemented via the processors *Inner Simplification* and *Inner Chaining*. We first introduce *Inner Simplification*, which allows us to replace all rules for a successfully analyzed function symbol with a single trivial rule.

Theorem 5.16 (Inner Simplification). *Let \mathcal{P} be an RNTS with runtime and size bounds rt and sz , and let $f \in \Sigma$ such that $\text{rt}(f)(\mathbf{x}) \neq \omega$. Moreover, let $tv \in \mathcal{V}$ be a fresh variable, let⁴*

$$\begin{aligned} \alpha_f &= f(\mathbf{x}) \xrightarrow{\text{rt}(f)(\mathbf{x})} tv \ [tv \leq \text{sz}(f)(\mathbf{x})], \quad \text{and let} \\ \mathcal{P}' &= \{\alpha \in \mathcal{P} \mid \text{root}(\alpha) \neq f\} \cup \{\alpha_f\}. \end{aligned}$$

Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for upper bounds and size.

5.3. COMPLEXITY BOUNDS FOR RNTSS

Note that the name “Inner Simplification” expresses that the rules resulting from this processor can be used for *Inner Chaining* afterwards, i.e., they can be used to eliminate *inner* calls to previously analyzed functions. The processor *Inner Simplification* itself is applicable to arbitrary previously analyzed function symbols.

Before we prove Theorem 5.16, we show that it suffices to consider finite rewrite sequence in such soundness proofs.

Lemma 5.17. *Let $\mathcal{P}, \mathcal{P}'$ be RNTSSs over Σ . If $t_0 \xrightarrow{\mathcal{P}}^* t_m$ implies $t_0 \xrightarrow{\mathcal{P}'}^* t'_m$ with $\mathcal{K}' \geq \mathcal{K}$ for all $t_0 \in \mathcal{T}_{\text{basic}}(\Sigma)$, then $\text{dh}_{\rightarrow_{\mathcal{P}'}}(t_0) \geq \text{dh}_{\rightarrow_{\mathcal{P}}}(t_0)$.*

Proof. We have

$$\begin{aligned}
 & \text{dh}_{\rightarrow_{\mathcal{P}}}(t_0) \\
 &= \sup\{\mathcal{K} \mid t_m \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}}), t_0 \xrightarrow{\mathcal{P}}^* t_m\} \quad \text{by Definition 2.12} \\
 &\leq \sup\{\mathcal{K}' \mid t'_m \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}}), t_0 \xrightarrow{\mathcal{P}'}^* t'_m\} \quad \text{as } t_0 \xrightarrow{\mathcal{P}}^* t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \\
 &\quad \text{with } \mathcal{K}' \geq \mathcal{K} \\
 &= \text{dh}_{\rightarrow_{\mathcal{P}'}}(t_0) \quad \text{by Definition 2.12.}
 \end{aligned}$$

□

Proof of Theorem 5.16. Let $\overline{\mathcal{P}} = \{\alpha \in \mathcal{P} \mid \text{root}(\alpha) = \mathbf{f}\}$. We prove soundness for upper bounds and size independently.

Claim 1. The processor is sound for upper bounds.

We prove

$$t_0 \xrightarrow{\mathcal{P}}^m t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \text{ where } \mathcal{K}' \geq \mathcal{K}.$$

for all terms t_0 . Then the claim follows with Lemma 5.17. We use induction on m . If $m = 0$, then $t_0 = t_m$ and $\mathcal{K} = 0$ and hence the claim is trivial. Assume $m > 0$. Let α and π be the rule and the position which are used for the first step in the rewrite sequence, i.e., we have $t_0 \xrightarrow{\alpha} t_1 \xrightarrow{\mathcal{P}}^{m-1} t_m$. If $\alpha \notin \overline{\mathcal{P}}$, then we have $t_0 \xrightarrow{\mathcal{P}'}^* t_1$ and the claim follows from the induction hypothesis. Assume $\alpha \in \overline{\mathcal{P}}$.

Case 1. $t_0|_{\pi}$ is reduced to an arithmetic expression

More precisely, in this case there is an $i \in \{1, \dots, m\}$ such that $t_i|_{\pi} \in \mathcal{T}(\Sigma_{\mathbb{N}})$. Then we may assume that the first $m_{\pi} > 0$ steps of the rewrite sequence $t_0 \xrightarrow{\mathcal{P}}^m t_m$ normalize $t_0|_{\pi}$ without loss of generality since RNTSSs are evaluated with an innermost strategy. Thus, we have

$$t_0 = t_0[\mathbf{f}(\mathbf{n})]_{\pi} \xrightarrow{\mathcal{P}}^{m_{\pi}} t_0[n_{\pi}]_{\pi} \xrightarrow{\mathcal{P}}^{m-m_{\pi}} t_m$$

for some $\mathcal{K}_{\pi}, n_{\pi} \in \mathcal{T}(\Sigma_{\mathbb{N}})$ and some $\mathbf{n} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}})$. Since rt and sz are runtime and size bounds for \mathcal{P} , we get $\mathcal{K}_{\pi} \leq \text{rt}(\mathbf{f})(\mathbf{n})$ and $n_{\pi} \leq \text{sz}(\mathbf{f})(\mathbf{n})$. Thus, we

⁴If $\text{sz}(\mathbf{f})(\mathbf{x}) = \omega$, then $tv \leq \text{sz}(\mathbf{f})(\mathbf{x})$ is a tautology, i.e., then we have $\text{guard}(\alpha_{\mathbf{f}}) = \text{true}$.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

get

$$t_0[f(\mathbf{n})]_\pi \xrightarrow{\text{rt}(\mathbf{f})(\mathbf{n})}_{\alpha_f} t_0[n_\pi]_\pi = t_1$$

by instantiating tv with n_π . Then the claim follows from the induction hypothesis.

Case 2. $t_0|_\pi$ is not reduced to an arithmetic expression

More precisely, in this case we have $\pi \in \text{pos}(t_m)$ and $t_m|_\pi \notin \mathcal{T}(\Sigma_{\mathbb{N}})$. Then we may again assume that the first $m_\pi > 0$ steps of the rewrite sequence $t_0 \xrightarrow{\mathcal{R}_P^m} t_m$ reduce $t_0|_\pi$ to $t_m|_\pi$ without loss of generality since RNTSSs are evaluated with an innermost strategy, i.e., we have

$$t_0 = t_0[f(\mathbf{n})]_\pi \xrightarrow{\mathcal{R}_P^{m_\pi}} t_0[t_m|_\pi]_\pi \xrightarrow{\mathcal{R}_P^{m-m_\pi}} t_m \quad (5.1)$$

for some $\mathcal{R}_\pi \in \mathcal{T}(\Sigma_{\mathbb{N}})$ and some $\mathbf{n} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}})$ where all rewrite steps $t_0[t_m|_\pi]_\pi \xrightarrow{\mathcal{R}_P^*} t_m$ take place at positions which are independent from π . Hence, we have

$$t_0[q]_\pi \xrightarrow{\mathcal{R}_P^{m-m_\pi}} t_m[q]_\pi \text{ for every term } q$$

by applying the same rules in the same order at the same positions with the same substitutions as in (5.1). Since rt is a runtime bound for \mathcal{P} , we get $\mathcal{R}_\pi \leq \text{rt}(\mathbf{f})(\mathbf{n})$. Thus, we get

$$t_0[f(\mathbf{n})]_\pi \xrightarrow{\text{rt}(\mathbf{f})(\mathbf{n})}_{\alpha_f} t_0[0]_\pi \xrightarrow{\mathcal{R}_P^{m-m_\pi}} t_m[0]_\pi$$

by instantiating tv with 0. Then the claim follows from the induction hypothesis.

Claim 2. The processor is sound for size.

To prove the claim, we prove

$$t_0 \xrightarrow{\mathcal{R}_P^m} t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \xrightarrow{\mathcal{R}_P^*} t_m.$$

As in the proof of *Claim 1*, we use induction on m . We only have to consider *Case 1*, since the rewrite sequence does not end with an arithmetic expression in *Case 2*. Note that we indeed have $t'_m = t_m$ in the proof of *Case 1*. Thus, the proof of *Claim 2* is analogous to the proof of *Claim 1, Case 1*. \square

5.3. COMPLEXITY BOUNDS FOR RNTSs

Example 5.18 (*Inner Simplification* – Example 5.15 continued). Reconsider the RNTS \mathcal{P} from Example 5.3 and assume that we already inferred the size bound $\text{sz}(\mathbf{f}_1)(x) = x^2$ (cf. Example 5.12) and the (trivial) size bound $\text{sz}(\mathbf{f}_3)(u) = 0$ for its minimal symbols. Since $\mathcal{P}^{\mathbf{f}_1}$ and $\mathcal{P}^{\mathbf{f}_3}$ are ITSs, we can directly compute runtime bounds like $\text{rt}(\mathbf{f}_1)(x) = x + 1$ and $\text{rt}(\mathbf{f}_3)(u) = u + 1$ using existing tools. Now we can simplify \mathcal{P} by applying *Inner Simplification* to \mathbf{f}_1 and \mathbf{f}_3 in order to eliminate the inner call to \mathbf{f}_3 on the right-hand side of β_3 via *Inner Chaining* later on (cf. Example 5.22). In this way, we obtain the following RNTS:

$$\begin{array}{llll} \beta_0 : & \mathbf{f}_0(x) & \xrightarrow{1} & \mathbf{f}_2(\mathbf{f}_1(x), u) \\ \beta_{\mathbf{f}_1} : & \mathbf{f}_1(x) & \xrightarrow{x+1} & tv \quad [tv \leq x^2] \\ \beta_3 : & \mathbf{f}_2(z, u) & \xrightarrow{1} & \mathbf{f}_2(z', \mathbf{f}_3(z')) \quad [z > 0 \wedge z' = z - 1] \\ \beta_{\mathbf{f}_3} : & \mathbf{f}_3(u) & \xrightarrow{u+1} & tv \quad [tv \leq 0] \end{array}$$

Example 5.19 (*Inner Simplification* – Example 5.4 continued). Assume that we already computed the runtime and size bounds $\text{rt}(\mathbf{plus})(x, y) = x + 1$ and $\text{sz}(\mathbf{plus})(x, y) = x + y$. Then applying *Inner Simplification* to \mathbf{plus} results in the following RNTS:

$$\begin{array}{llll} \gamma_0 : & \mathbf{times}(x, y) & \xrightarrow{1} & \mathbf{plus}(\mathbf{times}(x', y), y) \quad [x > 0 \wedge x' = x - 1] \\ \gamma_1 : & \mathbf{times}(x, y) & \xrightarrow{1} & 0 \quad [x = 0] \\ \gamma_{\mathbf{plus}} : & \mathbf{plus}(x, y) & \xrightarrow{x+1} & tv \quad [tv \leq x + y] \end{array}$$

However, later on we will see that the resulting trivial rule $\gamma_{\mathbf{plus}}$ is not suitable to eliminate the *outer* call to \mathbf{plus} in γ_0 (cf. Example 5.23).

After replacing the rules for successfully analyzed symbols by trivial rules, inner calls to these symbols can be eliminated via *chaining*.

Definition 5.20 (Chaining). Let \mathcal{P} be an RNTS with size bound sz . We lift sz to terms by defining $\text{sz}(x) = x$ if $x \in \mathcal{V}$, $\text{sz}(\mathbf{f}(\mathbf{t})) = \text{sz}(\mathbf{f})(\text{sz}(\mathbf{t}))$ if $\mathbf{f} \in \Sigma$, and $\text{sz}(\mathbf{f}(\mathbf{t})) = \mathbf{f}(\text{sz}(\mathbf{t}))$ if $\mathbf{f} \in \Sigma_{\mathbb{N}}$. Let \mathcal{P} contain the following rules:

$$\begin{array}{lll} \alpha_1 : \mathbf{f}_1(\mathbf{x}) & \xrightarrow{c_1} & C[\mathbf{f}_2(\mathbf{t})] \quad [\varphi_1] \\ \alpha_2 : \mathbf{f}_2(\mathbf{y}) & \xrightarrow{c_2} & t \quad [\varphi_2] \end{array}$$

W.l.o.g., assume $\mathcal{V}(\alpha_1) \cap \mathcal{V}(\alpha_2) = \emptyset$ (otherwise, one can rename the variables in one rule accordingly). Then *chaining* α_1 and α_2 yields

$$\alpha_{1.2} : \mathbf{f}_1(\mathbf{x}) \xrightarrow{c_1 + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}} C[t\{\mathbf{y}/\mathbf{t}\}] \quad [\varphi_1 \wedge \varphi_2\{\mathbf{y}/\mathbf{t}\}].$$

Note that $\alpha_{1.2}$ is only a valid RNTS rule if $\mathbf{t}|_i \notin \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$ implies $\mathbf{y}|_i \notin \mathcal{V}(\varphi_2)$ since otherwise $\text{guard}(\alpha_{1.2})$ contains function symbols. However, our processors which are based on chaining ensure that the resulting rule is valid. For example,

CHAPTER 5. UPPER BOUNDS FOR RNTSS

Theorem 5.21 can be used to eliminate “innermost” calls to previously analyzed function symbols. So in this case, we know that all arguments \mathbf{t} of the eliminated call are arithmetic expressions.

To eliminate such inner calls, we chain a rule α_1 of the form

$$f_1(\dots) \rightarrow g(\dots f_2(\dots) \dots)$$

with a trivial rule α_2 for f_2 , i.e., a rule of the form $f_2(\mathbf{y}) \rightarrow t$ where t is an arithmetic expression. Then α_1 is replaced by the new (chained) rule. This is only sound if we know that the subterm $f_2(\dots)$ of α_1 ’s right-hand side is eventually reduced with α_2 in cost-maximal rewrite sequences. Hence, we require that α_2 is the only f_2 -rule and that α_2 ’s guard is satisfiable for all valuations of \mathbf{y} .

Theorem 5.21 (Inner Chaining). *Let \mathcal{P} be an RNTS with rules α_1, α_2 as in Definition 5.20 where $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$, α_2 is the only rule with root f_2 , $t \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$, and $\varphi_2\sigma$ is satisfiable for all substitutions $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$ with $\text{dom}(\sigma) = \mathbf{y}$. Furthermore, let $\mathcal{P}' = \mathcal{P} \cup \{\alpha_{1.2}\} \setminus \{\alpha_1\}$. Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for upper bounds and size.*

Proof. First of all, note that $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$ ensures that $\alpha_{1.2}$ is a valid RNTS rule. We prove soundness for upper bounds and size separately.

Claim 1. The processor is sound for upper bounds.

It suffices to prove

$$t_0 \xrightarrow{\mathcal{P}}^* t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \text{ with } \mathcal{K}' \geq \mathcal{K}.$$

for all terms t_0 . Then the claim follows with Lemma 5.17. Instead, we prove the more general statement

$$t_0 \xrightarrow{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \text{ with } \mathcal{K}' \geq \mathcal{K}.$$

We use induction on the number of α_1 -steps in $t_0 \xrightarrow{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m$. If there is no such step, then the claim is trivial.

Otherwise, consider the last α_1 -step, i.e., we have

$$t_0 \xrightarrow{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_i = t_i[f_1(\mathbf{x})\sigma_1]_{\pi} \xrightarrow{c_1\sigma_1}_{\alpha_1} t_i[C[f_2(\mathbf{t})]\sigma_1]_{\pi} \xrightarrow{\mathcal{P}}^* t_m \quad (5.2)$$

with $\mathcal{K} = \mathcal{K}_1 + c_1\sigma_1 + \mathcal{K}_2$ for some position π and some integer substitution σ_1 . It remains to show

$$t_i[f_1(\mathbf{x})\sigma_1]_{\pi} \xrightarrow{\mathcal{K}}_{\mathcal{P}'}^* t'_m \text{ with } \overline{\mathcal{K}} \geq c_1\sigma_1 + \mathcal{K}_2. \quad (5.3)$$

Then the claim follows from the induction hypothesis. Let π' be the position of \square in C .

5.3. COMPLEXITY BOUNDS FOR RNTSS

Note that α_2 can be used to reduce $f_2(\mathbf{t})\sigma_1$ to an arithmetic expression. The reason is that we have $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$ and $\text{guard}(\varphi_2)$ is satisfiable for every instantiation of \mathbf{y} . Moreover, since α_2 is the only rule with root f_2 , this is the only way how $f_2(\mathbf{t})\sigma_1$ can be reduced. W.l.o.g., we can assume that $f_2(\mathbf{t})\sigma_1$ is indeed reduced to an arithmetic expression with α_2 in (5.2). Otherwise, we have $t_m|_{\pi.\pi'} = f_2(\mathbf{t})\sigma_1$, i.e., then there is a rewrite sequence

$$t_0 \xrightarrow{\mathcal{K}_{\mathcal{P} \cup \{\alpha_{1.2}\}}^*} t_m \xrightarrow{\alpha_2^{c_2\sigma_2}} t_m[t\sigma_2]_{\pi.\pi'}$$

for some integer substitution σ_2 with $\sigma_2|_{\mathbf{y}} = \{\mathbf{y}/\mathbf{t}\sigma_1\}$ which we can consider instead of the rewrite sequence $t_0 \xrightarrow{\mathcal{K}_{\mathcal{P} \cup \{\alpha_{1.2}\}}^*} t_m$.

Thus, the rewrite sequence (5.2) can be reordered such that the α_1 -step is directly succeeded by an α_2 -step at position $\pi.\pi'$ without affecting its cost or result. The reason is that we have $\mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$ and RNTSSs are evaluated with an innermost strategy. Thus, w.l.o.g. we assume

$$\begin{aligned} t_i[f_1(\mathbf{x})\sigma_1]_{\pi} & \xrightarrow{\alpha_1^{c_1\sigma_1}} t_i[C[f_2(\mathbf{t})]\sigma_1]_{\pi} \\ & = t_i[C[f_2(\mathbf{y})\sigma_2]\sigma_1]_{\pi} \\ & \xrightarrow{\alpha_2^{c_2\sigma_2}} t_i[C[t\sigma_2]\sigma_1]_{\pi} \\ & \xrightarrow{\mathcal{K}_{\mathcal{P}}'^*} t_m \end{aligned}$$

with $\mathcal{K}_2 = c_2\sigma_2 + \mathcal{K}_2'$. To finish the proof of (5.3), we prove

$$f_1(\mathbf{x})\sigma_1 \xrightarrow{\alpha_{1.2}^{c_1\sigma_1+c_2\sigma_2}} C[t\sigma_2]\sigma_1. \quad (5.4)$$

Let $\sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)}$. Then we have $\sigma_1|_{\mathcal{V}(\alpha_1)} = \sigma|_{\mathcal{V}(\alpha_1)}$ and thus $\sigma_1 \models \varphi_1$ implies $\sigma \models \varphi_1$. Moreover, we have:

$$\begin{aligned} & \sigma_2 \models \varphi_2 \\ \iff & \sigma_2|_{\mathcal{V}(\alpha_2)} \models \varphi_2 && \text{as } \mathcal{V}(\varphi_2) \subseteq \mathcal{V}(\alpha_2) \\ \iff & \{\mathbf{y}/\mathbf{t}\sigma_1\} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \models \varphi_2 && \text{as } \mathbf{t}\sigma_1 = \mathbf{y}\sigma_2 \\ \iff & \{\mathbf{y}/\mathbf{t}\sigma_1|_{\mathcal{V}(\alpha_1)}\} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \models \varphi_2 && \text{as } \mathcal{V}(\mathbf{t}) \subseteq \mathcal{V}(\alpha_1) \\ \iff & \{\mathbf{y}/\mathbf{t}\} \diamond \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \models \varphi_2 && \text{as } \mathcal{V}(\alpha_1) \cap \mathcal{V}(\alpha_2) = \emptyset \\ \iff & \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \models \varphi_2\{\mathbf{y}/\mathbf{t}\} \\ \iff & \sigma \models \varphi_2\{\mathbf{y}/\mathbf{t}\} && \text{as } \sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \end{aligned}$$

Hence, $\sigma \models \text{guard}(\alpha_{1.2})$. (5.5)

Moreover, we have

$$f_1(\mathbf{x})\sigma_1 = f_1(\mathbf{x})\sigma = \text{lhs}(\alpha_{1.2})\sigma \quad \text{and} \quad (5.6)$$

CHAPTER 5. UPPER BOUNDS FOR RNTSS

$$\begin{aligned}
C[t\sigma_2]\sigma_1 &= C[t\{\mathbf{y}/\mathbf{t}\sigma_1\}\sigma_2]\sigma_1 && \text{as } \mathbf{t}\sigma_1 = \mathbf{y}\sigma_2 \\
&= C[t\{\mathbf{y}/\mathbf{t}\sigma\}]\sigma && \text{as } \sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \\
&&& \text{and } \mathcal{V}(\alpha_1) \cap \mathcal{V}(\alpha_2) = \emptyset \\
&= C[t\{\mathbf{y}/\mathbf{t}\}]\sigma \\
&= \text{rhs}(\alpha_{1.2})\sigma.
\end{aligned} \tag{5.7}$$

Finally, we have:

$$\begin{aligned}
&c_1\sigma_1 && + && c_2\sigma_2 \\
= &c_1\sigma_1 && + && c_2\{\mathbf{y}/\mathbf{t}\sigma_1\}\sigma_2 && \text{as } \mathbf{t}\sigma_1 = \mathbf{y}\sigma_2 \\
= &c_1\sigma && + && c_2\{\mathbf{y}/\mathbf{t}\}\sigma && \text{as } \sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \\
&&& && && \text{and } \mathcal{V}(\alpha_1) \cap \mathcal{V}(\alpha_2) = \emptyset \\
= &c_1\sigma && + && c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma && \text{as } \mathbf{t} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}) \\
= &\text{cost}(\alpha_{1.2})\sigma
\end{aligned} \tag{5.8}$$

Together, (5.5) – (5.8) imply (5.4).

Claim 2. The processor is sound for size.

It suffices to prove

$$t_0 \rightarrow_{\mathcal{P}}^* t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \rightarrow_{\mathcal{P}'}^* t_m.$$

Instead, we again prove the more general statement

$$t_0 \rightarrow_{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \rightarrow_{\mathcal{P}'}^* t_m.$$

As in the proof of *Claim 1*, we use induction on the number of α_1 -steps in the sequence $t_0 \rightarrow_{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m$. Note that we indeed have $t'_m = t_m$ in the proof of *Claim 1*, i.e., the proof of *Claim 2* is analogous to the proof of *Claim 1*. \square

Note that the rules resulting from Theorem 5.16 satisfy the prerequisites of Theorem 5.21, i.e., these rules can be used for inner chaining.

Example 5.22 (Inner Chaining – Example 5.18 finished). Theorem 5.21 is applicable to β_0 and β_{f_1} , since the only argument x of the call $f_1(x)$ on the right-hand side of β_0 is an arithmetic expression and $\text{guard}(\beta_{f_1})$ is satisfiable for every instantiation of x (by setting tv to 0). Moreover, β_{f_1} is the only rule with root f_1 . Similarly, Theorem 5.21 is applicable to β_3 and β_{f_3} . Thus, by applying Theorem 5.21 twice we obtain the following RNTS:

$$\begin{array}{llll}
\beta_{0.f_1} : & f_0(x) & \xrightarrow{x+2} & f_2(tv, u) \quad [tv \leq x^2] \\
\beta_{f_1} : & f_1(x) & \xrightarrow{x+1} & tv \quad [tv \leq x^2] \\
\beta_{3.f_3} : & f_2(z, u) & \xrightarrow{z'+2} & f_2(z', tv) \quad [z > 0 \wedge z' = z - 1 \wedge tv \leq 0] \\
\beta_{f_3} : & f_3(u) & \xrightarrow{u+1} & tv \quad [tv \leq 0]
\end{array}$$

5.3. COMPLEXITY BOUNDS FOR RNTSs

Since this RNTS is also an ITS, standard complexity analysis tools can now compute runtime bounds like

$$\text{rt}(f_2)(z, u) = z^2 + z \text{ and } \text{rt}(f_0)(x) = x^4 + x^2 + x + 2.$$

This proves that the runtime complexity of Example 5.3 is in $\mathcal{O}(n^4)$.

However, Theorem 5.21 cannot be used to eliminate *outer* calls to previously analyzed symbols.

Example 5.23 (Inner Chaining – Example 5.19 continued). Even though the only remaining **plus** rule is trivial, Theorem 5.21 cannot be used to eliminate the call to **plus** on the right-hand side **plus**(**times**(x', y), y). The reason is that **times**(x', y) is not an arithmetic expression and thus the prerequisites of Theorem 5.21 are not satisfied, as Theorem 5.21 can only be applied if all arguments of the call which needs to be eliminated are arithmetic expressions.

To handle such calls, we introduce the processor *Size Simplification* (cf. Theorem 5.27). Like *Inner Simplification*, it transforms non-trivial into trivial rules. These trivial rules can then be used for *Outer Chaining* (cf. Theorem 5.29), i.e., they are suitable to eliminate outer calls to previously analyzed symbols like **plus**(**times**(x', y), y). In contrast to Theorem 5.16 where the right-hand side of the resulting trivial rule for **f** is a variable whose value is bounded by $\text{sz}(\mathbf{f})$, the right-hand side of the rules resulting from Theorem 5.27 is $\text{sz}(\mathbf{f})$ itself. Consequently, a condition like “ $tv \leq \text{sz}(\mathbf{f})(\mathbf{x})$ ” from Theorem 5.16 is not required, i.e., the guard of the rules resulting from Theorem 5.27 is “true”. Hence, if such rules are used for chaining, then the resulting chained rule is a valid RNTS rule even if the arguments of the eliminated call contain function symbols.

However, replacing all **f**-rules with such a trivial rule is only sound if all function symbols **g** that may occur above **f** behave monotonically w.r.t. their costs and results. Otherwise, replacing **f** with its *upper* bound $\text{sz}(\mathbf{f})$ might result in *smaller* costs or results for terms of the form $\mathbf{g}(\dots \mathbf{f}(\dots) \dots)$. To formalize this precondition, the following definition captures which function symbols may occur above other function symbols.

Definition 5.24 (Σ°). Let \mathcal{P} be an RNTS over Σ . We define

$$\Sigma^\circ = \{\mathbf{f} \in \Sigma \mid \ell \rightarrow r \text{ } [\varphi] \in \mathcal{P}, \pi, \tau \in \mathbb{N}^*, \pi < \tau, \text{root}(r|_\pi) = \mathbf{f}, \text{root}(r|_\tau) \in \Sigma\}.$$

We call a term t *properly nested* if $\text{root}(t|_\pi), \text{root}(t|_\tau) \in \Sigma$ with $\pi < \tau$ implies $\text{root}(t|_\pi) \in \Sigma^\circ$.

The following lemma states that properly nested terms are closed under rewriting. So in particular, Σ° contains all function symbols that may occur above other function symbols in rewrite sequences starting with basic terms (since each basic term is properly nested).

CHAPTER 5. UPPER BOUNDS FOR RNTSS

Lemma 5.25 (Properly Nested Terms are Closed Under Rewriting). *Let \mathcal{P} be an RNTS. If t is a properly nested term with $t \rightarrow_{\mathcal{P}} q$, then q is properly nested.*

Proof. Let μ and τ be positions with $\mu < \tau$ such that $\text{root}(q|_{\mu}), \text{root}(q|_{\tau}) \in \Sigma$ (if there are no such positions, then the claim is trivial). To prove the lemma, we prove $\text{root}(q|_{\mu}) \in \Sigma^o$. Let π and α be the position and the rule of the rewrite step $t \rightarrow_{\mathcal{P}} q$. If π and τ are independent or $\tau < \pi$, then we have $\text{root}(t|_{\mu}) = \text{root}(q|_{\mu})$ and $\text{root}(t|_{\tau}) = \text{root}(q|_{\tau})$. Thus, we have $\text{root}(q|_{\mu}) \in \Sigma^o$, since t is properly nested. Assume $\pi \leq \tau$. If $\pi \leq \mu$, then $\text{root}(q|_{\tau})$ occurs below $\text{root}(q|_{\mu})$ in $\text{rhs}(\alpha)$ and thus we have $\text{root}(q|_{\mu}) \in \Sigma^o$ by the definition of Σ^o . If $\mu < \pi$, then there is a non-empty position μ' such that $\pi = \mu.\mu'$ and we have $t|_{\mu} = q|_{\mu}[s]_{\mu'}$ where $\text{root}(s) \in \Sigma$. Thus, we have $\text{root}(q|_{\mu}) \in \Sigma^o$, since t is properly nested. \square

Moreover, Definition 5.24 gives rise to the following straightforward corollary

Corollary 5.26. *Let \mathcal{P} be an RNTS over Σ . \mathcal{P} is an ITS if and only if $\Sigma^o = \emptyset$.*

To transform an induced sub-RNTS \mathcal{P}^f into an ITS, we eliminate all calls to Σ^o . To this end, *Size Simplification* simultaneously replaces all rules for Σ^o -symbols with trivial rules which behave monotonically w.r.t. cost and size. In this way, it is ensured that all function symbols that may occur above other function symbols behave monotonically. Thus, *Size Simplification* is sound, even though the resulting trivial rules may yield larger results than the original rules. Then all remaining calls to Σ^o (i.e., those calls that could not be eliminated via *Inner Simplification* and *Inner Chaining*) can be eliminated via *Outer Chaining* afterwards, resulting in an ITS.

Theorem 5.27 (Size Simplification). *Let \mathcal{P} be an RNTS over Σ with runtime and size bounds rt and sz where $\text{rt}(f) \neq \omega$ for all $f \in \Sigma^o$. For each $f \in \Sigma^o$ let*

$$\alpha_f = f(\mathbf{x}) \xrightarrow{\text{rt}(f)(\mathbf{x})} r$$

where $r = tv \in \mathcal{V}$ is a fresh variable if $\text{sz}(f)(\mathbf{x}) = \omega$ and $r = \text{sz}(f)(\mathbf{x})$ otherwise. Furthermore, let

$$\mathcal{P}' = \{\alpha \in \mathcal{P} \mid \text{root}(\alpha) \notin \Sigma^o\} \cup \{\alpha_f \mid f \in \Sigma^o\}.$$

Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for upper bounds and size.

Proof. We prove both claims individually.

5.3. COMPLEXITY BOUNDS FOR RNTSS

Claim 1. The processor is sound for upper bounds.

We prove

$$t_0 \xrightarrow{\bar{\ell}_1}_{\mathcal{P}} \dots \xrightarrow{\bar{\ell}_m}_{\mathcal{P}} t_m \text{ implies } t_0 \xrightarrow{\bar{\ell}'}_{\mathcal{P}'}^* t'_m \text{ with } \bar{\ell}' \geq \sum_{i=1}^m \bar{\ell}_i \quad (5.9)$$

for all properly nested terms t_0 . Then the claim follows with Lemma 5.17, since each basic term is properly nested. Let $\bar{\ell} = \sum_{i=1}^m \bar{\ell}_i$. By Lemma 5.25,

$$t_1, \dots, t_m \text{ are properly nested.} \quad (5.10)$$

W.l.o.g., we assume that Σ^o -rules (i.e., the rules $\{\alpha \in \mathcal{P} \mid \text{root}(\alpha) \in \Sigma^o\}$) are applied with a lower priority than $\Sigma \setminus \Sigma^o$ -rules in the rewrite sequence $t_0 \xrightarrow{\bar{\ell}'}_{\mathcal{P}'}^* t'_m$. Otherwise, the rewrite steps can be reordered accordingly due to the innermost evaluation strategy of RNTSSs.

Case 1. $t_m \in \mathcal{T}(\Sigma_{\mathbb{N}})$

Then there is a $k \in \{0, \dots, m\}$ such that $t_0 \xrightarrow{\bar{\ell}'}_{\mathcal{P}'}^* t_k$ and $\Sigma(t_k) \subseteq \Sigma^o$ due to (5.10) and our assumption w.r.t. the evaluation strategy above. We show that

$$t_k \xrightarrow{\bar{\ell}}_{\mathcal{P}}^* t_m \text{ implies } t_k \xrightarrow{\bar{\ell}}_{\mathcal{P}'}^* t'_m \text{ with } \bar{\ell} \geq \bar{\ell} \text{ and } t'_m \geq t_m.$$

We use induction on $t_k = f(\mathbf{t})$. Let $\mathbf{n} \subseteq \mathcal{T}(\Sigma_{\mathbb{N}})$ be the normal forms of \mathbf{t} in the rewrite sequence $t_k \rightarrow_{\mathcal{P}}^* t_m$, i.e., we have

$$f(\mathbf{t}) \xrightarrow{\bar{\ell}_1}_{\mathcal{P}}^* f(\mathbf{n}) \xrightarrow{\bar{\ell}_2}_{\mathcal{P}}^* t_m \text{ with } \bar{\ell} = \bar{\ell}_1 + \bar{\ell}_2.$$

By the induction hypothesis, we have $f(\mathbf{t}) \xrightarrow{\bar{\ell}_1}_{\mathcal{P}'}^* f(\mathbf{n}')$ where $\mathbf{n}' \geq \mathbf{n}$ and $\bar{\ell}_1 \geq \bar{\ell}_1$.

If $f \in \Sigma_{\mathbb{N}}$, then we have $\bar{\ell}_2 = 0$ and $t_m = f(\mathbf{n})$. Thus, we have $\bar{\ell} = \bar{\ell}_1 \geq \bar{\ell}_1 = \bar{\ell}$. Furthermore, we have $t'_m = f(\mathbf{n}') \geq t_m$ by monotonicity of f .

Assume $f \in \Sigma^o$. If $\text{sz}(f) = \omega$, then we have $f(\mathbf{n}') \xrightarrow{\text{rt}(f)(\mathbf{n}')}_{\mathcal{P}'} t_m$ by instantiating tv with t_m . Since rt is a runtime bound, we have $\text{rt}(f)(\mathbf{n}) \geq \bar{\ell}_2$. Thus, $\text{rt}(f)(\mathbf{n}') \geq \bar{\ell}_2$ follows by monotonicity of rt . Hence, we have $\bar{\ell} = \bar{\ell}_1 + \text{rt}(f)(\mathbf{n}') \geq \bar{\ell}_1 + \bar{\ell}_2 = \bar{\ell}$.

If $\text{sz}(f) \neq \omega$, then we have

$$f(\mathbf{n}') \xrightarrow{\text{rt}(f)(\mathbf{n}')}_{\mathcal{P}'} \text{sz}(f)(\mathbf{n}') = t'_m.$$

Since sz is a size bound, we have $\text{sz}(f)(\mathbf{n}) \geq t_m$. By monotonicity of sz , we get

$$t'_m = \text{sz}(f)(\mathbf{n}') \geq \text{sz}(f)(\mathbf{n}) \geq t_m.$$

Moreover, we again have $\text{rt}(f)(\mathbf{n}') \geq \bar{\ell}_2$ and thus $\bar{\ell} = \bar{\ell}_1 + \text{rt}(f)(\mathbf{n}') \geq \bar{\ell}_1 + \bar{\ell}_2 = \bar{\ell}$ as above since rt is a monotonic runtime bound.

Case 2. $t_m \notin \mathcal{T}(\Sigma_{\mathbb{N}})$

Then there is a $k \in \{0, \dots, m\}$ such that $t_0 \xrightarrow{\bar{\ell}'}_{\mathcal{P}'}^* t_k$ and $\text{root}(t_k|_{\pi}) \in \Sigma \setminus \Sigma^o$

CHAPTER 5. UPPER BOUNDS FOR RNTSS

implies $t_k|_\pi = t_m|_\pi$ due to (5.10) and our assumption w.r.t. the evaluation strategy above. We prove that

$$t_k \xrightarrow{\bar{\kappa}}^*_{\mathcal{P}} t_m \text{ implies } t_k \xrightarrow{\tilde{\kappa}}^*_{\mathcal{P}'} t'_m \text{ with } \tilde{\kappa} \geq \bar{\kappa} \text{ for some term } t'_m.$$

We use induction on $t_k = f(\mathbf{t})$. If $f \in \Sigma \setminus \Sigma^o$, then $t_k = t_m$ and thus the claim is trivial. Assume $f \in \Sigma^o \cup \Sigma_{\mathbb{N}}$ and let \mathbf{t}' be the normal forms of \mathbf{t} in the rewrite sequence $t_k \rightarrow^*_{\mathcal{P}} t_m$, i.e., we have

$$f(\mathbf{t}) \xrightarrow{\bar{\kappa}_1}^*_{\mathcal{P}} f(\mathbf{t}') \xrightarrow{\bar{\kappa}_2}^*_{\mathcal{P}} t_m \text{ with } \bar{\kappa} = \bar{\kappa}_1 + \bar{\kappa}_2.$$

If $\mathbf{t}' \not\subseteq \mathcal{T}(\Sigma_{\mathbb{N}})$, then we have $\bar{\kappa} = \bar{\kappa}_1$ and $f(\mathbf{t}') = t_m$. Thus, the claim follows by the induction hypothesis, which implies

$$f(\mathbf{t}) \xrightarrow{\tilde{\kappa}_1}^*_{\mathcal{P}'} f(\mathbf{t}'') \text{ with } \tilde{\kappa}_1 \geq \bar{\kappa}_1 \text{ for some terms } \mathbf{t}''.$$

If $\mathbf{t}' \subseteq \mathcal{T}(\Sigma_{\mathbb{N}})$, then we get

$$f(\mathbf{t}) \xrightarrow{\tilde{\kappa}_1}^*_{\mathcal{P}'} f(\mathbf{t}'') \text{ where } \tilde{\kappa}_1 \geq \bar{\kappa}_1 \text{ and } \mathbf{t}'' \geq \mathbf{t}'$$

due to *Case 1*. Furthermore, we get $f(\mathbf{t}'') \xrightarrow{\text{rt}(f)(\mathbf{t}'')}_{\alpha_f} t'_m$ for some term t'_m by definition of α_f . Since rt is a runtime bound, we have $\text{rt}(f)(\mathbf{t}') \geq \bar{\kappa}_2$. By monotonicity of rt , we get $\text{rt}(f)(\mathbf{t}'') \geq \bar{\kappa}_2$. Thus, we have

$$\tilde{\kappa} = \tilde{\kappa}_1 + \text{rt}(f)(\mathbf{t}'') \geq \bar{\kappa}_1 + \bar{\kappa}_2 = \bar{\kappa}.$$

Claim 2. The processor is sound for size.

It suffices to prove

$$t_0 \rightarrow^*_{\mathcal{P}} t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \rightarrow^*_{\mathcal{P}'} t'_m \text{ with } t'_m \geq t_m.$$

Note that we only need to consider *Case 1*, as we have $t_m \notin \mathcal{T}(\Sigma_{\mathbb{N}})$ in *Case 2*. Indeed, we have $t'_m \geq t_m$ in the proof of *Case 1*. Thus, the proof is analogous to *Claim 1*, *Case 1*. \square

Example 5.28 (Size Simplification – Example 5.23 continued). By applying Theorem 5.27 to the RNTS from Example 5.19 where we have $\Sigma^o = \{\text{plus}\}$ we obtain:

$$\begin{array}{llll} \gamma_0 : & \text{times}(x, y) & \xrightarrow{1} & \text{plus}(\text{times}(x', y), y) & [x > 0 \wedge x' = x - 1] \\ \gamma_1 : & \text{times}(x, y) & \xrightarrow{1} & 0 & [x = 0] \\ \gamma_{\text{plus}}^{sz} : & \text{plus}(x, y) & \xrightarrow{x+1} & x + y & \end{array}$$

The rules created by Theorem 5.27 are suitable to perform *Outer Chaining*. This technique eliminates outer calls, i.e., it chains a rule α_1 of the form

$$f_1(\dots) \rightarrow f_2(\dots g(\dots) \dots)$$

5.3. COMPLEXITY BOUNDS FOR RNTSs

with a trivial rule α_2 of the form $f_2(\mathbf{y}) \rightarrow t$ where t is an arithmetic expression. Then α_1 is replaced by the new (chained) rule. Similarly to the preconditions of *Inner Chaining*, this is only sound if we know that the outer occurrence of f_2 in α_1 's right-hand side is eventually reduced with α_2 in cost-maximal rewrite sequences. Hence, we require that α_2 is the only f_2 -rule and that α_2 's guard is “true”. As mentioned above, the latter also ensures that the resulting chained rule is a valid RNTS rule.

Finally, we require that the cost of α_2 only contains variables from \mathbf{y} . While this requirement is not crucial for correctness, it eases the proof of Theorem 5.29 without affecting the power of our technique. The reason is that the cost of α_2 is unbounded if it depends on the value of some variable $tv \notin \mathbf{y}$, which implies that the runtime complexity of the analyzed RNTS is unbounded and hence our technique is bound to fail anyways. To see this, recall that the guard of α_2 is “true” and hence cannot impose any restrictions on tv 's value.

Theorem 5.29 (Outer Chaining). *Let \mathcal{P} be an RNTS with rules α_1, α_2 as in Definition 5.20 where $t \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$, $\mathcal{V}(c_2) \subseteq \mathbf{y}$, α_2 is the only rule with root f_2 , and $\text{guard}(\alpha_2) = \text{true}$. Moreover, let $\mathcal{P}' = \mathcal{P} \setminus \{\alpha_1\} \cup \{\alpha_{1,2}\}$. Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for size. If every variable from \mathbf{y} occurs at least once in t , then it is also sound for upper bounds.*

To prove soundness of Theorem 5.29, we need the following lemma.

Lemma 5.30 (Correctness of sz on Terms). *Let \mathcal{P} be an RNTS with size bound sz . If $t_0 \rightarrow_{\mathcal{P}}^* t_m$ for some $t_m \in \mathcal{T}(\Sigma_{\mathbb{N}})$, then $\text{sz}(t_0) \geq t_m$.*

Proof. We use induction on t_0 . Note that $t_m \in \mathcal{T}(\Sigma_{\mathbb{N}})$ implies that t_0 is ground, i.e., we have $t_0 = f(\mathbf{t})$. Let \mathbf{q} be the normal forms of \mathbf{t} in the rewrite sequence $f(\mathbf{t}) \rightarrow_{\mathcal{P}}^* t_m$. By the induction hypothesis, we have $\text{sz}(\mathbf{t}) \geq \mathbf{q}$. If $f \in \Sigma_{\mathbb{N}}$, then the lemma follows by weak monotonicity of f . If $f \in \Sigma$, then we have $\text{sz}(f)(\mathbf{q}) \geq t_m$ since sz is a size bound. Thus, we obtain $\text{sz}(f(\mathbf{t})) = \text{sz}(f)(\text{sz}(\mathbf{t})) \geq \text{sz}(f)(\mathbf{q}) \geq t_m$ by weak monotonicity of $\text{sz}(f)$. \square

Proof of Theorem 5.29. First of all, note that $\text{guard}(\alpha_2) = \text{true}$ implies that $\alpha_{1,2}$ is a valid RNTS rule. We prove soundness for size and upper bounds separately.

Claim 1. The processor is sound for upper bounds.

To prove soundness for upper bounds, we may assume that every variable from \mathbf{y} occurs at least once in t by the prerequisites of the theorem. By Lemma 5.17, it suffices to prove that

$$t_0 \xrightarrow{\mathcal{P}}^* t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \text{ with } \mathcal{E}' \geq \mathcal{E}.$$

CHAPTER 5. UPPER BOUNDS FOR RNTSS

for all terms t_0 . Instead, we prove the more general statement

$$t_0 \xrightarrow{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m \text{ implies } t_0 \xrightarrow{\mathcal{P}'}^* t'_m \text{ with } \mathcal{K}' \geq \mathcal{K}.$$

We use induction on the number of α_1 -steps in $t_0 \xrightarrow{\mathcal{P} \cup \{\alpha_{1.2}\}}^* t_m$. If there is no such step, then the claim is trivial. Otherwise, consider the last α_1 -step, i.e., we have

$$t_0 \xrightarrow{\mathcal{P}}^* t_i = t_i[f_1(\mathbf{x})\sigma_1]_\pi \xrightarrow{c_1\sigma_1}_{\alpha_1} t_i[C[f_2(\mathbf{t})]\sigma_1]_\pi \xrightarrow{\mathcal{P}'}^* t_m \quad (5.11)$$

for some position π , some integer substitution σ_1 , and some costs $\mathcal{K}_1, \mathcal{K}_2$ with $\mathcal{K}_1 + c_1\sigma_1 + \mathcal{K}_2 = \mathcal{K}$. It remains to show

$$t_i[f_1(\mathbf{x})\sigma_1]_\pi \xrightarrow{\mathcal{P}'}^* t'_m \text{ with } \mathcal{K}' \geq c_1\sigma_1 + \mathcal{K}_2. \quad (5.12)$$

Then the claim follows from the induction hypothesis.

Case 1. $f_2(\mathbf{t})\sigma_1$ is reduced to an arithmetic expression

Since RNTSSs are evaluated with an innermost strategy, we may assume that $f_2(\mathbf{t})\sigma_1$ is normalized immediately after the last α_1 -step without loss of generality, i.e., we have

$$\begin{array}{ccc} t_i[f_1(\mathbf{x})\sigma_1]_\pi & \xrightarrow{c_1\sigma_1}_{\alpha_1} & t_i[C[f_2(\mathbf{t})]\sigma_1]_\pi \\ & \xrightarrow{\mathcal{K}_{2.1}}^*_{\mathcal{P}'} & t_i[C[f_2(\mathbf{y})\sigma_2]\sigma_1]_\pi \\ & \xrightarrow{c_2\sigma_2}_{\alpha_2} & t_i[C[t\sigma_2]\sigma_1]_\pi \\ & \xrightarrow{\mathcal{K}_{2.2}}^*_{\mathcal{P}'} & t_m \end{array} \quad (5.13)$$

for some integer substitution σ_2 such that $\mathbf{y}\sigma_2$ are the normal forms of $\mathbf{t}\sigma_1$ in the rewrite sequence (5.11) and $\mathcal{K}_{2.1}, \mathcal{K}_{2.2}$ such that

$$\mathcal{K}_2 = \mathcal{K}_{2.1} + c_2\sigma_2 + \mathcal{K}_{2.2}. \quad (5.14)$$

Note that the last step of any rewrite sequence which reduces $f_2(\mathbf{t})\sigma_1$ to an arithmetic expression needs to be an α_2 -step, since α_2 is the only rule with root f_2 .

Let $\sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)}$. Since $\sigma_1 \models \varphi_1$ and $\text{guard}(\alpha_2) = \text{true}$, we have $\sigma \models \text{guard}(\alpha_{1.2})$. Hence, we have

$$\begin{array}{lll} & t_i[f_1(\mathbf{x})\sigma_1]_\pi & \\ = & t_i[f_1(\mathbf{x})\sigma]_\pi & \text{as } \sigma = \sigma_1|_{\mathcal{V}(\alpha_1)} \diamond \sigma_2|_{\mathcal{V}(\alpha_2)} \\ \xrightarrow{c_1\sigma + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma}_{\alpha_{1.2}} & t_i[C[t\{\mathbf{y}/\mathbf{t}\}]\sigma]_\pi & \text{by definition of } \alpha_{1.2} \\ = & t_i[C[t\{\mathbf{y}/\mathbf{t}\sigma_1}\sigma_2]\sigma_1]_\pi & \\ \xrightarrow{\mathcal{K}'_{2.1}}^*_{\mathcal{P}'} & t_i[C[t\{\mathbf{y}/\mathbf{y}\sigma_2}\sigma_2]\sigma_1]_\pi & \text{as } \mathbf{y}\sigma_2 \text{ are} \\ & & \text{normal forms of } \mathbf{t}\sigma_1 \\ = & t_i[C[t\sigma_2]\sigma_1]_\pi & \\ \xrightarrow{\mathcal{K}_{2.2}}^*_{\mathcal{P}'} & t_m & \text{by (5.13),} \end{array} \quad (5.15)$$

5.3. COMPLEXITY BOUNDS FOR RNTSS

i.e., we have

$$t_i \xrightarrow{\bar{\mathcal{K}}}_{\mathcal{P}'}^* t_m \text{ with } \bar{\mathcal{K}} = c_1\sigma + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2}.$$

Here, $\mathcal{K}'_{2.1}$ are the costs of normalizing all occurrences of terms from $\mathbf{t}\sigma_1$ in $t\{\mathbf{y}/\mathbf{t}\sigma_1\}$ to $\mathbf{y}\sigma_2$. Thus we have $\mathcal{K}'_{2.1} \geq \mathcal{K}_{2.1}$ since every variable from \mathbf{y} occurs at least once in t .

To finish the proof of (5.12), it remains to prove $\bar{\mathcal{K}} \geq c_1\sigma_1 + \mathcal{K}_2$. Note that we have $\text{sz}(\mathbf{t}\sigma_1) \geq \mathbf{y}\sigma_2$ by Lemma 5.30. Thus, we have

$$\begin{aligned} & c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma \\ = & c_2\{\mathbf{y}/\text{sz}(\mathbf{t}\sigma)\} && \text{as } \mathcal{V}(c_2) \subseteq \mathbf{y} \\ = & c_2\{\mathbf{y}/\text{sz}(\mathbf{t}\sigma_1)\} && \text{as } \mathcal{V}(\mathbf{t}) \subseteq \mathcal{V}(\alpha_1) \\ \geq & c_2\{\mathbf{y}/\mathbf{y}\sigma_2\} && \text{as } \text{sz}(\mathbf{t}\sigma_1) \geq \mathbf{y}\sigma_2 \text{ and } c_2 \text{ is monotonic} \\ = & c_2\sigma_2 && \text{as } \mathcal{V}(c_2) \subseteq \mathbf{y}. \end{aligned}$$

Hence, we have

$$\begin{aligned} \bar{\mathcal{K}} &= c_1\sigma + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2} \\ &= c_1\sigma_1 + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2} && \text{as } \mathcal{V}(c_1) \subseteq \mathcal{V}(\alpha_1) \\ &\geq c_1\sigma_1 + c_2\sigma_2 + \mathcal{K}_{2.1} + \mathcal{K}_{2.2} \\ &= c_1\sigma_1 + \mathcal{K}_2 && \text{by (5.14)} \end{aligned}$$

as desired.

Case 2. $\mathbf{f}_2(\mathbf{t})\sigma_1$ is not reduced to an arithmetic expression

Let π' be the unique position of \square in C . Then we have $t_m|_{\pi.\pi'} = \mathbf{f}_2(\mathbf{t}')$ for some terms \mathbf{t}' such that $\mathbf{t}\sigma_1 \rightarrow_{\mathcal{P}'}^* \mathbf{t}'$. Since RNTSSs are evaluated with an innermost strategy, we may assume that the terms $\mathbf{t}\sigma_1$ are reduced to \mathbf{t}' directly after the last α_1 -step without loss of generality. Thus, we have

$$t_i[\mathbf{f}_1(\mathbf{x})\sigma_1]_{\pi} \xrightarrow{\frac{c_1\sigma_1}{\mathcal{K}_{2.1}}}_{\alpha_1} t_i[C[\mathbf{f}_2(\mathbf{t})]\sigma_1]_{\pi} \xrightarrow{\frac{\mathcal{K}_{2.1}}{\mathcal{K}_{2.2}}}_{\mathcal{P}'}^* t_i[C[\mathbf{f}_2(\mathbf{t}')]_{\sigma_1}]_{\pi} \xrightarrow{\mathcal{K}_{2.2}}_{\mathcal{P}'}^* t_m$$

with

$$\mathcal{K}_2 = \mathcal{K}_{2.1} + \mathcal{K}_{2.2} \tag{5.16}$$

where all rewrite steps $t_i[C[\mathbf{f}_2(\mathbf{t}')]_{\sigma_1}]_{\pi} \rightarrow_{\mathcal{P}'}^* t_m$ reduce positions which are independent from $\pi.\pi'$. Hence, we get

$$t_i[C[q]\sigma_1]_{\pi} \xrightarrow{\mathcal{K}_{2.2}}_{\mathcal{P}'}^* t_m[q]_{\pi.\pi'} \tag{5.17}$$

for all terms q by applying the same rules in the same order at the same positions with the same substitutions.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

Since $\text{guard}(\alpha_2) = \text{true}$, we have $\sigma_1 \models \text{guard}(\alpha_{1.2})$. Thus, we get

$$\begin{array}{ccc}
 \frac{c_1\sigma_1 + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma_1}{\xrightarrow[\mathcal{P}']{*}} & \xrightarrow{\alpha_{1.2}} & \begin{array}{l} t_i[f_1(\mathbf{x})\sigma_1]_\pi \\ t_i[C[t\{\mathbf{y}/\mathbf{t}\}]\sigma_1]_\pi \quad \text{by definition of } \alpha_{1.2} \\ t_i[C[t\{\mathbf{y}/\mathbf{t}'\}]\sigma_1]_\pi \quad \text{as } \mathbf{t}\sigma_1 \xrightarrow[\mathcal{P}']{*} \mathbf{t}' \\ t_m[t\{\mathbf{y}/\mathbf{t}'\}\sigma_1]_{\pi.\pi'} \quad \text{by (5.17)} \\ t'_m \end{array} \\
 \frac{\mathcal{K}'_{2.1}}{\xrightarrow[\mathcal{P}']{*}} & & \\
 \frac{\mathcal{K}_{2.2}}{\xrightarrow[\mathcal{P}']{*}} & & \\
 = & &
 \end{array} \quad (5.18)$$

i.e., we have

$$t_i \xrightarrow[\alpha_{1.2}]{\overline{\mathcal{K}}} t'_m \text{ where } \overline{\mathcal{K}} = c_1\sigma_1 + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma_1 + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2}$$

where $\mathcal{K}'_{2.1}$ is the cost of reducing all occurrences of terms from $\mathbf{t}\sigma_1$ in $t\{\mathbf{y}/\mathbf{t}\}\sigma_1$ to \mathbf{t}' . Hence, we have $\mathcal{K}'_{2.1} \geq \mathcal{K}_{2.1}$, since every variable from \mathbf{y} occurs at least once in t .

To prove (5.12), it remains to prove $\overline{\mathcal{K}} \geq c_1\sigma_1 + \mathcal{K}_2$. We have

$$\begin{aligned}
 \overline{\mathcal{K}} &= c_1\sigma_1 + c_2\{\mathbf{y}/\text{sz}(\mathbf{t})\}\sigma_1 + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2} \\
 &\geq c_1\sigma_1 + \mathcal{K}'_{2.1} + \mathcal{K}_{2.2} \\
 &\geq c_1\sigma_1 + \mathcal{K}_{2.1} + \mathcal{K}_{2.2} \\
 &= c_1\sigma_1 + \mathcal{K}_2 \quad \text{by (5.16)}
 \end{aligned}$$

as desired.

Claim 2. The processor is sound for size.

It suffices to prove

$$t_0 \xrightarrow[\mathcal{P}']{*} t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \xrightarrow[\mathcal{P}']{*} t_m.$$

Instead, we prove the more general statement

$$t_0 \xrightarrow[\mathcal{P} \cup \{\alpha_{1.2}\}]{*} t_m \text{ with } t_m \in \mathcal{T}(\Sigma_{\mathbb{N}}) \text{ implies } t_0 \xrightarrow[\mathcal{P}']{*} t_m.$$

As in the proof of *Claim 1*, we use induction on the number of α_1 -steps in $t_0 \xrightarrow[\mathcal{P} \cup \{\alpha_{1.2}\}]{*} t_m$. Note that we just have to consider *Case 1*, as the rewrite sequence does not end with an arithmetic expression in *Case 2*. Indeed, we have $t'_m = t_m$ in the proof of *Case 1*. Thus, the proof of *Claim 2* is analogous to the proof of *Claim 1*, *Case 1*. \square

Example 5.31 (Outer Chaining – Example 5.28 finished). The rule $\gamma_{\text{plus}}^{\text{sz}}$ is suitable to perform outer chaining. Thus, we obtain the following RNTS by applying Theorem 5.29:

$$\begin{array}{lll}
 \gamma_{0.\text{plus}}^{\text{sz}} : & \text{times}(x, y) & \xrightarrow{\text{sz}(\text{times})(x', y) + 2} \text{times}(x', y) + y \quad [x > 0 \wedge x' = x - 1] \\
 \gamma_1 : & \text{times}(x, y) & \xrightarrow{1} 0 \quad [x = 0] \\
 \gamma_{\text{plus}}^{\text{sz}} : & \text{plus}(x, y) & \xrightarrow{x+1} x + y
 \end{array}$$

5.3. COMPLEXITY BOUNDS FOR RNTSs

Here, $\text{sz}(\text{times})$ is left abstract, since we do not know a size bound for times yet. However, the costs of $\gamma_{0,\text{plus}}^{\text{sz}}$ are not needed to compute such a size bound. Thus, since the RNTS above is also an ITS, standard complexity analysis tools can be applied to compute $\text{sz}(\text{times})(x, y) = x \cdot y$ via Theorem 5.11. Now $\gamma_{0,\text{plus}}^{\text{sz}}$ can be refined to

$$\text{times}(x, y) \xrightarrow{x' \cdot y + 2} \text{times}(x', y) + y \ [x > 0 \wedge x' = x - 1].$$

Finally, standard complexity analysis tools can be used to compute a runtime bound like $\text{rt}(\text{times})(x, y) = x^2 \cdot y + 2 \cdot x + 1$. Thus, the runtime complexity of the RNTS from Example 5.4 is in $\mathcal{O}(n^3)$ by Theorem 5.9.

To see why soundness for upper bounds requires that every variable from α_2 's left-hand side has to occur at least once in α_2 's right-hand side, assume that we chain $\alpha_1 = g(x) \xrightarrow{1} h(f(x))$ with $\alpha_2 = h(x) \xrightarrow{1} 0$. Then we obtain the rule $g(x) \xrightarrow{2} 0$, which does not take the costs of f into account. If we chain α_1 with $h(x) \xrightarrow{1} x$ instead, then we obtain $g(x) \xrightarrow{2} f(x)$, i.e., then the call to f on α_1 's right-hand side is preserved.

Since *Size Simplification* does not ensure that every variable from the left-hand side occurs at least once in the right-hand side of the resulting trivial rule, we now adapt *Size Simplification* such that the resulting rules are always suitable for *Outer Chaining*.

Theorem 5.32 (Outer Simplification). *Let \mathcal{P} be an RNTS over Σ with runtime and size bounds rt and sz . For each $f \in \Sigma^o$ let*

$$\alpha_f = f(\mathbf{x}) \xrightarrow{\text{rt}(f)(\mathbf{x})} r + 0 \cdot \sum_{x \in \mathbf{x} \setminus \mathcal{V}(r)} x$$

where $r = tv \in \mathcal{V}$ is a fresh variable if $\text{sz}(f)(\mathbf{x}) = \omega$ and $r = \text{sz}(f)(\mathbf{x})$ otherwise. Furthermore, let

$$\mathcal{P}' = \{\alpha \in \mathcal{P} \mid \text{root}(\alpha) \notin \Sigma^o\} \cup \{\alpha_f \mid f \in \Sigma^o\}.$$

Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for upper bounds and size.

Proof. The proof is analogous to the proof of Theorem 5.27. □

However, the technique presented so far has one serious drawback.

Example 5.33 (Duplicating Redexes). Let $\alpha_1 = g(x) \xrightarrow{1} h(f(x))$ and $\alpha_2 = h(x) \xrightarrow{1} x + x$. Then outer chaining yields $g(x) \xrightarrow{2} f(x) + f(x)$, i.e., the call to f on α_1 's right-hand side is duplicated. As a result, its costs are taken into account twice, which increases the runtime complexity of the RNTS.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

Hence, for correctness every variable from α_2 's left-hand side has to occur at least once in α_2 's right-hand side, but if such a variable occurs more than once, then we lose precision. To solve this problem, we can use the following optimization: Before applying *Outer Chaining*, we add variants with cost 0 of all rules to the analyzed RNTS.

Theorem 5.34 (Duplication). *Let \mathcal{P} be an RNTS over Σ and let $\Sigma_0 = \{f_0 \mid f \in \Sigma\}$. Let \mathcal{P}_0 result from \mathcal{P} by replacing all function symbols from Σ with the corresponding function symbols from Σ_0 and by setting the costs of all rules to 0. Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P} \cup \mathcal{P}_0)$ is sound for runtime and size.*

Proof. The processor is sound since \mathcal{P} and \mathcal{P}_0 are RNTSs over disjoint signatures. Thus, every \mathcal{P} -rewrite sequence starting with a term from $\mathcal{T}_{\text{basic}}(\mathcal{P})$ is also a valid rewrite sequence w.r.t. $\mathcal{P} \cup \mathcal{P}_0$. \square

Then whenever *Outer Chaining* duplicates a term t , we replace all but one occurrence of t in the right-hand side of the chained rule by s which results from t by replacing each function symbol from Σ with the corresponding function symbol from Σ_0 . In this way, the cost of evaluating t is taken into account at most once. To this end, we introduce the following variant of Definition 5.20

Definition 5.35 (Runtime Preserving Chaining). Let \mathcal{P} be an RNTS with size bound sz which contains the rules:

$$\begin{array}{lll} \alpha_1 : f_1(\mathbf{x}) & \xrightarrow{c_1} C[f_2(\mathbf{t})] & [\varphi_1] \\ \alpha_2 : f_2(\mathbf{y}) & \xrightarrow{c_2} t & [\varphi_2] \end{array}$$

W.l.o.g., assume $\mathcal{V}(\alpha_1) \cap \mathcal{V}(\alpha_2) = \emptyset$ (otherwise, one can rename the variables in one rule accordingly). Moreover, let t' result from linearizing t w.r.t. \mathbf{y} , i.e., no variable from \mathbf{y} occurs more than once in t' , $\mathcal{V}(t) \subseteq \mathcal{V}(t')$, all variables in $\mathcal{V}(t') \setminus \mathcal{V}(t)$ are fresh, and $t'\mu = t$ for some substitution $\mu : \mathcal{V} \rightarrow \mathbf{y}$ with $\text{dom}(\mu) = \mathcal{V}(t') \setminus \mathcal{V}(t)$. Finally, let \mathbf{s} be terms such that $\mathcal{V}(\mathbf{s}) = \mathcal{V}(\mathbf{t})$ and $\mathbf{t}|_i \theta \xrightarrow{*}_{\mathcal{P}} n \in \mathcal{T}(\Sigma_{\mathbb{N}})$ implies $\mathbf{s}|_i \theta \xrightarrow{*}_{\mathcal{P}} n$ for all $i \in \{1, \dots, \text{len}(\mathbf{t})\}$ and all integer substitutions θ . Then *runtime preserving chaining* of α_1 and α_2 yields

$$\alpha_{1.2} : f_1(\mathbf{x}) \xrightarrow{c_1 + c_2 \{ \mathbf{y} / sz(\mathbf{t}) \}} C[t' \{ \mathbf{y} / \mathbf{t} \} \mu \{ \mathbf{y} / \mathbf{s} \}] \quad [\varphi_1 \wedge \varphi_2 \{ \mathbf{y} / \mathbf{t} \}].$$

So after applying Theorem 5.34, the terms \mathbf{s} in Definition 5.35 can be obtained by replacing all function symbols from Σ in \mathbf{t} with the corresponding function symbols from Σ_0 . Then evaluating $\mathbf{s}\theta$ has cost 0 for all integer substitutions θ . The right-hand side $C[t' \{ \mathbf{y} / \mathbf{t} \} \mu \{ \mathbf{y} / \mathbf{s} \}]$ of the chained rule contains each term from \mathbf{t} at most once (since t' is linear w.r.t. \mathbf{y}). All remaining occurrences of variables from \mathbf{y} in t are replaced with the corresponding terms from \mathbf{s} by

5.3. COMPLEXITY BOUNDS FOR RNTSs

applying the substitutions μ and $\{\mathbf{y}/\mathbf{s}\}$. Now we can use the following variant of *Outer Chaining*.

Theorem 5.36 (Runtime Preserving Outer Chaining). *Let \mathcal{P} be an RNTS with rules α_1, α_2 as in Definition 5.35 where $t \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V})$, $\mathcal{V}(c_2) \subseteq \mathbf{y}$, α_2 is the only rule with root \mathbf{f}_2 , and $\text{guard}(\alpha_2) = \text{true}$. Moreover, let $\mathcal{P}' = \mathcal{P} \setminus \{\alpha_1\} \cup \{\alpha_{1.2}\}$. Then the processor mapping $cp(\mathcal{P})$ to $cp(\mathcal{P}')$ is sound for size. If every variable from \mathbf{y} occurs at least once in t , then it is also sound for upper bounds.*

Proof. We adapt the proof from Theorem 5.29. Instead of (5.15), we now obtain

$$\begin{aligned}
& \frac{c_1 \sigma + c_2 \{\mathbf{y}/\text{sz}(\mathbf{t})\} \sigma}{\xrightarrow{\mathcal{R}'_{2.1}}_{\mathcal{P}'}} \quad \begin{array}{l} t_i[\mathbf{f}_1(\mathbf{x})\sigma]_{\pi} \\ t_i[C[t'\{\mathbf{y}/\mathbf{t}\}\mu\{\mathbf{y}/\mathbf{s}\}]\sigma]_{\pi} \\ t_i[C[t'\{\mathbf{y}/\mathbf{t}\sigma_1\}\mu\{\mathbf{y}/\mathbf{s}\sigma_1\}\sigma_2]\sigma_1]_{\pi} \\ t_i[C[t'\{\mathbf{y}/\mathbf{y}\sigma_2\}\mu\{\mathbf{y}/\mathbf{y}\sigma_2\}\sigma_2]\sigma_1]_{\pi} \end{array} \quad \begin{array}{l} \text{by definition of } \alpha_{1.2} \\ \\ \text{as } \mathbf{y}\sigma_2 \text{ are} \\ \text{normal forms of } \mathbf{t}\sigma_1 \\ \text{and hence also of } \mathbf{s}\sigma_1 \end{array} \\
& = \quad t_i[C[t'\mu\{\mathbf{y}/\mathbf{y}\sigma_2\}\sigma_2]\sigma_1]_{\pi} \quad \text{as } \text{dom}(\mu) \cap \mathbf{y} = \emptyset \\
& = \quad t_i[C[t\{\mathbf{y}/\mathbf{y}\sigma_2\}\sigma_2]\sigma_1]_{\pi} \quad \text{as } t'\mu = t \\
& = \quad t_i[C[t\sigma_2]\sigma_1]_{\pi} \\
& \xrightarrow{\mathcal{R}'_{2.2}}_{\mathcal{P}'} \quad t_m \quad \text{by (5.13)}
\end{aligned}$$

and instead of (5.18), we now obtain

$$\begin{aligned}
& \frac{c_1 \sigma_1 + c_2 \{\mathbf{y}/\text{sz}(\mathbf{t})\} \sigma_1}{\xrightarrow{\mathcal{R}'_{2.1}}_{\mathcal{P}'}} \quad \begin{array}{l} t_i[\mathbf{f}_1(\mathbf{x})\sigma_1]_{\pi} \\ t_i[C[t'\{\mathbf{y}/\mathbf{t}\}\mu\{\mathbf{y}/\mathbf{s}\}]\sigma_1]_{\pi} \\ t_i[C[t'\{\mathbf{y}/\mathbf{t}'\}\mu\{\mathbf{y}/\mathbf{s}\}]\sigma_1]_{\pi} \\ t_m[t'\{\mathbf{y}/\mathbf{t}'\}\mu\{\mathbf{y}/\mathbf{s}\}\sigma_1]_{\pi} \end{array} \quad \begin{array}{l} \text{by definition of } \alpha_{1.2} \\ \\ \text{as } \mathbf{t}\sigma_1 \xrightarrow{\mathcal{R}'_{2.1}}_{\mathcal{P}'} \mathbf{t}' \\ \text{by (5.17)} \end{array} \\
& \xrightarrow{\mathcal{R}'_{2.2}}_{\mathcal{P}'} \quad t'_m.
\end{aligned}$$

The rest of the proof remains unchanged. \square

Example 5.37 (Runtime Preserving Outer Chaining – Example 5.33 continued). After applying *Duplication*, applying *Runtime Preserving Outer Chaining* to the rules from Example 5.33 yields $\mathbf{g}(x) \xrightarrow{2} \mathbf{f}(x) + \mathbf{f}_0(x)$ where $\mathbf{f}(x)\theta \xrightarrow{\mathcal{P}} n \in \mathcal{T}(\Sigma_{\mathbb{N}})$ implies $\mathbf{f}_0(x)\theta \xrightarrow{0}_{\mathcal{P}} n$ for all integer substitutions θ . Thus, the costs of $\mathbf{f}(x)$ are taken into account only once.

Our overall algorithm to infer bounds for RNTSs is summarized in Algorithm 3. It clearly terminates, as in every loop iteration, we either fail to compute a runtime bound for a function symbol from $\Sigma(\mathcal{P}^f)$ and thus return immediately, or we obtain a finite runtime bound for \mathbf{f} , i.e., the number of function symbols whose runtime bound is ω decreases.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

For soundness of our algorithm, note that runtime and size bounds for induced sub-RNTSSs are clearly also valid for the overall RNTS, which justifies Step 3.7 and Step 3.11.

Moreover, note that we obtain an ITS after Step 3.5. The reason is that we have $\text{rt}(\mathbf{f}) \neq \omega$ for all $\mathbf{f} \in \Sigma^o$ after Step 3.2 (otherwise, $\mathcal{P}^{\mathbf{f}}$ had nested recursion). So all inner calls to these function symbols are eliminated in Step 3.2 and all remaining (outer) calls are eliminated in Step 3.3 and 3.5. Thus, we have $\Sigma^o = \emptyset$ after Step 3.5 which, according to Corollary 5.26, means that we obtain an ITS.

When implementing Algorithm 3, several improvements are possible. First of all, it is not always ideal to continue the analysis with one of the smallest symbols \mathbf{f} such that $\text{rt}(\mathbf{f})(\mathbf{x}) = \omega$. E.g., the RNTS which results from *Inner Chaining* in Example 5.22 is already an ITS and hence can be analyzed with existing tools directly instead of continuing the analysis in a bottom-up fashion. Moreover, when analyzing $\mathcal{P}^{\mathbf{f}}$ the result of Steps 3.2 – 3.5 and 3.8 can be cached and reused later when analyzing some $\mathcal{P}^{\mathbf{g}}$ with $\mathbf{g} \sqsupset \mathbf{f}$.

Algorithm 3 Computing Runtime Bounds for RNTSSs

- 1 Let $\text{rt}(\mathbf{f})(\mathbf{x}) := \text{sz}(\mathbf{f})(\mathbf{x}) := \omega$ for each $\mathbf{f} \in \Sigma$
 - 2 If \mathcal{P} has nested recursion, then return rt
 - 3 While there is an $\mathbf{f} \in \Sigma$ with $\text{rt}(\mathbf{f})(\mathbf{x}) = \omega$:
 - 3.1 Set $\mathcal{P}' := \mathcal{P}^{\mathbf{f}}$ where \mathbf{f} is one of the smallest symbols with $\text{rt}(\mathbf{f})(\mathbf{x}) = \omega$
 - 3.2 Apply *Inner Simplification* and *Inner Chaining* to \mathcal{P}' exhaustively
 - 3.3 Apply *Outer Simplification* to \mathcal{P}'
 - 3.4 Apply *Duplication* to \mathcal{P}'
 - 3.5 Apply *Runtime Preserving Outer Chaining* to \mathcal{P}' exhaustively where the costs of the chained rules are left abstract
 - 3.6 Compute a size bound $\text{sz}_{\mathbf{f}}$ for \mathcal{P}' using existing tools via Theorems 5.11 and A.1 if possible
 - 3.7 Set $\text{sz}(\mathbf{g}) := \text{sz}_{\mathbf{f}}(\mathbf{g})$ for each $\mathbf{g} \in \Sigma(\mathcal{P}')$
 - 3.8 Instantiate the costs in the chained rules from Step 3.5
 - 3.9 Compute a runtime bound $\text{rt}_{\mathbf{f}}$ for \mathcal{P}' using existing tools
 - 3.10 If $\text{rt}_{\mathbf{f}}(\mathbf{g})(\mathbf{x}) = \omega$ for some $\mathbf{g} \in \Sigma(\mathcal{P}')$, return rt
 - 3.11 Set $\text{rt}(\mathbf{g}) := \text{rt}_{\mathbf{f}}(\mathbf{g})$ for each $\mathbf{g} \in \Sigma(\mathcal{P}')$
 - 4 Return rt
-

5.4 Related Work

There exist several approaches that also analyze complexity by inferring both runtime and size bounds. Wegbreit [128] tries to generate closed forms for the exact runtime and size of the result of each analyzed function, whereas we estimate runtime and size by upper bounds. Hence, [128] fails whenever finding such exact closed forms automatically is infeasible. Serrano et al. [112] also compute runtime and size bounds, but in contrast to us they work on logic programs, and their approach is based on abstract interpretation. Our technique in Section 5.3 was inspired by the tool KoAT [27], which composes results of alternating size and runtime complexity analyses for ITSs. KoAT also supports a “bottom-up” technique that corresponds to the approach of Section 5.3 when restricting it to standard ITSs *without (non-tail) recursion*. But in contrast to Section 5.3, KoAT’s support for recursion is very limited, as it disregards the return values of “inner” calls. Moreover, [27] does not contain an approach like Theorem 5.11 in Section 5.2 which allows us to obtain size bounds from techniques that compute runtime bounds.

RaML [82, 83, 84] reduces the inference of resource annotated types (and hence complexity bounds) for ML programs to linear optimization. Like our technique, RaML’s support for arithmetic is currently restricted to natural number [85]. With respect to modularity, RaML has two theoretical boundaries [83]: (A) The number of linear constraints arising from type inference grows exponentially in the size of the program. (B) To achieve context-sensitivity, functions are typed differently for different invocations. In our setting, a blow-up similar to (A) may occur within the used ITS tool, but as the program is analyzed one function at a time, this blow-up is exponential in the size of a single function instead of the whole program. To avoid (B), we analyze each function only once. However, RaML takes amortization effects into account and obtains impressive results in practice.

C⁴B [35] adapts RaML’s approach to C programs. However, C⁴B can only infer linear bounds automatically and requires user interaction to infer non-linear bounds. Recently, this limitation has been overcome [34], resulting in the new tool Pastis which analyzes the complexity of LLVM programs.

Another leading tool for the inference of complexity bounds for recursive integer programs is CoFloCo [48, 50]. It analyzes *cost relations*, which correspond to possibly recursive integer programs where procedures may have several outputs. The same formalism is analyzed by the earlier tool PUBS [1, 3], which inspired CoFloCo. To achieve modularity, both PUBS and CoFloCo analyze program parts independently and use linear invariants to compose the results. So their approaches differ significantly from Section 5.3, which can also infer non-linear size bounds. Thus, the technique from Section 5.3 is especially suitable for examples where non-linear growth of data causes non-linear runtime. For instance, in Example 5.31 the quadratic size bound for `times` is crucial to prove a (tight) cubic runtime bound with the technique of Section 5.3. Consequently, linear invariants are not sufficient and hence CoFloCo and PUBS fail for this RNTS.

CHAPTER 5. UPPER BOUNDS FOR RNTSS

However, CoFloCo’s amortized analysis often results in very precise bounds, i.e., it is orthogonal to our approach. Similarly, PUBS often infers very precise concrete bounds, whereas we mainly focus on asymptotic bounds. Moreover, both PUBS and CoFloCo can also infer best-case lower bounds, which is not possible with our technique. Also, the *cost expressions* resp. *cost structures* which are used by PUBS resp. CoFloCo are more expressive than the expressions supported by our technique. We chose such a restrictive format for two reasons: First, it is supported by all existing complexity analysis tools for ITSs or related formalisms, such that our approach remains independent from the underlying analyzer for linear ITSs. Second, our approach exploits that our expressions are monotonic and extending it to richer, non-monotonic expressions is non-trivial. Finally, the semantics of cost relations allow to handle tail and non-tail recursion uniformly, whereas non-tail recursion is challenging in our setting.

Furthermore, there are various other tools to analyze the complexity of non-recursive integer programs, i.e., ITSs or related formalism. Examples include ABC [21], Loopus [114], Rank [8], and SPEED [72]. These approaches and our technique from Section 5.3 complement each other, since they lack support for recursive programs, whereas our technique relies on existing tools for the analysis of non-recursive integer programs in order to analyze (non-tail) recursive programs.

Finally, the recent tool CAMPY [116] *verifies* that a given expression is a (concrete) upper bound for the analyzed program. In contrast, our technique infers upper bounds automatically.

5.5 Experiments

We implemented our contributions in the tool **AProVE** [62]. As mentioned in Section 1.3.2, the technique presented in this chapter was initially developed as backend for a transformation from term rewrite systems to integer programs [103]. Thus, we evaluated its power on 919 examples of the category “Runtime Complexity - Innermost Rewriting” of the *Termination and Complexity Competition 2016* [121] which were transformed to RNTS by **AProVE** as described in [103]. Here, we excluded the 103 examples where **AProVE** shows $\text{irc}(n) \in \Omega(\omega)$. Note that, in this category, the examples from the *Termination and Complexity Competition 2017* are a subset of the examples from 2016. All TRSs were pre-processed with the technique from [53] to remove rules which are not reachable from basic terms.

In our experiments, we analyzed the RNTSs resulting from **AProVE**’s transformation with **AProVE**’s implementation of the technique from Section 5.3 and with **CoFloCo**. Thereby, **AProVE** used the external tools **CoFloCo** and **KoAT** to compute runtime bounds for the ITSs resulting from the technique in Section 5.3. We did not compare with **RaML** and **Pastis**, since their input languages (ML resp. LLVM) differ significantly from our RNTSs and **CoFloCo**’s cost relations. Moreover, **Pastis** is not publicly available at the moment.

While we restricted ourselves to polynomial arithmetic for simplicity in this thesis, **KoAT**’s ability to prove exponential bounds for ITSs also enables **AProVE** to infer exponential upper bounds for some RNTSs. Thus, the capabilities of the back-end ITS tool determine which kinds of bounds can be derived by **AProVE**.

Table 5.1 shows the results of our experiments. We used a timeout of 60 seconds per example. While **CoFloCo** often infers more precise bounds than **AProVE** (**CoFloCo** proves a smaller bound than **AProVE** in 113 cases), the results also clearly show that both approaches are orthogonal: In 36 cases, **AProVE** successfully infers an upper bound, whereas **CoFloCo** fails. In 24 of these cases, **AProVE** infers at least one super-linear size bound, i.e., these examples are potentially infeasible for **CoFloCo** which just infers linear invariants and hence cannot track super-linear growth of data. From the 10 examples where **AProVE** infers an exponential upper bound and **CoFloCo** fails, **AProVE** infers exponential

| | | CoFloCo | | | | | | | | |
|--------|-----------------------|------------------|------------------|--------------------|--------------------|--------------------|--------------------|-----------------------|-------|-----------------------|
| AProVE | $\text{rc}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n^5)$ | $\mathcal{O}(n^{10})$ | EXP | $\mathcal{O}(\omega)$ |
| | $\mathcal{O}(1)$ | 43 | – | – | – | – | – | – | – | – |
| | $\mathcal{O}(n)$ | 1 | 158 | – | – | – | – | – | – | 2 |
| | $\mathcal{O}(n^2)$ | – | 29 | 54 | – | – | – | – | – | 5 |
| | $\mathcal{O}(n^3)$ | – | – | 2 | 9 | – | – | – | – | 16 |
| | $\mathcal{O}(n^4)$ | – | – | 1 | – | – | – | – | – | 3 |
| | $\mathcal{O}(n^5)$ | – | – | – | – | – | 2 | – | – | – |
| | $\mathcal{O}(n^{10})$ | – | – | – | – | – | – | 1 | – | – |
| | EXP | – | 2 | – | – | – | – | – | – | 10 |
| | $\mathcal{O}(\omega)$ | – | 24 | 48 | 6 | – | – | – | – | 503 |

Table 5.1: AProVE vs. CoFloCo

CHAPTER 5. UPPER BOUNDS FOR RNTSS

size bounds in 8 cases, i.e., AProVE can also handle cases where exponential runtime is caused by exponential growth of data. According to [49], this kind of reasoning has so far only been supported by KoAT. See Appendix B for a list of those examples where our technique succeeds by computing super-linear polynomial resp. exponential size bounds, whereas CoFloCo fails.

The versions of AProVE, CoFloCo, and KoAT that we used as well as the strategies to run AProVE with CoFloCo resp. the technique from Section 5.3 as backend are available at [52].

One of the most important reasons why CoFloCo outperforms AProVE in many cases is AProVE's restriction to monotonic bounds. To see this, consider the following example.

Example 5.38. The technique from Section 5.3 cannot infer a runtime bound for the RNTS consisting of the following two rules:

$$\begin{array}{ccc} f(x) & \xrightarrow{1} & f(\text{minus}(x, 1)) \\ \text{minus}(x, y) & \xrightarrow{1} & x' \quad [x' = x - y] \end{array}$$

The reason is that AProVE cannot infer $\text{sz}(\text{minus}) = x - y$, since $x - y$ is not monotonic. Instead, AProVE can at best deduce the imprecise size bound $\text{sz}(\text{minus}) = x$. Then eliminating the inner call to `minus` in the first rule yields the non-terminating rule $f(x) \xrightarrow{2} f(tv) [tv \leq x]$.

In contrast, the *cost structures* which are internally used by CoFloCo may also contain non-monotonic expressions. Thus, in future work one should extend the technique presented in this chapter to less restrictive classes of size and runtime bounds.

5.6 Conclusion and Future Work

We presented a modular approach to lift any technique that infers upper bounds on the runtime complexity of ITSs to handle (non-nested, but otherwise *arbitrary*) recursion as well. The main idea of our approach is to summarize functions via *runtime* and *size bounds*. Then the rules for a successfully analyzed function f can be replaced by a *trivial rule* which immediately evaluates to an arithmetic expression. Afterwards, calls of previously analyzed functions can be eliminated via *chaining*. In this way, sub-RNTSs with nested function calls are transformed into ITSs which can be analyzed by existing tools. These existing tools are also used to compute size bounds via a novel encoding. Given an ITS \mathcal{P} , it constructs an ITS \mathcal{P}_\uparrow whose runtime corresponds to the size of the result computed by \mathcal{P} .

The main challenge of our approach is to handle functions which behave non-monotonically, i.e., functions whose runtime or result might decrease for greater arguments. To this end, we use different techniques to eliminate inner calls (i.e., calls whose result is passed to other functions) and outer calls (i.e., calls that take the result of other functions as arguments) of previously analyzed functions.

Clearly, our approach is useful for the analysis of recursive arithmetic programs in general. However, building upon a suitable *size abstraction* which maps data structures to natural numbers, it can also be used to analyze other kinds of programs like term rewrite systems [103]. Finally, to infer runtime bounds, we also compute size bounds, which may be useful on their own as well.

The main limitations of the presented approach are its restriction to natural numbers and, as mentioned in Section 5.5, to monotonic bounds. While support for integers could be achieved by computing bounds in terms of the absolute values of the program variables, this approach has serious restrictions w.r.t. precision. To see this, consider the following variant of Example 5.38.

Example 5.39. Assume that the variables in the following rewrite system range over \mathbb{Z} .

$$\begin{array}{ll} f(x) & \xrightarrow{1} f(\text{minus}(x, 1)) \ [x > 0] \\ \text{minus}(x, y) & \xrightarrow{1} x - y \end{array}$$

In terms of the absolute values of x and y , the optimal size bound for `minus` is $|x| + |y|$. So while we failed to infer that evaluating `minus(x , 1)` decreases the value of x in Example 5.38, we now even fail to infer that evaluating `minus(x , 1)` does not increase the value of x .

Hence, lifting the presented technique to integers requires a more subtle approach. Computing *lower* and *upper* size bound might be a more promising idea. Then $x - y$ would be a valid lower as well as upper size bound for `minus` in Example 5.39. However, $x - y$ is not monotonic. Thus, this idea essentially

CHAPTER 5. UPPER BOUNDS FOR RNTSS

requires non-monotonic bounds, which shows that the problems of supporting integers and non-monotonic bounds are interrelated.

Part III

Complexity Analysis of Term Rewrite Systems

Introduction

We now consider term rewrite systems (TRSs), i.e., rewrite systems operating on tree-shaped data structures. As explained in Section 1.2, such rewrite systems are an important tool for the analysis of programs operating on data structures like trees or lists.

There exist numerous methods to infer *upper bounds* for the runtime complexity of TRSs [15, 79, 86, 87, 105, 125, 131]. While we also present a new technique which is specific to upper bounds in Chapter 10, we start with the first automatic approaches to infer *lower bounds* for the runtime complexity of TRSs in Chapters 8 and 9 after introducing preliminaries in Chapter 7. The technique presented in Chapter 11 has applications for both, lower and upper bounds.

While most methods to infer upper bounds are adaptations of termination techniques, our approaches for the inference of lower bounds are related to methods that prove non-termination of TRSs. The *loop detection* technique from Chapter 8 is based on *decreasing loops*, a generalization of *loops*. Loops are used to prove non-termination of TRSs [67, 108, 125, 132, 134], i.e., the existence of a loop gives rise to a non-terminating rewrite sequence. In contrast, decreasing loops give rise to families of rewrite sequences with linear, exponential, or infinite runtime complexity.

The *induction technique* from Chapter 9 uses automated induction proofs to show the existence of certain families of rewrite sequences and infers lower bounds on the complexity of these families by analyzing the structure of the inductive proofs. It is inspired by the technique to prove non-termination of (possibly non-looping) TRSs from [41]. Both techniques generate “meta-rules” (called *rewrite lemmas* in this thesis) which represent infinitely many rewrite sequences. However, our rewrite lemmas are more general than the meta-rules in [41], as they can be parameterized by *several* variables.

Chapter 10 results from the observation that *loop detection* can prove linear lower bounds in almost all cases. Thus, in Chapter 8 we investigate the question if the existence of a linear lower bound is semi-decidable for certain classes of TRSs. For the sake of completeness, we also consider the following, closely

CHAPTER 6. INTRODUCTION

related question: Is it semi-decidable whether a TRS has constant runtime complexity? A positive answer to this question, together with a semi-decision procedure, is presented in Chapter 10. In combination with our *loop detection* technique, it can be used to prove or disprove constant complexity in almost all cases (cf. Chapter 12).

Chapter 11 presents a powerful sufficient criterion to prove that innermost and full runtime complexity coincide. In this way, all existing (and future) techniques for the inference of upper bounds on the innermost runtime complexity also become applicable for the inference of upper bounds on the full runtime complexity of a large class of term rewrite systems. Dual, techniques for the inference of lower bounds on the full runtime complexity can also be used to analyze the innermost runtime complexity of term rewrite systems. In the case of upper bounds, this results in a significant improvement of the state of the art, as shown in the extensive experimental evaluation of all presented techniques for term rewriting in Chapter 12.

Preliminaries

Term Rewrite Systems We now introduce our program model, i.e., *term rewrite systems*. A term rewrite system is a set of rules where the left-hand sides and the right-hand sides are terms.

Definition 7.1 (Term Rewrite System). Let Σ be a finite signature and let \mathcal{V} be a set of variables. A term rewrite rule over Σ and \mathcal{V} is of the form $\ell \xrightarrow{k} r$ where $\ell \in \mathcal{T}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$, $r \in \mathcal{T}(\Sigma, \mathcal{V}(\ell))$, and $k \in \{0, 1\}$. A *term rewrite system* (TRS) over Σ and \mathcal{V} is a set of term rewrite rules over Σ and \mathcal{V} .

A TRS is called *left linear* (resp. *right linear*) if ℓ (resp. r) is linear for each $\ell \xrightarrow{k} r \in \mathcal{R}$. It is *linear* if it is left and right linear. If all of its rules have cost 1, then we call a TRS *ordinary*.

We usually just write $\ell \rightarrow r$ instead of $\ell \xrightarrow{1} r$. Moreover, we lift Σ to rules by defining $\Sigma(\ell \xrightarrow{k} r) = \Sigma(\{\ell, r\})$ and we lift \mathcal{V} to rules analogously. Finally, we define $\text{root}(\ell \xrightarrow{k} r) = \text{root}(\ell)$.

Note that we only allow costs 0 and 1. This corresponds to the usual notion of *relative* term rewriting, where the TRS \mathcal{R} is partitioned into two sets \mathcal{R}_1 and \mathcal{R}_0 and the complexity of a rewrite sequence is defined to be the number of $\rightarrow_{\mathcal{R}_1}$ -steps (i.e., the rules in \mathcal{R}_1 have cost 1 and the rules in \mathcal{R}_0 have cost 0).

Example 7.2 (TRS). The following rules constitute the TRS $\mathcal{R}_{\text{contains}}$ which checks if a list of natural numbers (encoded as terms) contains **zero**.

$$\begin{array}{lll} \alpha_0 : & \text{contains}(\text{nil}) & \rightarrow \text{false} \\ \alpha_1 : & \text{contains}(\text{cons}(\text{succ}(x), xs)) & \rightarrow \text{contains}(xs) \\ \alpha_2 : & \text{contains}(\text{cons}(\text{zero}, xs)) & \rightarrow \text{true} \end{array}$$

The transition relation of TRSs is the *term rewrite relation*.

Definition 7.3 (Term Rewrite Relation). Let \mathcal{R} be a TRS. We have $s \xrightarrow{\ell} t$ if there is a context C , a rule $\ell \xrightarrow{\ell} r$, and a substitution σ such that $C[\ell\sigma] = s$ and $C[r\sigma] = t$. The rewrite step $s \xrightarrow{\ell} t$ is *innermost* ($s \xrightarrow{i} t$) if all proper subterms of $\ell\sigma$ are normal forms.

We write $s \xrightarrow{\ell} t$ (resp. $s \xrightarrow{i} t$) if $\mathcal{R} = \{\ell\}$. Furthermore, we write $s \xrightarrow{\ell, \pi} t$ if π is the unique position of \square in C .

Note that $\rightarrow_{\mathcal{R}}$ and $\xrightarrow{i}_{\mathcal{R}}$ are weighted relations, cf. Definition 2.10. In contrast to the usual relative term rewrite relation (which is defined as $\rightarrow_{\mathcal{R}_0}^* \circ \rightarrow_{\mathcal{R}_1} \circ \rightarrow_{\mathcal{R}_0}^*$ where \mathcal{R}_0 contains all rules with cost 0 and \mathcal{R}_1 contains all rules with cost \mathcal{R}_1), Definition 7.3 also allows rewrite sequences with cost 0. However, since such rewrite sequences are clearly irrelevant for worst-case complexity (since the image of the runtime complexity function does not contain negative numbers and thus 0 is a trivial lower bound), all results from Part III immediately carry over to relative term rewriting.

Example 7.4 (Term Rewrite Relation). We have

$$\begin{array}{ccc} \text{contains}(\text{cons}(\text{succ}(\text{zero}), \text{cons}(\text{zero}, \text{nil}))) & \xrightarrow{\alpha_1} & \text{contains}(\text{cons}(\text{zero}, \text{nil})) \\ & \xrightarrow{\alpha_2} & \text{true}. \end{array}$$

Since both of these rewrite steps are innermost, we also have

$$\text{contains}(\text{cons}(\text{succ}(\text{zero}), \text{cons}(\text{zero}, \text{nil}))) \xrightarrow{i}^* \text{true}.$$

By essentially replacing matching with unification in the definition of the term rewrite relation $\rightarrow_{\mathcal{R}}$, we obtain the *reduction relation*. Reducing a term t allows us to systematically explore how instances $t\sigma$ of t can be reduced with $\rightarrow_{\mathcal{R}}$.

Definition 7.5 (Narrowing Relation). Let \mathcal{R} be a TRS. We have $s \xrightarrow{\sigma} t$ if there is a position $\pi \in \text{pos}(s)$ with $s|_{\pi} \notin \mathcal{V}$ and a (variable-renamed) rule $\ell \xrightarrow{\ell} r \in \mathcal{R}$ such that $\sigma = \text{mgu}(\ell, s|_{\pi})$ and $s[r]_{\pi}\sigma = t$. We write $s \xrightarrow{\sigma}^* t$ if $s \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} t$ and $\sigma = \sigma_1 \diamond \dots \diamond \sigma_n$. Moreover, we write $s \rightsquigarrow t$ if the substitution σ is irrelevant.

Example 7.6. For the TRS from Example 7.2, we have, e.g.,

$$\begin{array}{ccc} \text{contains}(xs) & \xrightarrow{\{xs/\text{nil}\}} \mathcal{R}_{\text{contains}} & \text{false}, \\ \text{contains}(xs) & \xrightarrow{\{xs/\text{cons}(\text{succ}(x), xs')\}} \mathcal{R}_{\text{contains}} & \text{contains}(xs'), \text{ and} \\ \text{contains}(xs) & \xrightarrow{\{xs/\text{cons}(\text{zero}, xs')\}} \mathcal{R}_{\text{contains}} & \text{true}. \end{array}$$

The function symbols of a TRS can naturally be partitioned into those function symbols which represent algorithms (like `contains`) and those which represent

data (like `succ` and `zero`). This partitioning is captured by the notions of “defined symbols” and “constructors”.

Definition 7.7 (Constructors and Defined Symbols). Let \mathcal{R} be a TRS over Σ . We call $\Sigma_d(\mathcal{R}) = \{\text{root}(\alpha) \mid \alpha \in \mathcal{R}\}$ the *defined symbols* and $\Sigma_c(\mathcal{R}) = \Sigma \setminus \Sigma_d(\mathcal{R})$ the *constructors* of \mathcal{R} . $\Sigma_c(t)$ resp. $\Sigma_d(t)$ is the set of all defined symbols resp. constructors occurring in t .

We lift Σ_c and Σ_d to rules and sets of terms like Σ .

Example 7.8 (Constructors and Defined Symbols). If

$$\Sigma = \{\text{contains}, \text{zero}, \text{succ}, \text{nil}, \text{cons}, \text{true}, \text{false}\},$$

then the defined symbols of $\mathcal{R}_{\text{contains}}$ are $\Sigma_d(\mathcal{R}_{\text{contains}}) = \{\text{contains}\}$ and its constructors are $\Sigma_c(\mathcal{R}_{\text{contains}}) = \{\text{zero}, \text{succ}, \text{nil}, \text{cons}, \text{true}, \text{false}\}$.

To analyze the complexity of a TRS, we sometimes proceed bottom up, i.e., we analyze auxiliary functions before their callers. As in Section 5.3, we use *call graphs* to formalize the notion “bottom up”.

Definition 7.9 (Call Graph). Let \mathcal{R} be a TRS. The node set of the *call graph* of \mathcal{R} is $\Sigma_d(\mathcal{R})$ and its edge set is

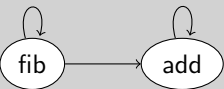
$$\{(\mathbf{f}, \mathbf{g}) \mid \ell \rightarrow r \in \mathcal{R}, \mathbf{f} \in \Sigma_d(\ell), \mathbf{g} \in \Sigma_d(r)\}.$$

We write $\mathbf{f} \sqsubset \mathbf{g}$ if there is a non-empty path from \mathbf{f} to \mathbf{g} in the call graph of \mathcal{R} and $\mathbf{f} \sqsupseteq \mathbf{g}$ if $\mathbf{f} \sqsubset \mathbf{g}$ or $\mathbf{f} = \mathbf{g}$.

So the call graph of $\mathcal{R}_{\text{contains}}$ consists of a single node for `contains` with a self loop. The following example shows a slightly more interesting call graph.

Example 7.10 (Call Graph of \mathcal{R}_{fib}). Consider the following TRS \mathcal{R}_{fib} which computes the Fibonacci numbers.

$$\begin{aligned} \beta_0 : & \quad \text{add}(\text{zero}, y) \rightarrow y \\ \beta_1 : & \quad \text{add}(\text{succ}(x), y) \rightarrow \text{add}(x, \text{succ}(y)) \\ \beta_2 : & \quad \text{fib}(\text{zero}) \rightarrow \text{zero} \\ \beta_3 : & \quad \text{fib}(\text{succ}(\text{zero})) \rightarrow \text{succ}(\text{zero}) \\ \beta_4 : & \quad \text{fib}(\text{succ}(\text{succ}(x))) \rightarrow \text{add}(\text{fib}(\text{succ}(x)), \text{fib}(x)) \end{aligned}$$

Its call graph is . Hence, a bottom-up analysis of \mathcal{R}_{fib} would analyze `add` before `fib`.

CHAPTER 7. PRELIMINARIES

When analyzing the complexity of a program, then one is usually interested in the cost of evaluating a function (i.e., a defined symbol) applied to data (i.e., terms built from constructors). Hence, we exclude start terms like, e.g.,

$$\text{cons}(\text{contains}(\dots), \text{cons}(\text{contains}(\dots), \dots)) \quad (7.1)$$

that correspond to *several* evaluations of a function. More precisely, we restrict our attention to rewrite sequences starting with *basic terms*.

Definition 7.11 (Basic Term [79]). Let \mathcal{R} be a TRS over Σ . A term $f(\mathbf{t})$ is *basic* w.r.t. \mathcal{R} if $f \in \Sigma_d(\mathcal{R})$ and $\mathbf{t} \subseteq \mathcal{T}(\Sigma_c(\mathcal{R}), \mathcal{V})$. The set of all *basic terms* w.r.t. \mathcal{R} is $\mathcal{T}_{\text{basic}}(\mathcal{R})$. If $\ell \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ for all $\ell \xrightarrow{\mathcal{R}} r \in \mathcal{R}$, then \mathcal{R} is a *constructor system*.

If one also considers start terms like (7.1) (i.e., if one considers arbitrary start terms), then the resulting notion of complexity is known as *derivational complexity*, whereas the techniques presented in this thesis analyze the *runtime complexity* [79] of a TRS. However, since the runtime complexity of a TRS is a lower bound on its derivational complexity, the techniques for the inference of lower bounds on the runtime complexity of TRSs from Chapter 8 and Chapter 9 immediately apply to derivational complexity, too.

So we are interested in the cost of $\rightarrow_{\mathcal{R}}$ -sequences starting with basic terms, i.e., we have fixed the set of start terms and the weighted relation we are interested in. The only missing piece in order to define the *canonical complexity problem* of a TRS is a suitable size measure. For term rewriting, the established size measure is *term size*.

Definition 7.12 (Term Size). Let Σ be a signature and let \mathcal{V} be a set of variables. We define $\|\cdot\|_t : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathbb{N}$ as follows:

$$\|t\|_t = \begin{cases} 1 + \sum \|\mathbf{t}\|_t & \text{if } t = f(\mathbf{t}), f \in \Sigma \\ 1 & \text{if } t \in \mathcal{V} \end{cases}$$

Now we can define the canonical complexity problem of a TRS.

Definition 7.13 (Canonical Complexity Problem of TRSs). Let \mathcal{R} be a TRS. The canonical complexity problem of \mathcal{R} is

$$cp(\mathcal{R}) = (\mathcal{T}_{\text{basic}}(\mathcal{R}), \rightarrow_{\mathcal{R}}, \|\cdot\|_t).$$

The *innermost* canonical complexity problem of \mathcal{R} is

$$cp_i(\mathcal{R}) = (\mathcal{T}_{\text{basic}}(\mathcal{R}), \rightarrow_i, \|\cdot\|_t).$$

Thus, the canonical complexity problem of $\mathcal{R}_{\text{contains}}$ is

$$cp(\mathcal{R}_{\text{contains}}) = (\mathcal{T}_{\text{basic}}(\mathcal{R}_{\text{contains}}), \rightarrow_{\mathcal{R}_{\text{contains}}}, \|\cdot\|_t).$$

Turing Machines Several proofs in Part III rely on reductions from *Turing machines*.

Definition 7.14 (Turing Machine). A triple $\mathcal{M} = (Q, \Gamma, \delta)$ where Q is a finite set of *states*, Γ is the finite *tape alphabet*, $\square \in \Gamma$ is the *blank symbol*, and $\delta : (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\})$ is the *transition function* is called a *Turing machine*. A *configuration* of \mathcal{M} has the form (q, w, a, w') with $q \in Q$, $w, w' \in \Gamma^\omega$, and $a \in \Gamma$. The transition function δ induces a transition relation $\rightarrow_{\mathcal{M}}$ on configurations where $(q_1, w_1, a_1, w'_1) \rightarrow_{\mathcal{M}} (q_2, w_2, a_2, w'_2)$ if either

- $w_1 = a_2.w_2, w'_2 = b.w'_1$, and $\delta(q_1, a_1) = (q_2, b, L)$ or
- $w_2 = b.w_1, w'_1 = a_2.w'_2$, and $\delta(q_1, a_1) = (q_2, b, R)$.

The meaning of a configuration (q, w, a, w') is that q is the current state, the symbol at the current position of the tape is a , the symbols right of the current position are described by the infinite word w' , and the symbols left of it are described by the infinite word w . To ease the formulation, if $w = b.\bar{w}$ then this means that b is the symbol directly left of the current position, i.e., w is the word obtained when reading the symbols on the tape from right to left.

Lower Bounds for Term Rewriting by Loop Detection

As mentioned in Chapter 6, many non-termination techniques for TRSs try to detect *loops*. In this section, we show how to adapt such techniques in order to infer lower complexity bounds.

In Section 8.1 we adapt the notion of loops to prove linear lower bounds. Section 8.2 extends this approach to exponential bounds. Thus, the presented technique is particularly suitable to detect families of runs with exponential complexity in programs operating on tree-shaped data structures, cf. Section 1.2. Such families of program runs witness bugs or denial of service vulnerabilities in many cases.

Since our technique from Section 8.1 can prove linear lower bounds for almost all TRSs which have at least linear complexity, the question whether $rc_{cp}(\mathcal{R})(n) \in \Omega(n)$ is decidable arises. Consequently, this question is investigated in Section 8.3. Finally, we adapt our technique to innermost rewriting in Section 8.4, discuss related work in Section 8.5, and conclude in Section 8.6. An extensive experimental evaluation of the presented technique can be found in Chapter 12.

8.1 Loop Detection for Linear Bounds

A *loop* is a reduction sequence $s \xrightarrow{\mathcal{R}}^+ C[s\sigma]$ for some context C and some substitution σ . Each loop gives rise to a non-terminating reduction

$$s \xrightarrow{\mathcal{R}}^+ C[s\sigma] \xrightarrow{\mathcal{R}}^+ C[C\sigma[s\sigma^2]] \xrightarrow{\mathcal{R}}^+ \dots$$

Thus, if s is basic and $\mathcal{R} > 0$, then the existence of a loop proves $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(\omega)$. The idea of the technique in this section is to detect rewrite sequences which are similar to loops, but at some position π of s , a context D of constant size is removed (i.e., we want to detect so-called *decreasing loops*). Moreover, we are not interested in rewrite sequences with cost 0, i.e., each “iteration” of the decreasing loop must have cost $\mathcal{R} > 0$. Hence, we want to find infinite families of rewrite sequences of the form

$$\begin{array}{ccc} s[D^n[t]]_\pi & \xrightarrow{\mathcal{R}}^+ & C_1[s[D^{n-1}[t]]_\pi \sigma] & \supseteq & s[D^{n-1}[t]]_\pi \sigma \\ & \xrightarrow{\mathcal{R}}^+ & C_2[s[D^{n-2}[t]]_\pi \sigma^2] & \supseteq & s[D^{n-2}[t]]_\pi \sigma^2 \\ & \xrightarrow{\mathcal{R}}^+ \circ \supseteq & \dots & \xrightarrow{\mathcal{R}}^+ \circ \supseteq & s[t]_\pi \sigma^n. \end{array}$$

Again, $s' \supseteq s$ means that s is a subterm of s' , cf. Definition 2.3. If there is a decreasing loop, then the runtime complexity of \mathcal{R} is at least linear, i.e., $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$. To find such families of rewrite sequences, we look for a rewrite step of the form $s[D[x]]_\pi \xrightarrow{\mathcal{R}} C[s[x]_\pi \sigma]$ for a variable x . Then the term $s[D^n[x]]_\pi$ starts a reduction of length n . This term is obtained by applying the substitution θ^n with $\theta = \{x/D[x]\}$ to $s[x]_\pi$.

Example 8.1 (Decreasing Loop). The rule $\alpha_1 \in \mathcal{R}_{\text{contains}}$ removes the context $D = \text{cons}(\text{succ}(x), \square)$ around the variable xs in every rewrite step. The size of this context is constant. Thus, if one starts with a context D^n of size $3 \cdot n$, then one can perform n rewrite steps to remove this context, which shows $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$.

Note that the variable xs occurs exactly once in the left-hand side

$$\ell = \text{contains}(\text{cons}(\text{succ}(x), xs)),$$

at position $\pi = 1.2$ (i.e., ℓ is *linear*). Moreover, this variable also appears in the right-hand side r at a position $\xi = 1$ that is above π (i.e., $\xi < \pi$). Thus, every rewrite step removes the context that is around xs in $\ell|_\xi = \text{cons}(\text{succ}(x), xs)$. Let $\bar{\ell}$ be the term that results from ℓ by replacing the subterm $\ell|_\xi$ by the variable xs , i.e., $\bar{\ell} = \ell[xs]_\xi = \text{contains}(xs)$. Moreover, let θ be the substitution that replaces xs by $\ell|_\xi$ again (i.e., $\theta = \{xs/\text{cons}(\text{succ}(x), xs)\}$). Suppose that $\bar{\ell}\sigma = r$ for some matcher σ that does not instantiate the variables in $\ell|_\xi$ (i.e., σ does not *interfere* with θ). A rewrite rule $\ell \rightarrow r$ satisfying these conditions is called a *decreasing loop*. In our example, $\bar{\ell} = \text{contains}(xs)$ matches the right-hand side $r = \text{contains}(xs)$ with the matcher $\sigma = \text{id}$, i.e., with the identical

8.1. LOOP DETECTION FOR LINEAR BOUNDS

substitution. Thus, rewriting $\bar{\ell}\theta = \ell$ results in an instance of $\bar{\ell}$ again (i.e., in $r = \bar{\ell}\sigma$). Hence, every decreasing loop results in a rewrite sequence of linear length: if one starts with $\bar{\ell}\theta^n$, then this rewrite step can be repeated n times, removing one application of θ in each step. More precisely, we have

$$\bar{\ell}\theta^n = \ell\theta^{n-1} \xrightarrow{\mathcal{R}} r\theta^{n-1} = \bar{\ell}\sigma\theta^{n-1} = \bar{\ell}\theta^{n-1}\sigma' \text{ for some substitution } \sigma',$$

as σ does not interfere with θ . Hence, in our example, the term $\bar{\ell}\theta^n = \text{contains}(D^n[xs])$ with $D = \text{cons}(\text{succ}(x), \square)$ starts a reduction of length n .

Based on this idea, three improvements enhance the applicability of the resulting technique: First, it suffices to require that $\bar{\ell}$ matches a *subterm* r of the right-hand side (i.e., the right-hand side may have the form $C[r]$ for some context C). Second, instead of creating $\bar{\ell}$ by replacing one subterm $\ell|_{\xi}$ of ℓ with a variable $x \in \mathcal{V}(\ell|_{\xi})$, we can replace *several* subterms $\ell|_{\xi_1}, \dots, \ell|_{\xi_m}$ with variables $x_i \in \mathcal{V}(\ell|_{\xi_i})$. Here, ξ_1, \dots, ξ_m must be independent positions, i.e., we have $\xi_i \not\leq \xi_j$ and $\xi_j \not\leq \xi_i$ whenever $i \neq j$. The structure of ℓ , $\bar{\ell}$, and r is illustrated in Figure 8.1. Here, a dashed arrow labeled with a substitution like $\bar{\ell} \xrightarrow{\theta} \ell$ means that applying the substitution θ to $\bar{\ell}$ results in ℓ . Third, instead of checking whether a single rule $\ell \xrightarrow{\mathcal{R}} C[r]$ is a decreasing loop, we can also consider rewrite sequences $\ell \xrightarrow{\mathcal{R}}^+ C[r]$. To find such rewrite sequences, we repeatedly narrow the right-hand sides of those rules whose left-hand sides are basic. This leads to Definition 8.2. (Note that here, (a) implies that ξ_1, \dots, ξ_m are independent positions, since the $r|_{\xi_i}$ are pairwise different variables.)

Definition 8.2 (Decreasing Loop). Let $\ell \xrightarrow{\mathcal{R}}^+ C[r]$ for some $\mathcal{R} > 0$, some linear basic term ℓ , and some $r \notin \mathcal{V}$. We call $\ell \xrightarrow{\mathcal{R}}^+ C[r]$ a *decreasing loop* if there are pairwise different variables x_1, \dots, x_m (with $m \geq 0$) and positions π_1, \dots, π_m with $x_i = \ell|_{\pi_i}$ for all $i \in \{1, \dots, m\}$ such that:

- (a) for each x_i , there is a $\xi_i < \pi_i$ such that $r|_{\xi_i} = x_i$
- (b) there is a substitution σ with $\bar{\ell}\sigma = r$ for $\bar{\ell} = \ell[x_1]_{\xi_1} \dots [x_m]_{\xi_m}$

We call σ the *result substitution*, $\theta = \{x_i/\ell|_{\xi_i} \mid i \in \{1, \dots, m\}\}$ the *pumping substitution*, and ξ_1, \dots, ξ_m the *abstracted positions* of the decreasing loop.

The following TRS has a decreasing loop with a non-trivial result substitution.

Example 8.3 (Result Substitution). Consider the TRS \mathcal{R}_{add} consisting of the add rules from Example 7.10.

$$\text{add}(\text{zero}, y) \rightarrow y \qquad \text{add}(\text{succ}(x), y) \rightarrow \text{add}(x, \text{succ}(y))$$

The second rule is a decreasing loop with $\bar{\ell} = \text{add}(x, y)$, $\xi = 1$, $\theta = \{x/\text{succ}(x)\}$, and $\sigma = \{y/\text{succ}(y)\}$. Indeed, we have $\bar{\ell}\theta^n \rightarrow_{\mathcal{R}_{\text{add}}}^n \bar{\ell}\sigma^n$.

CHAPTER 8. LOOP DETECTION

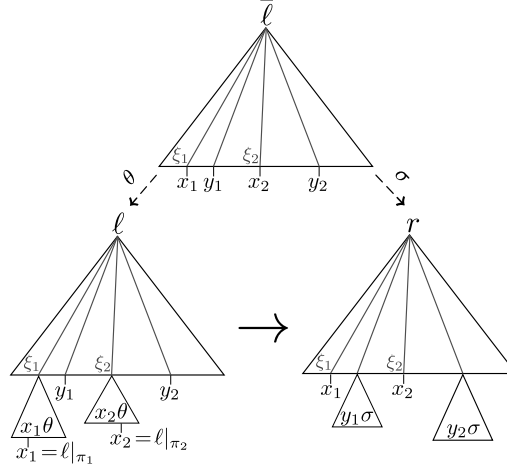


Figure 8.1: Decreasing Loop

Example 8.4 shows that requiring linearity in Definition 8.2 is crucial to ensure that the existence of a decreasing loop implies $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$.

Example 8.4 (Linearity). To see why we require linearity of ℓ , consider

$$\mathcal{R} = \{f(\text{succ}(x), x) \rightarrow f(x, x)\}.$$

If non-linear terms ℓ were allowed by Definition 8.2, then \mathcal{R} 's only rule $f(\text{succ}(x), x) \rightarrow f(x, x)$ would be a decreasing loop with the abstracted position $\xi = 1$. Thus, we would falsely conclude a linear lower runtime bound although $\text{rc}_{cp(\mathcal{R})}$ is constant.

Finally, Example 8.5 shows why we require that the right-hand side of a decreasing loop is not a variable.

Example 8.5 (Non-Variable Right-Hand Sides). The requirement $r \notin \mathcal{V}$ in Definition 8.2 is needed to ensure that θ instantiates variables by constructor terms. Otherwise, for the TRS $\mathcal{R} = \{f(x) \rightarrow x\}$ we would falsely detect a decreasing loop although $\text{rc}_{cp(\mathcal{R})}$ is constant. The reason is that for $\bar{\ell} = x$ and $\theta = \{x/f(x)\}$, $\bar{\ell}\theta^n$ starts a rewrite sequence of length n , but $\bar{\ell}\theta^n$ is not a basic term.

Theorem 8.6 states that any decreasing loop gives rise to a linear lower bound.

Theorem 8.6 (Linear Lower Bounds by Loop Detection). *If a TRS \mathcal{R} has a decreasing loop, then we have $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$.*

Proof. For all $n \in \mathbb{N}$ and any substitution δ , we prove $\bar{\ell}\theta^n\delta \xrightarrow{\mathcal{R}}^+ \circ \supseteq \bar{\ell}\theta^{n-1}\delta'$

8.1. LOOP DETECTION FOR LINEAR BOUNDS

for some substitution δ' . Thus, these rewrite steps can be repeated n times. We have

$$\begin{aligned}\bar{\ell}\theta^n\delta &= \ell\theta^{n-1}\delta \xrightarrow{\mathcal{R}}^+ C[r]\theta^{n-1}\delta \supseteq r\theta^{n-1}\delta \\ &= \bar{\ell}\sigma\theta^{n-1}\delta \stackrel{(\star)}{=} \bar{\ell}\theta^{n-1}(\sigma \diamond \theta^{n-1})|_{\text{dom}(\sigma)}\delta = \bar{\ell}\theta^{n-1}\delta'\end{aligned}$$

for the substitution $\delta' = (\sigma \diamond \theta^{n-1})|_{\text{dom}(\sigma)} \diamond \delta$. The step marked with (\star) holds since σ does not instantiate variables in the domain or range of θ . To see why $\text{dom}(\sigma)$ is disjoint from

$$\text{dom}(\theta) = \{x_1, \dots, x_m\}, \quad (8.1)$$

note that $x_i\sigma \neq x_i$ would mean $\bar{\ell}|_{\xi_i}\sigma \neq r|_{\xi_i}$. As $\bar{\ell}|_{\xi_i}\sigma = \bar{\ell}\sigma|_{\xi_i}$, this would imply $\bar{\ell}\sigma|_{\xi_i} \neq r|_{\xi_i}$, which contradicts $\bar{\ell}\sigma = r$. To see why $\text{dom}(\sigma)$ is disjoint from $\text{rng}(\theta)$, note that by definition of θ we have

$$\mathcal{V}(\text{rng}(\theta)) = \bigcup_{i=1}^m \mathcal{V}(\ell|_{\xi_i}). \quad (8.2)$$

Since ℓ is linear, by definition of $\bar{\ell}$ we have

$$\begin{aligned}\mathcal{V}(\bar{\ell}) &= (\mathcal{V}(\ell) \setminus \bigcup_{i=1}^m \mathcal{V}(\ell|_{\xi_i})) \cup \{x_1, \dots, x_m\} \\ &= (\mathcal{V}(\ell) \setminus \mathcal{V}(\text{rng}(\theta))) \cup \text{dom}(\theta) \quad \text{by (8.1) and (8.2)}.\end{aligned} \quad (8.3)$$

Clearly, the substitution σ that matches $\bar{\ell}$ to r can be chosen such that its domain only includes variables occurring in $\bar{\ell}$. Thus, (8.3) implies

$$\text{dom}(\sigma) \subseteq (\mathcal{V}(\ell) \setminus \mathcal{V}(\text{rng}(\theta))) \cup \text{dom}(\theta). \quad (8.4)$$

Since $\text{dom}(\sigma)$ and $\text{dom}(\theta)$ are disjoint, (8.4) implies

$$\text{dom}(\sigma) \subseteq \mathcal{V}(\ell) \setminus \mathcal{V}(\text{rng}(\theta)).$$

Hence σ also does not instantiate any variables occurring in the range of θ .

Thus, for each $n \in \mathbb{N}$, there is a rewrite sequence with cost $\mathcal{K} \cdot n$ starting with $\bar{\ell}\theta^n$. This term is basic, since the range of θ only contains terms of the form $\ell|_{\xi_i}$. Each $\ell|_{\xi_i}$ is a constructor term, since ξ_i cannot be the root position, due to $r \notin \mathcal{V}$. By construction, θ does not duplicate variables, as ℓ and thus $\bar{\ell}|_{\xi_1}, \dots, \bar{\ell}|_{\xi_m}$ only contain each x_i once. Therefore, we have $\|\bar{\ell}\theta^n\|_t \in \mathcal{O}(n)$. If $\theta \neq \text{id}$, then $\|\bar{\ell}\theta^n\|_t$ is strictly monotonically increasing in n and we obtain $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ by Lemma 4.41. Otherwise, $\ell \xrightarrow{\mathcal{R}}^+ C[r]$ is a loop, which implies $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(\omega)$, since ℓ is basic and $\mathcal{K} > 0$. \square

Of course, the linear bound recognized by Theorem 8.6 is just a lower bound. In particular, as in the proof of Theorem 8.6, if $\theta = \text{id}$ (i.e., $\bar{\ell} = \ell$), then we have the loop

$$\ell \xrightarrow{\mathcal{R}}^+ C[r] \supseteq r = \bar{\ell}\sigma = \ell\sigma.$$

CHAPTER 8. LOOP DETECTION

Thus, there is even an infinite lower bound (i.e., a basic term starts an infinite reduction). Hence, loops where ℓ is basic are indeed special cases of decreasing loops.

Corollary 8.7 (Infinite Lower Bounds by Loop Detection). *If there is a decreasing loop for a TRS \mathcal{R} whose pumping substitution is id , then we have $\text{rc}_{cp}(\mathcal{R})(n) \in \Omega(\omega)$.*

Proof. The corollary holds since we have $\ell \xrightarrow[\mathcal{R}]{\ell^+} C[r] \supseteq r = \bar{\ell}\sigma = \ell\sigma$. □

8.2 Loop Detection for Exponential Bounds

We now adapt the criterion of Theorem 8.6 in order to detect exponential lower bounds. Theorem 8.6 characterizes TRSs where a context around a variable x is removed in each rewrite step and the same rewrite rule is again applicable to the right-hand side. We now consider reduction sequences $\ell \xrightarrow{\mathcal{R}}^+ r$ such that $r = C_1[r_1]_{\iota_1} = C_2[r_2]_{\iota_2}$ for independent positions ι_1 and ι_2 where both $\ell \xrightarrow{\mathcal{R}}^+ C_1[r_1]_{\iota_1}$ and $\ell \xrightarrow{\mathcal{R}}^+ C_2[r_2]_{\iota_2}$ are decreasing loops. Then each rewrite step removes some context, but at the same time it creates *two* redexes on the right-hand side where the same rewrite rule is applicable again. This give rise to an (asymptotic) exponential lower bound.

Example 8.8 (Exponential Bound for Fibonacci Numbers). Reconsider the TRS \mathcal{R}_{fib} from Example 7.10, whose runtime complexity is exponential. In the rule β_4 , there are *two* recursive calls on the right-hand side where each recursive call gives rise to a decreasing loop. More precisely,

$$\text{fib}(\text{succ}(\text{succ}(x))) \xrightarrow{\mathcal{R}_{\text{fib}}} C_1[\text{fib}(\text{succ}(x))]$$

is a decreasing loop with $\bar{\ell}_1 = \text{fib}(\text{succ}(x))$, pumping substitution θ_1 with $x\theta_1 = \text{succ}(x)$, and result substitution $\sigma_1 = \text{id}$. On the other hand,

$$\text{fib}(\text{succ}(\text{succ}(x))) \xrightarrow{\mathcal{R}_{\text{fib}}} C_2[\text{fib}(x)]$$

is a decreasing loop with $\bar{\ell}_2 = \text{fib}(x)$, pumping substitution θ_2 with $x\theta_2 = \text{succ}(\text{succ}(x))$, and result substitution $\sigma_2 = \text{id}$.

The two decreasing loops give rise to an exponential lower bound. We have $\bar{\ell}_1\theta_1^n\theta_2^n \xrightarrow{\mathcal{R}_{\text{fib}}} \text{add}(\bar{\ell}_1\theta_1^{n-1}\theta_2^n, \bar{\ell}_2\theta_1^{n-1}\theta_2^n)$. Note that the pumping substitutions θ_1 and θ_2 *commute*, i.e., $\theta_1 \diamond \theta_2 = \theta_2 \diamond \theta_1$. Thus, for the subterm in the second argument of *add*, we have $\bar{\ell}_2\theta_1^{n-1}\theta_2^n = \bar{\ell}_2\theta_2\theta_1^{n-1}\theta_2^{n-1}$. Hence after each application of the recursive *fib*-rule to $\bar{\ell}_i\theta_1^n\theta_2^n$ we obtain *two* new similar terms where one application of θ_i has been removed, but $2 \cdot n - 1$ applications of pumping substitutions remain. Since the pumping substitutions commute, the next reduction step again yields *two* new similar terms with $2 \cdot n - 2$ remaining applications of pumping substitutions. Thus, rewriting $\bar{\ell}_1\theta_1^n\theta_2^n$ yields a binary “tree” of reductions which is complete up to height n . Hence, the resulting rewrite sequence has an exponential length.

The commutation of the pumping substitutions is indeed crucial. Otherwise, it would not be sound to infer an exponential lower bound from the existence of two independent decreasing loops, as the following example shows.

Example 8.9 (Commutation). Without requiring commutation of the pumping substitutions, we would obtain incorrect exponential bounds for typical algorithms that traverse trees (the TRS $\mathcal{R}_{\text{traverse}}$ below represents the simplest

possible tree traversal algorithm).

$$\begin{aligned} \text{traverse}(\text{leaf}) &\rightarrow \text{leaf} \\ \text{traverse}(\text{tree}(xs, ys)) &\rightarrow \text{tree}(\text{traverse}(xs), \text{traverse}(ys)) \end{aligned}$$

Each recursive call in the right-hand side of the last rule gives rise to a decreasing loop. For

$$\text{traverse}(\text{tree}(xs, ys)) \xrightarrow{1}_{\mathcal{R}_{\text{traverse}}} C_1[\text{traverse}(xs)]$$

we have $\bar{\ell}_1 = \text{traverse}(xs)$ with the result substitution $\sigma_1 = \text{id}$ and the pumping substitution $\theta_1 = \{xs/\text{tree}(xs, ys)\}$. The decreasing loop

$$\text{traverse}(\text{tree}(xs, ys)) \xrightarrow{1}_{\mathcal{R}_{\text{traverse}}} C_2[\text{traverse}(ys)]$$

has $\bar{\ell}_2 = \text{traverse}(ys)$ with $\sigma_2 = \text{id}$ and $\theta_2 = \{ys/\text{tree}(xs, ys)\}$. However, this does *not* imply an exponential lower bound. The reason is that θ_1 and θ_2 do not commute. Thus, we have $\bar{\ell}_1\theta_1^n\theta_2^n \xrightarrow{1}_{\mathcal{R}_{\text{traverse}}} \text{tree}(\bar{\ell}_1\theta_1^{n-1}\theta_2^n, \bar{\ell}_2\theta_1^{n-1}\theta_2^n)$, but instead of $\bar{\ell}_2\theta_1^{n-1}\theta_2^n = \bar{\ell}_2\theta_2\theta_1^{n-1}\theta_2^{n-1}$ as in Example 8.8, we have $\bar{\ell}_2\theta_1^{n-1}\theta_2^n = \bar{\ell}_2\theta_2^n$. Hence, the resulting runtime is only linear.

The following example shows that in addition to the commutation property of the pumping substitutions, the result substitution of one decreasing loop must not interfere with the pumping substitution of the other loop.

Example 8.10 (Interference of Result and Pumping Substitution). The rule $\ell \rightarrow r$ with $\ell = f(\text{succ}(x), \text{succ}(y))$ and $r = c(f(x, \text{succ}(\text{zero})), f(x, y))$ gives rise to two decreasing loops. The first one is $\ell \rightarrow C_1[f(x, \text{succ}(\text{zero}))]$ with $\bar{\ell}_1 = f(x, \text{succ}(y))$, $r_1 = f(x, \text{succ}(\text{zero}))$, $\theta_1 = \{x/\text{succ}(x)\}$, and $\sigma_1 = \{y/\text{zero}\}$. The second one is $\ell \rightarrow C_2[f(x, y)]$ with $\bar{\ell}_2 = f(x, y)$, $r_2 = f(x, y)$, $\theta_2 = \{x/\text{succ}(x), y/\text{succ}(y)\}$, and $\sigma_2 = \text{id}$. However, this does *not* imply an exponential lower bound. The reason is that the domain of the result substitution σ_1 contains the variable y which also occurs in the domain and range of θ_2 . Hence, we have:

$$\begin{array}{l|l} f(x, \text{succ}(y))\theta_1^n\theta_2^n & \bar{\ell}_1\theta_1^n\theta_2^n \\ = f(\text{succ}(x), \text{succ}(y))\theta_1^{n-1}\theta_2^n & = \ell\theta_1^{n-1}\theta_2^n \\ \rightarrow c(f(x, \text{succ}(\text{zero})), f(x, y))\theta_1^{n-1}\theta_2^n & \rightarrow C_1[r_1]\theta_1^{n-1}\theta_2^n \\ \supseteq f(x, \text{succ}(\text{zero}))\theta_1^{n-1}\theta_2^n & \supseteq \bar{\ell}_1\sigma_1\theta_1^{n-1}\theta_2^n \\ = f(\text{succ}(x), \text{succ}(\text{zero}))\theta_1^{n-2}\theta_2^n & = \ell\sigma_1\theta_1^{n-2}\theta_2^n \\ \rightarrow c(f(x, \text{succ}(\text{zero})), \underline{f(x, \text{zero})})\theta_1^{n-2}\theta_2^n & \rightarrow C_2[r_2]\sigma_1\theta_1^{n-2}\theta_2^n \end{array}$$

To obtain the desired rewrite sequence of exponential length, each f -term in the resulting term should again create a binary “tree” of reductions which is complete up to height $n - 2$ (as there are still at least $n - 2$ applications of each pumping substitution). However, the underlined subterm $f(x, \text{zero})$

8.2. LOOP DETECTION FOR EXPONENTIAL BOUNDS

(i.e., $r_2\sigma_1$) is a normal form. The problem is that the result substitution $\sigma_1 = \{y/\text{zero}\}$ was applied in the first reduction step and this prevents the subsequent use of $\theta_2 = \{y/\text{succ}(y)\}$ in order to turn the subterm $f(x, y)$ of the right-hand side into a redex again.

In general, after one rule application one obtains the terms $\bar{\ell}_1\sigma_1\theta_1^{n-1}\theta_2^n$ and $\bar{\ell}_2\sigma_2\theta_2\theta_1^{n-1}\theta_2^{n-1} = \bar{\ell}_2\sigma_2\theta_1^{n-1}\theta_2^n$, which are “similar” to the start term $\bar{\ell}_1\theta_1^n\theta_2^n$ up to the result substitutions σ_1 and σ_2 . Therefore, one has to require that the result substitutions do not interfere with the pumping substitutions. Then these result substitutions do not prevent the desired exponentially long rewrite sequence.

The following definition introduces the concept of *compatible* decreasing loops. Two decreasing loops are compatible if (a) they result from the same rewrite sequence, (b) they operate on independent positions of the right-hand side, (c) the result substitution of each loop does not interfere with the pumping substitution of the other loop, and (d) their pumping substitutions commute.

Definition 8.11 (Compatible Decreasing Loops). Let $\ell \xrightarrow{\mathcal{R}}^+ C[r]_\iota$ and $\ell \xrightarrow{\mathcal{R}}^+ C'[r']_{\iota'}$ be decreasing loops with pumping substitutions θ resp. θ' , result substitutions σ resp. σ' , and abstracted positions ξ_1, \dots, ξ_m resp. $\xi'_1, \dots, \xi'_{m'}$. We call $\ell \xrightarrow{\mathcal{R}}^+ C[r]_\iota$ and $\ell \xrightarrow{\mathcal{R}}^+ C'[r']_{\iota'}$ *compatible* if

- (a) $C[r]_\iota = C'[r']_{\iota'}$
- (b) ι and ι' are independent positions
- (c) $\text{dom}(\sigma) \cap \mathcal{V}(\text{rng}(\theta')) = \text{dom}(\sigma') \cap \mathcal{V}(\text{rng}(\theta)) = \emptyset$
- (d) $\theta \diamond \theta' = \theta' \diamond \theta$

Theorem 8.12 shows that several compatible decreasing loops lead to exponential runtime.

Theorem 8.12 (Exponential Lower Bounds by Loop Detection). *If a TRS \mathcal{R} has $d \geq 2$ pairwise compatible decreasing loops, then $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(c^n)$ for some $c > 0$.*

Proof. For each $j \in \{1, \dots, d\}$, let θ_j be the pumping substitution and σ_j be the result substitution of the decreasing loop $\ell \xrightarrow{\mathcal{R}}^+ C_j[r_j]_{\iota_j}$ where $r = C_j[r_j]_{\iota_j}$. If ξ_1, \dots, ξ_m are the abstracted positions of the j^{th} decreasing loop and $x_i = r_j|_{\xi_i}$ for all $i \in \{1, \dots, m\}$, then let $\bar{\ell}_j = \ell[x_1]_{\xi_1} \dots [x_m]_{\xi_m}$. Thus, we have $\bar{\ell}_j\theta_j = \ell$ and $\bar{\ell}_j\sigma_j = r_j$.

For all $j \in \{1, \dots, d\}$, all $n_1, \dots, n_d \in \mathbb{N}$, and any substitution δ , $\bar{\ell}_j\theta_1^{n_1} \dots \theta_d^{n_d}\delta$ starts a reduction of asymptotic length $d^{\min(n_1, \dots, n_d)}$. To show this, we prove $\bar{\ell}_j\theta_1^{n_1} \dots \theta_d^{n_d}\delta \xrightarrow{\mathcal{R}}^+ q$ for some q such that for all $k \in \{1, \dots, d\}$, there is a

CHAPTER 8. LOOP DETECTION

substitution δ'_k with $q|_{\iota_k} = \bar{\ell}_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta'_k$.

Hence, q contains d terms of the form $\bar{\ell}_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta'_k$ at independent positions. We have

$$\begin{aligned} \bar{\ell}_j \theta_1^{n_1} \dots \theta_d^{n_d} \delta &\stackrel{(\dagger)}{=} \bar{\ell}_j \theta_j \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta = \\ \ell \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta &\rightarrow_{\mathcal{R}}^+ r \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta = q \end{aligned}$$

where (\dagger) holds as θ_j commutes with all θ_i by Definition 8.11 (d). For any $k \in \{1, \dots, d\}$,

$$\begin{aligned} q|_{\iota_k} &= r_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta \\ &= \bar{\ell}_k \sigma_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta \\ &\stackrel{(\star)}{=} \bar{\ell}_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} (\sigma_k \diamond \theta_1^{n_1} \diamond \dots \diamond \theta_j^{n_j-1} \diamond \dots \diamond \theta_d^{n_d})|_{\text{dom}(\sigma_k)} \delta \\ &= \bar{\ell}_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta'_k \end{aligned}$$

for the substitution $\delta'_k = (\sigma_k \diamond \theta_1^{n_1} \diamond \dots \diamond \theta_j^{n_j-1} \diamond \dots \diamond \theta_d^{n_d})|_{\text{dom}(\sigma_k)} \diamond \delta$.

For the step marked with (\star) , as in the proof of Theorem 8.6, σ_k does not instantiate variables in the domain or the range of θ_k . By Definition 8.11 (c) it also does not instantiate variables in the domain or the range of any other pumping substitution θ_i .

Since q contains $\bar{\ell}_k \theta_1^{n_1} \dots \theta_j^{n_j-1} \dots \theta_d^{n_d} \delta'_k$ at independent positions ι_k (for $k \in \{1, \dots, d\}$), this results in a d -ary tree of rewrite sequences with root $\bar{\ell}_j \theta_1^{n_1} \dots \theta_d^{n_d}$, $j \in \{1, \dots, d\}$, which is complete up to height $n \in \mathbb{N}$. The reason is that in the beginning, there are n substitutions θ_j for each $j \in \{1, \dots, d\}$ and each rewrite step removes one of them.

Hence, the tree has at least $\left\lfloor \frac{d^{n+1}-1}{d-1} \right\rfloor$ nodes. By Lemma A.2,¹ $\theta_1 \diamond \dots \diamond \theta_d$ does not duplicate variables. If $\theta_1 \diamond \dots \diamond \theta_d = \text{id}$, then we have $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(\omega)$ as in the proof of Theorem 8.6. Otherwise, $\|\bar{\ell}_j \theta_1^{n_1} \dots \theta_d^{n_d}\|_t$ is linear and strictly monotonically increasing in n and we get $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(c^n)$ for some $c > 1$ by Lemma 4.41. \square

¹Note that Lemma A.2 is only needed to avoid the additional requirement that $\theta \diamond \theta'$ does not duplicate variables in Definition 8.11, which is superfluous, as it is implied by the remaining properties of compatible decreasing loops. As its proof is lengthy and quite technical, it is only presented in the appendix of this thesis.

8.3 Incompleteness of Loop Detection

In practice, loop detection is an extremely powerful technique for the inference of linear lower bounds, cf. Chapter 12. In fact, we are not aware of an ordinary left-linear constructor TRS whose runtime complexity is at least linear, but which does not have a decreasing loop. Thus, it is natural to ask if *every* such TRS has a decreasing loop, i.e., if loop detection is a semi-decision procedure to check if $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ holds for a given ordinary left-linear constructor TRS \mathcal{R} .

However, as shown by the following theorem, this is not the case. The reason is that even for quite restricted classes of TRSs \mathcal{R} it is not semi-decidable if $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ holds. This can be proven by a reduction of the immortality problem for Turing machines (cf. Definition 7.14) to the problem of linear lower bounds, where immortality of Turing machines is known to be not semi-decidable [88]. A Turing machine is *mortal* if it terminates for every initial configuration, including configurations with infinitely many non-blank symbols on the tape.

Theorem 8.13 (Undecidability of Linear Bounds). *It is not semi-decidable if $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ holds for ordinary linear TRSs \mathcal{R} where $\ell, r \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ for all rewrite rules $\ell \rightarrow r \in \mathcal{R}$. Hence for this class of TRSs, loop detection is not complete for the inference of linear lower bounds.*

To prove Theorem 8.13, we need several auxiliary lemmas. In the following, we restrict ourselves to ordinary, linear, and *basic* TRSs \mathcal{R} , i.e., TRSs which only contain rules $\ell \rightarrow r$ where both ℓ and r are basic. We first show that non-termination of the narrowing relation is equivalent to the existence of a linear lower bound on the complexity of the rewrite relation. The crucial observation for the “if” direction is that every rewrite sequence with basic terms gives rise to a corresponding narrowing sequence starting with a basic term $f(x_1, \dots, x_k)$. For our restricted class of TRSs, a basic term s narrows to t if and only if there is a variable-renamed rule $\ell \rightarrow r \in \mathcal{R}$ with $\sigma = \text{mgu}(s, \ell)$ and $t = r\sigma$.

Lemma 8.14 (From Rewrite Sequences to Narrowing Sequences). *Let $m \in \mathbb{N}$ and let $s \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ with $\text{root}(s) = f$ such that $s \rightarrow_{\mathcal{R}}^m t$. Then we have*

$$f(x_1, \dots, x_k) \rightsquigarrow_{\mathcal{R}}^m t'$$

for pairwise different variables x_1, \dots, x_k , where t' matches t .

Proof. We prove the more general Lemma 10.3 in Chapter 10. □

For the “only if” direction, we show that a narrowing sequence of length n induces a rewrite sequence of the same length where the size of the start term is linear in n . To this end, we need the following two lemmas, which allow

CHAPTER 8. LOOP DETECTION

us to estimate the size of the start term of the rewrite sequence induced by a narrowing sequence.

Lemma 8.15 (Size of Unified Terms). *Let $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ be linear terms such that $\mathcal{V}(s) \cap \mathcal{V}(t) = \emptyset$ and let $\text{mgu}(s, t) = \theta$. Then*

$$\sum_{x \in \mathcal{V}(s) \cap \text{dom}(\theta)} \|x\theta\|_t \leq \|t\|_t.$$

Proof. We use induction on s . First assume $s = x \in \mathcal{V}$. If $x \notin \text{dom}(\theta)$, then the claim is trivial. If $x \in \text{dom}(\theta)$, then we have $x\theta = t$ and hence $\|x\theta\|_t = \|t\|_t$.

Now assume $s = f(s_1, \dots, s_n)$. If $t = y \in \mathcal{V}$, then we have $\theta = \{y/f(s_1, \dots, s_n)\}$ and thus the claim is trivial. Assume $t = f(t_1, \dots, t_n)$. Note that s and t have the same root symbol, since they are unifiable. By the induction hypothesis, we know

$$\sum_{x \in \mathcal{V}(s_i) \cap \text{dom}(\theta_i)} \|x\theta_i\|_t \leq \|t_i\|_t$$

for all $i \in \{1, \dots, n\}$ where θ_i is the mgu of s_i and t_i . Since s and t are linear, we have $\theta = \theta_1 \diamond \dots \diamond \theta_n$. Thus we get

$$\sum_{x \in \mathcal{V}(s) \cap \text{dom}(\theta)} \|x\theta\|_t = \sum_{\substack{i \in \{1, \dots, n\} \\ x \in \mathcal{V}(s_i) \cap \text{dom}(\theta_i)}} \|x\theta_i\|_t \leq \sum_{i \in \{1, \dots, n\}} \|t_i\|_t < \|t\|_t$$

as desired. □

Lemma 8.16. *Let $t_0 \xrightarrow{\sigma_1} \mathcal{R} \dots \xrightarrow{\sigma_n} \mathcal{R} t_n$ be a narrowing sequence where t_0 is linear. Then $\|t_0\sigma_1 \dots \sigma_n\|_t \leq \|t_0\|_t + n \cdot \max\{\|\ell\|_t \mid \ell \rightarrow r \in \mathcal{R}\}$.*

Proof. We use induction on n . If $n = 0$, then the claim is trivial. If $n > 0$, we get

$$\|t_0\sigma_1 \dots \sigma_{n-1}\|_t \leq \|t_0\|_t + (n-1) \cdot \max\{\|\ell\|_t \mid \ell \rightarrow r \in \mathcal{R}\} \quad (8.5)$$

by the induction hypothesis. Note that we have $\mathcal{V}(t_{n-1}) \subseteq \mathcal{V}(t_0\sigma_1 \dots \sigma_{n-1})$ since $t_0\sigma_1 \dots \sigma_{n-1} \xrightarrow{*}_{\mathcal{R}} t_{n-1}$. Let $\ell \rightarrow r \in \mathcal{R}$ be the rule which is used for the narrowing step $t_{n-1} \xrightarrow{\sigma_n}_{\mathcal{R}} t_n$ where ℓ and $t_0\sigma_1 \dots \sigma_{n-1}$ are variable disjoint without loss of generality. Since \mathcal{R} and t_0 are linear, ℓ , t_{n-1} , and $t_0\sigma_1 \dots \sigma_{n-1}$ are linear. Thus, since σ_n is the mgu of t_{n-1} and ℓ , we have

$$\sum_{x \in \mathcal{V}(t_{n-1}) \cap \text{dom}(\sigma_n)} \|x\sigma_n\|_t \leq \|\ell\|_t \quad (8.6)$$

by Lemma 8.15. Since ℓ and $t_0\sigma_1 \dots \sigma_{n-1}$ are variable disjoint, we have $x \notin$

8.3. INCOMPLETENESS OF LOOP DETECTION

$\text{dom}(\sigma_n)$ for all $x \in \mathcal{V}(t_0\sigma_1 \dots \sigma_{n-1}) \setminus \mathcal{V}(t_{n-1})$ and thus (8.6) implies

$$\sum_{x \in \mathcal{V}(t_0\sigma_1 \dots \sigma_{n-1}) \cap \text{dom}(\sigma_n)} \|x\sigma_n\|_t \leq \|\ell\|_t.$$

Hence, by linearity of $t_0\sigma_1 \dots \sigma_{n-1}$, we get $\|t_0\sigma_1 \dots \sigma_n\|_t \leq \|t_0\sigma_1 \dots \sigma_{n-1}\|_t + \|\ell\|_t$. With (8.5), this implies $\|t_0\sigma_1 \dots \sigma_n\|_t \leq \|t_0\|_t + n \cdot \max\{\|\ell\|_t \mid \ell \rightarrow r \in \mathcal{R}\}$ as desired. \square

Lemma 8.16 gives rise to the following interesting corollary, which states that there are no ordinary, linear, and basic TRSs with sub-linear, but non-constant complexity.

Corollary 8.17. *We have*

$$\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n) \text{ if and only if } \text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$$

for all ordinary, linear, and basic TRSs \mathcal{R} .

Proof. Clearly, $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ implies $\text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$. To see why $\text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$ also implies $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$, assume $\text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$ and $\text{rc}_{cp(\mathcal{R})}(n) \notin \Omega(n)$. Then $\text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$ implies that there are rewrite sequences $s_1 \rightarrow_{\mathcal{R}}^1 t_1, s_2 \rightarrow_{\mathcal{R}}^2 t_2, \dots$ where s_1, s_2, \dots are basic. As \mathcal{R} is linear, we may assume that s_1, s_2, \dots are also linear. Finally, as $\Sigma_d(\mathcal{R})$ is finite, we may also assume $\text{root}(s_1) = \text{root}(s_2) = \dots = f$. Then by Lemma 8.14, there are narrowing sequences $f(\mathbf{x}) \xrightarrow{\sigma_1^1}_{\mathcal{R}} t'_1, f(\mathbf{x}) \xrightarrow{\sigma_2^2}_{\mathcal{R}} t'_2, \dots$. By Lemma 8.16, we have $\|f(\mathbf{x})\sigma_n\|_t \leq \|f(\mathbf{x})\|_t + n \cdot \max\{\|\ell\|_t \mid \ell \rightarrow r \in \mathcal{R}\}$, i.e., $\|f(\mathbf{x})\sigma_n\|_t \in \mathcal{O}(n)$. However, as $f(\mathbf{x}) \xrightarrow{\sigma_n^n}_{\mathcal{R}} t'_n$ implies $f(\mathbf{x})\sigma_n \rightarrow_{\mathcal{R}}^n t'_n$, we obtain a contradiction as $\text{rc}_{cp(\mathcal{R})}(n) \notin \Omega(n)$ implies $\|f(\mathbf{x})\sigma_n\|_t \notin \mathcal{O}(n)$. \square

Now we can show that a linear lower bound is equivalent to non-termination of narrowing for a term of the form $f(x_1, \dots, x_k)$.

Lemma 8.18 (Linear Lower Bound \iff Non-Termination of Narrowing).

There is a non-terminating narrowing sequence that starts with a basic term $f(x_1, \dots, x_k)$ for pairwise different variables x_1, \dots, x_k if and only if we have $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$.

Proof. For the “only if” direction, we have an infinite sequence $f(x_1, \dots, x_k) = t_0 \xrightarrow{\sigma_1^1}_{\mathcal{R}} t_1 \xrightarrow{\sigma_2^2}_{\mathcal{R}} \dots$ for pairwise different variables x_i , where σ_i is the mgu used in the i^{th} narrowing step. By Lemma 8.16, we have

$$\|t_0\sigma_1 \dots \sigma_i\|_t \leq \|t_0\|_t + i \cdot \max\{\|\ell\|_t \mid \ell \rightarrow r \in \mathcal{R}\}$$

CHAPTER 8. LOOP DETECTION

for each $i \in \mathbb{N}$, i.e., $\|t_0\sigma_1 \dots \sigma_i\|_t$ is linear in i . Thus, the infinite family of rewrite sequences $t_0\sigma_1 \dots \sigma_i \rightarrow_{\mathcal{R}}^i t_i$ is a witness for $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$.

For the “if” direction, $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ implies that the TRS does not have constant runtime complexity. Hence, for each $m \in \mathbb{N}$ there is a rewrite sequence of length m starting with a basic term $f(\dots)$. Since $\Sigma_d(\mathcal{R})$ is finite, there exists an $f \in \Sigma_d(\mathcal{R})$ such that there are rewrite sequences of lengths $m_1 < m_2 < m_3 < \dots$ that start with basic terms with root symbol f . By Lemma 8.14 this means that the term $f(x_1, \dots, x_k)$ starts narrowing sequences of lengths $m_1 < m_2 < m_3 < \dots$, i.e., the narrowing tree with the root $f(x_1, \dots, x_k)$ has infinitely many nodes. Since $\rightarrow_{\mathcal{R}}$ is finitely branching, by König’s Lemma the tree has an infinite path, i.e., there is an infinite narrowing sequence starting with $f(x_1, \dots, x_k)$. \square

We now show that non-termination of narrowing and rewriting on possibly *infinite* basic terms are equivalent. The motivation is that the immortality problem for Turing machines allows configurations with infinitely many non-blank symbols on the tape, which can naturally be represented as infinite terms. To prove this equivalence, we need the following auxiliary lemma.

Lemma 8.19 (Unification with Infinite Terms). *Let s, t be variable-disjoint linear finite terms. If there is a substitution σ such that $s\sigma = t\sigma$ and $\text{rng}(\sigma)$ contains infinite terms, then s and t unify and the range of $\text{mgu}(s, t)$ consists of linear finite terms.*

Proof. We use structural induction on s . If $s \in \mathcal{V}$, then $\text{mgu}(s, t) = \{s/t\}$ and if $t \in \mathcal{V}$, then $\text{mgu}(s, t) = \{t/s\}$. Now let $s = f(s_1, \dots, s_k)$ and since $s\sigma = t\sigma$, we have $t = f(t_1, \dots, t_k)$. By the induction hypothesis, the ranges of the substitutions $\sigma_1 = \text{mgu}(s_1, t_1), \dots, \sigma_k = \text{mgu}(s_k, t_k)$ consist of linear finite terms. Let $\mathcal{V}(\sigma_i) = \text{dom}(\sigma_i) \cup \mathcal{V}(\text{rng}(\sigma_i))$ for all $i \in \{1, \dots, n\}$. Since s and t are variable-disjoint and linear, the sets $\mathcal{V}(\sigma_i)$ are pairwise disjoint and we have $(\mathcal{V}(s_i) \cup \mathcal{V}(t_i)) \cap \mathcal{V}(\sigma_j) = \emptyset$ for all $i \neq j$. Hence, we get $\text{mgu}(s, t) = \sigma_1 \diamond \dots \diamond \sigma_n$. As, by the induction hypothesis, $\text{rng}(\sigma_i)$ consists of linear terms and the sets $\mathcal{V}(\sigma_i)$ are pairwise disjoint, the range of $\text{mgu}(s, t)$ consists of linear terms, too. Similarly, as $\text{rng}(\sigma_i)$ only contains finite terms, this also holds for $\text{rng}(\sigma_1 \diamond \dots \diamond \sigma_n)$. \square

Lemma 8.20 (Narrowing and Rewriting with Infinite Terms). *The narrowing relation $\rightarrow_{\mathcal{R}}$ terminates on basic terms of the form $f(x_1, \dots, x_k)$ if and only if $\rightarrow_{\mathcal{R}}$ terminates on possibly infinite basic terms.*

Proof. For the “if” direction, assume there is an infinite sequence $f(x_1, \dots, x_k) = t_1 \xrightarrow{\sigma_1}_{\mathcal{R}} t_2 \xrightarrow{\sigma_2}_{\mathcal{R}} \dots$. Then $t_1\sigma_1^\omega \rightarrow_{\mathcal{R}} t_2\sigma_2^\omega \rightarrow_{\mathcal{R}} \dots$ is an infinite $\rightarrow_{\mathcal{R}}$ -sequence

8.3. INCOMPLETENESS OF LOOP DETECTION

where $\sigma_i^\omega = \sigma_i \sigma_{i+1} \dots$. Since the terms in the rules of \mathcal{R} are basic, all terms $t_i \sigma_i^\omega$ are basic, too.

For the “only if” direction, assume that there is an infinite rewrite sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots$ on possibly infinite basic terms t_i . We now show that for every finite prefix $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t_m$ of this sequence, there is a rewrite sequence $t'_1 \rightarrow_{\mathcal{R}} t'_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t'_m$ with *finite* and *linear* basic terms t'_i . This suffices for the current lemma, because it implies that the TRS does not have constant runtime complexity. As in the proof of Lemma 8.18 one can then show that there is an infinite narrowing sequence starting with a term $f(x_1, \dots, x_k)$.

It remains to prove that for every finite rewrite sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t_m$ with possibly infinite basic terms t_i , there is a rewrite sequence $t'_1 \rightarrow_{\mathcal{R}} t'_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t'_m$ with finite linear basic terms t'_i , where there exists a substitution σ such that $t'_i \sigma = t_i$ for all $i \in \{1, \dots, m\}$. We prove this claim by induction on m .

The case $m = 1$ is trivial. In the induction step, we consider the rewrite sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t_m \rightarrow_{\mathcal{R}} t_{m+1}$ of possibly infinite basic terms. By the induction hypothesis we have $t'_1 \rightarrow_{\mathcal{R}} t'_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t'_m$ for finite linear basic terms t'_i where $t'_i \sigma = t_i$ for all i . Let $\ell \rightarrow r$ be the rule applied in the rewrite step from t_m to t_{m+1} , i.e., $t_m = \ell \delta$ and $t_{m+1} = r \delta$. As $t'_m \sigma = t_m = \ell \delta$ and as w.l.o.g., ℓ is variable-disjoint from t'_1, \dots, t'_m , this means that t'_m and ℓ are unifiable using a substitution whose domain contains infinite terms. By Lemma 8.19, $\theta = \text{mgu}(t'_m, \ell)$ exists and its range consists of linear finite terms, as t'_m and ℓ are linear and finite. Let μ be a substitution such that $\theta \diamond \mu$ is like σ on t'_1, \dots, t'_m and like δ on ℓ . We define $t''_i = t'_i \theta$ for all $i \in \{1, \dots, m\}$ and $t''_{m+1} = r \theta$. Then we have $t''_1 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} t''_m = t'_m \theta = \ell \theta \rightarrow_{\mathcal{R}} r \theta = t''_{m+1}$. Moreover, we have $t''_i \mu = t'_i \theta \mu = t'_i \sigma = t_i$ for all $i \in \{1, \dots, m\}$ and $t''_{m+1} \mu = r \theta \mu = r \delta = t_{m+1}$.

It remains to show that all t''_i are linear. Let $\mathcal{V}' = \mathcal{V}(t'_1) \cup \dots \cup \mathcal{V}(t'_m)$. Since ℓ is variable-disjoint from t'_1, \dots, t'_m and $\theta = \text{mgu}(t'_m, \ell)$, $\text{rng}(\theta|_{\mathcal{V}'})$ does not contain variables from \mathcal{V}' . Since each t'_i is linear and $\text{rng}(\theta)$ consists of linear terms, this implies that t''_1, \dots, t''_m are linear, too. Similarly, t''_{m+1} is linear, as $\theta|_{\mathcal{V}(r)}$ does not contain variables from $\mathcal{V}(r)$, r is linear, and $\text{rng}(\theta)$ consists of linear terms. \square

Lemma 8.18 and Lemma 8.20 imply that for ordinary, linear, and basic TRSs, a linear lower bound is equivalent to non-termination of rewriting on possibly infinite basic terms. We now reduce the immortality problem for Turing machines to this latter problem in order to show that it is not semi-decidable.

Let $\mathcal{M} = (Q, \Gamma, \delta)$ be a Turing machine. We say that \mathcal{M} is *mortal* if and only if there is no infinite sequence $(q_1, w_1, a_1, w'_1) \rightarrow_{\mathcal{M}} (q_2, w_2, a_2, w'_2) \rightarrow_{\mathcal{M}} \dots$ of configurations. The difference to the halting problem is that for the mortality problem, one may start with a tape containing infinitely many non-blank symbols. Moreover, one can begin with any state $q_1 \in Q$. As shown in [88], the immortality problem for Turing machines is not semi-decidable.

To reduce immortality of Turing machines to non-termination of rewriting

CHAPTER 8. LOOP DETECTION

with infinite terms, we use the following encoding. For any Turing machine $\mathcal{M} = (Q, \Gamma, \delta)$, we define the TRS $\mathcal{R}_{\mathcal{M}}$. Here, f has arity 4, all symbols from Γ become function symbols of arity 1, and $Q \cup \{\underline{a} \mid a \in \Gamma\}$ are constants.

$$\begin{aligned} \mathcal{R}_{\mathcal{M}} = & \{f(q_1, a_2(xs), \underline{a_1}, ys) \rightarrow f(q_2, xs, \underline{a_2}, b(ys)) \mid a_2 \in \Gamma, \delta(q_1, a_1) = (q_2, b, L)\} \\ & \cup \{f(q_1, xs, \underline{a_1}, a_2(ys)) \rightarrow f(q_2, b(xs), \underline{a_2}, ys) \mid a_2 \in \Gamma, \delta(q_1, a_1) = (q_2, b, R)\} \end{aligned}$$

$\mathcal{R}_{\mathcal{M}}$ is an ordinary, linear, and basic TRS. Lemma 8.21 shows that immortality of \mathcal{M} is equivalent to non-termination of $\mathcal{R}_{\mathcal{M}}$ on possibly infinite basic terms.

Lemma 8.21 (Mortality of Turing Machines and Rewriting). *A Turing machine \mathcal{M} is immortal if and only if there is a possibly infinite basic term that starts an infinite rewrite sequence with $\mathcal{R}_{\mathcal{M}}$.*

Proof. We define the following functions word and word^{-1} to convert possibly infinite words over Γ to infinite constructor terms and vice versa.

$$\begin{aligned} \text{word}(a.w) &= a(\text{word}(w)) \\ \text{word}^{-1}(t) &= \begin{cases} a.\text{word}^{-1}(t') & \text{if } t = a(t') \text{ and } a \in \Gamma \\ \square^\omega & \text{otherwise} \end{cases} \end{aligned}$$

For each configuration (q, w, a, w') let

$$\text{term}(q, w, a, w') = f(q, \text{word}(w), \underline{a}, \text{word}(w')).$$

For the “only if” direction, it suffices to show that

$$(q_1, w_1, a_1, w'_1) \rightarrow_{\mathcal{M}} (q_2, w_2, a_2, w'_2)$$

implies

$$\text{term}(q_1, w_1, a_1, w'_1) \rightarrow_{\mathcal{R}_{\mathcal{M}}} \text{term}(q_2, w_2, a_2, w'_2).$$

We regard the case where $\delta(q_1, a_1) = (q_2, b, L)$ (the case $\delta(q_1, a_1) = (q_2, b, R)$ works analogously). Then $w_1 = a_2.w_2$ and $w'_2 = b.w'_1$. Thus,

$$\begin{aligned} \text{term}(q_1, w_1, a_1, w'_1) &= f(q_1, a_2(\text{word}(w_2)), \underline{a_1}, \text{word}(w'_1)) \\ &\rightarrow_{\mathcal{R}_{\mathcal{M}}} f(q_2, \text{word}(w_2), \underline{a_2}, b(\text{word}(w'_1))) \\ &= \text{term}(q_2, w_2, a_2, w'_2), \end{aligned}$$

as desired.

For the “if” direction, it suffices to show that if t_1 is a possibly infinite basic term with $t_1 \rightarrow_{\mathcal{R}_{\mathcal{M}}} t_2$, then $\text{conf}_1 \rightarrow_{\mathcal{M}} \text{conf}_2$ with

$$\text{conf}_i = (t_i|_1, \text{word}^{-1}(t_i|_2), \text{character}(t_i|_3), \text{word}^{-1}(t_i|_4))$$

for both $i \in \{1, 2\}$, where $\text{character}(\underline{a}) = a$. Clearly, we have $t_i|_1 \in Q$ and $t_i|_3 \in \{\underline{a} \mid a \in \Gamma\}$. We regard the case where the rule of $\mathcal{R}_{\mathcal{M}}$ used for the

8.3. INCOMPLETENESS OF LOOP DETECTION

rewrite step corresponds to a shift to the left (the right shift works analogously). Then we have $t_1 = f(q_1, a_2(s), \underline{a_1}, s')$ and $t_2 = f(q_2, s, \underline{a_2}, b(s'))$ for $a_1, a_2, b \in \Gamma$, some possibly infinite constructor terms s, s' , and $q_1, q_2 \in Q$. Moreover, by construction we have $\delta(q_1, a_1) = (q_2, b, L)$. Hence,

$$\begin{aligned} conf_1 &= (q_1, a_2.\text{word}^{-1}(s), a_1, \text{word}^{-1}(s')) \\ &\rightarrow_{\mathcal{M}} (q_2, \text{word}^{-1}(s), a_2, b.\text{word}^{-1}(s')) \\ &= conf_2. \end{aligned}$$

□

Proof of Theorem 8.13. For any Turing machine \mathcal{M} we have the following:

$$\begin{aligned} &\mathcal{M} \text{ is immortal} \\ \iff &\rightarrow_{\mathcal{R}_{\mathcal{M}}} \text{ does not terminate on possibly infinite basic terms (Lemma 8.21)} \\ \iff &\rightsquigarrow_{\mathcal{R}_{\mathcal{M}}} \text{ does not terminate on basic terms } f(x_1, \dots, x_k) \quad (\text{Lemma 8.20}) \\ \iff &\text{rc}_{cp(\mathcal{R}_{\mathcal{M}})}(n) \in \Omega(n) \quad (\text{Lemma 8.18}) \end{aligned}$$

Thus, a semi-decision procedure for $\text{rc}_{cp(\mathcal{R}_{\mathcal{M}})}(n) \in \Omega(n)$ would result in a semi-decision procedure for immortality of Turing machines. However, immortality of Turing machines is known to be not semi-decidable [88]. Thus, $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ cannot be semi-decidable for the class of ordinary, linear, and basic TRSs.

Note that the existence of decreasing loops is semi-decidable, since one can recursively enumerate all possible rewrite sequences $\ell \rightarrow_{\mathcal{R}}^+ C[r]$ and since it is decidable whether an actual rewrite sequence is a decreasing loop. This implies that loop detection by decreasing loops cannot be complete for ordinary, linear, and basic TRSs. □

8.4 Innermost Decreasing Loops

So far, we used loop detection to prove lower bounds on the runtime complexity of *full* rewriting. However, TRSs resulting from the translation of programs often have to be evaluated with an *innermost* strategy (e.g., [63, 107]). Hence, we now show how to adapt loop detection to innermost runtime complexity. To do so, we introduce innermost decreasing loops, which are like decreasing loops, but here only *non-overlapping* rewrite sequences are considered.

Definition 8.22 (Non-Overlapping Rewriting). For a TRS \mathcal{R} , we say that a term s reduces to t by *non-overlapping* rewriting (denoted $s \xrightarrow{n}_{\mathcal{R}} t$) if there is a context C , a substitution σ , and a rule $\ell \xrightarrow{\mathcal{K}} r \in \mathcal{R}$ such that $s = C[\ell\sigma]$, $t = C[r\sigma]$, and no proper non-variable subterm of $\ell\sigma$ unifies with any (variable-renamed) left-hand side of a rule in \mathcal{R} .

Clearly, any non-overlapping rewrite step is an innermost step (i.e., $s \xrightarrow{n}_{\mathcal{R}} t$ implies $s \xrightarrow{i}_{\mathcal{R}} t$), but not vice versa. For innermost decreasing loops, instead of reductions $\ell \rightarrow_{\mathcal{R}}^+ C[r]$ we now consider reductions of the form $\ell \xrightarrow{n}_{\mathcal{R}}^+ C[r]$.

To find non-overlapping rewrite sequences, *non-overlapping narrowing* can be used instead of narrowing. Like non-overlapping rewriting, non-overlapping narrowing does not allow reduction steps where a proper non-variable subterm of the redex unifies with a (variable-renamed) left-hand side of a rule.

The following theorem shows that each non-overlapping decreasing loop gives rise to a linear lower bound on the innermost runtime complexity of a TRS, provided that there are no infinite rewrite sequences with cost 0.

Theorem 8.23 (Lower Bounds for Innermost Rewriting by Loop Detection). *Let \mathcal{R} be a TRS. If $\mathcal{R}_0 = \{\ell \xrightarrow{\mathcal{K}} r \in \mathcal{R} \mid \mathcal{K} = 0\}$ terminates and \mathcal{R} has a decreasing loop $\ell \xrightarrow{n}_{\mathcal{R}}^+ C[r]$, then $rc_{cp_i}(\mathcal{R})(n) \in \Omega(n)$.*

Proof. If \mathcal{R} is not innermost terminating on basic terms, then the claim is trivial since \mathcal{R}_0 terminates and thus non-termination of \mathcal{R} on basic terms implies $rc_{cp_i}(\mathcal{R})(n) \in \Omega(\omega)$. Otherwise, let $\bar{\delta}$ be a substitution such that for all $x \in \mathcal{V}(\bar{\ell}\theta^n)$, we have $x\bar{\delta} \xrightarrow{i}_{\mathcal{R}}^* x\bar{\delta}$ and $x\bar{\delta}$ is in normal form.

Similar to the proof of Theorem 8.6, we have

$$\begin{aligned} \bar{\ell}\theta^n\bar{\delta} &= \ell\theta^{n-1}\bar{\delta} \xrightarrow{i}_{\mathcal{R}}^+ C[r]\theta^{n-1}\bar{\delta} \supseteq r\theta^{n-1}\bar{\delta} \\ &= \bar{\ell}\sigma\theta^{n-1}\bar{\delta} = \bar{\ell}\theta^{n-1}(\sigma \diamond \theta^{n-1})|_{\text{dom}(\sigma)}\bar{\delta} = \bar{\ell}\theta^{n-1}\bar{\delta}' \end{aligned}$$

for the substitution $\bar{\delta}' = (\sigma \diamond \theta^{n-1})|_{\text{dom}(\sigma)} \diamond \bar{\delta}$.

The rewrite sequence $\ell\theta^{n-1}\bar{\delta} \xrightarrow{i}_{\mathcal{R}}^+ C[r]\theta^{n-1}\bar{\delta}$ is indeed an innermost reduction. To see this, recall that $\bar{\delta}$ only instantiates variables by normal forms. Moreover, θ has no defined symbols in its range, since ℓ is basic. For this reason, $\theta^{n-1}\bar{\delta}$ also instantiates all variables by normal forms, i.e., no rewrite step is possible

8.4. INNERMOST DECREASING LOOPS

for the terms in the range of $\theta^{n-1}\bar{\delta}$. Moreover, the subterms of the redexes in the reduction $\ell \xrightarrow{n\rightarrow\mathcal{R}}^+ C[r]$ do not unify with left-hand sides of rules. Hence, these subterms remain in normal form if one instantiates them with $\theta^{n-1}\bar{\delta}$. This implies $\ell\theta^{n-1}\bar{\delta} \xrightarrow{i\rightarrow\mathcal{R}}^+ C[r]\theta^{n-1}\bar{\delta}$.

Thus, for each $n \in \mathbb{N}$, there is a rewrite sequence which has at least cost $\mathcal{K} \cdot n$ starting with $\bar{\ell}\theta^n$. This term is basic, since the range of θ only contains terms of the form $\bar{\ell}|_{\xi_i}$. Each $\bar{\ell}|_{\xi_i}$ is a constructor term, since ξ_i cannot be the root position, due to $r \notin \mathcal{V}$. By construction, θ does not duplicate variables, as ℓ and thus $\bar{\ell}|_{\xi_1}, \dots, \bar{\ell}|_{\xi_m}$ only contain each x_i once. Therefore, we have $\|\bar{\ell}\theta^n\|_t \in \mathcal{O}(n)$. Moreover, we have $\theta \neq \text{id}$ (otherwise \mathcal{R} would not terminate on basic terms) and thus $\|\bar{\ell}\theta^n\|_t$ is strictly monotonically increasing in n . Hence, we obtain $\text{rc}_{cp_i}(\mathcal{R})(n) \in \Omega(n)$ by Lemma 4.41. \square

The following example shows that we indeed have to require $\ell \xrightarrow{n\rightarrow\mathcal{R}}^+ C[r]$ instead of just $\ell \xrightarrow{i\rightarrow\mathcal{R}}^+ C[r]$ in Theorem 8.23. The essential property of non-overlapping rewriting is that if a substitution δ instantiates all variables with normal forms, then $s \xrightarrow{n\rightarrow\mathcal{R}} t$ still implies $s\delta \xrightarrow{i\rightarrow\mathcal{R}} t\delta$. In contrast, $s \xrightarrow{i\rightarrow\mathcal{R}} t$ does not imply $s\delta \xrightarrow{i\rightarrow\mathcal{R}} t\delta$.

Example 8.24 (Non-Overlapping Rewriting). Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lll} f(y) & \xrightarrow{1} & h(g(y)), \\ h(g(y)) & \xrightarrow{1} & f(g(y)), \\ g(g(y)) & \xrightarrow{1} & y \end{array} \right\}.$$

We clearly have $f(y) \xrightarrow{i\rightarrow\mathcal{R}}^+ f(g(y))$, but $f(y) \not\xrightarrow{n\rightarrow\mathcal{R}}^+ f(g(y))$. If we replaced “ $\ell \xrightarrow{n\rightarrow\mathcal{R}}^+ C[r]$ ” by “ $\ell \xrightarrow{i\rightarrow\mathcal{R}}^+ C[r]$ ” in Theorem 8.23, then we would falsely deduce a linear lower bound from the decreasing loop $f(y) \xrightarrow{i\rightarrow\mathcal{R}}^+ f(g(y))$. However, all innermost rewrite sequences that start with basic terms have at most length 4 for this TRS, i.e., $\text{rc}_{cp_i}(\mathcal{R})(n) \in \Theta(1)$. The problem is that the rewrite sequence $f(y) \xrightarrow{i\rightarrow\mathcal{R}}^+ f(g(y))$ does not remain an innermost sequence anymore if one instantiates y with the normal form $g(y)$, i.e., we have $f(g(y)) \not\xrightarrow{i\rightarrow\mathcal{R}}^+ f(g(g(y)))$.

It is also crucial to require termination of \mathcal{R}_0 . The reason is that the innermost evaluation of a basic term can lead to a non-terminating redex s whose infinite reduction has cost 0. Then the evaluation “gets stuck” in the infinite reduction of s without any costs. This problem is irrelevant for full rewriting, where we can simply continue the reduction with a non-innermost redex in such cases. The following example illustrates this problem.

Example 8.25 (Termination of \mathcal{R}_0). Let

$$\mathcal{R} = \{f(\text{succ}(x), y) \xrightarrow{1} f(x, a), a \xrightarrow{0} a\}.$$

CHAPTER 8. LOOP DETECTION

If we wouldn't require termination of $\mathcal{R}_0 = \{\mathbf{a} \xrightarrow{0} \mathbf{a}\}$ in Theorem 8.23, then we would falsely deduce $\text{rc}_{cp_i}(\mathcal{R})(n) \in \Omega(n)$ due to the decreasing loop

$$\mathbf{f}(\text{succ}(x), y) \xrightarrow{\frac{1}{n}}_{\mathcal{R}} \mathbf{f}(x, \mathbf{a}).$$

However, we have $\text{rc}_{cp_i}(\mathcal{R})(n) \in \Theta(1)$, as we have

$$\mathbf{f}(\text{succ}(x), y)\sigma \xrightarrow{\frac{1}{i}}_{\mathcal{R}} \mathbf{f}(x, \mathbf{a})\sigma \xrightarrow{0}_{\mathcal{R}} \mathbf{f}(x, \mathbf{a})\sigma \xrightarrow{0}_{\mathcal{R}} \dots$$

for every substitution σ such that $\mathbf{f}(\text{succ}(x), y)\sigma$ is basic.

Based on the notion of non-overlapping decreasing loops, it is straightforward to adapt the concept of *compatible* decreasing loops in Definition 8.11 and Theorem 8.12 to innermost rewriting.

8.5 Related Work

As mentioned in Chapter 6, loop detection is related to techniques that search for loops in order to prove non-termination of TRSs [67, 108, 125, 132, 134]. To find loops, sophisticated techniques have been proposed. For example, [67, 108] rely on semi-unification in combination with pruning of rules [108] resp. embedded into the *Dependency Pair Framework* [67]. The tool **Matchbox** [125, 134] is specialized to string rewriting and obtains impressive results regarding the detection of non-termination for this variant of rewriting at the annual *Termination and Complexity Competition* [121]. The technique proposed in [132] relies on a SAT encoding to search for loops.

In contrast, we use a naive approach to search for decreasing loops: We perform a fixed number of narrowing steps to obtain a representative set of rewrite sequences and check the number of compatible decreasing loops for each of them. While our experiments (cf. Chapter 12) indicate that such an approach is sufficient in almost all cases, the performance of our technique could certainly be improved by using more sophisticated heuristics. Moreover, better heuristics might allow for more precise bounds in some cases (e.g., exponential or infinite bounds for TRSs where our current approach infers a linear lower bound).

A technique to decide whether a given loop is also an *innermost* loop is presented in [123], i.e., this technique can decide if a given loop witnesses non-termination w.r.t. an innermost reduction strategy. In contrast, our adaption of loop detection to innermost rewriting (Section 8.4) is based on non-overlapping rewriting. Thus, it is incomplete, as there are decreasing loops which witness a linear lower bound for innermost rewriting, but cannot be detected by our technique as they are overlapping. As an example, consider the TRS

$$h(x) \rightarrow f(g(x)) \quad f(g(\text{succ}(x))) \rightarrow h(x) \quad g(\text{succ}(\text{zero})) \rightarrow a$$

It admits the decreasing loop $h(\text{succ}(x)) \rightarrow f(g(\text{succ}(x))) \rightarrow h(x)$, which is overlapping as the subterm $g(\text{succ}(x))$ of the second redex unifies with the left-hand side $g(\text{succ}(\text{zero}))$. Nevertheless, the family of rewrite sequences $h(\text{succ}^n(x)) \rightarrow^* h(x)$ witnesses a linear lower bound on the innermost runtime complexity of the TRS. Thus, it might be worthwhile to investigate an adaption of the technique from [123] to decreasing loops.

The paper [22] uses a proof quite similar to the one from Section 8.3 to show undecidability of the question whether a TRS has a *finite forward closure* resp. the *finite variant property*. Like our proof, it relies on the undecidability of the mortality problem for Turing machines. Moreover, finiteness of the forward closure of a TRS is closely related to termination of narrowing w.r.t. a finite set of start terms. Similarly, our proof exploits that (non-)termination of narrowing w.r.t. a finite set of start terms is closely related to the existence of a linear lower bound for the runtime complexity of a TRS.

8.6 Conclusion and Future Work

We introduced *loop detection*, a powerful technique for the inference of linear (Section 8.1) and exponential (Section 8.2) lower bounds on the runtime complexity of term rewrite systems. It searches for *decreasing loops*, a generalization of the well-known notion of *loops* from termination analysis of TRSs. Consequently, loop detection can also infer infinite lower bounds.

Given a single rewrite sequence $s \rightarrow_{\mathcal{R}}^* t$, loop detection works purely syntactically, i.e., it deduces lower bounds by examining the structure of the terms s and t . Rewrite sequences that give rise to high (i.e., preferably infinite or exponential, but at least linear) lower bounds can be searched via *narrowing*.

Regarding linear bounds, we are not aware of an ordinary left-linear constructor system which has at least linear runtime complexity, but no decreasing loop. Note that this fragment of term rewriting essentially corresponds to first-order functional programs and hence it is particularly interesting in the context of program verification. Thus, the question whether loop detection is a semi-decision procedure for $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ arises. In Section 8.3, we give a negative answer to this question by reducing *immortality* of Turing machines to the question whether a TRS has a linear lower bound.

In future work, one should investigate if more sophisticated techniques to search for decreasing loops, which might be inspired by techniques to detect looping non-termination (cf. Section 8.5), can improve the performance and the precision of our technique. Moreover, finding an ordinary left-linear constructor system with at least linear complexity, but without a decreasing loop (which has to exist due to the results from Section 8.3) would be enlightening. During the last days of my work on this thesis, I discovered the paper [36], which describes a small, immortal Turing Machine which is aperiodic, which essentially means that there is no non-empty run from a configuration (with potentially infinitely many non-blank symbols on the tape) to itself. The corresponding TRS

$$\begin{aligned}
b(xs, 0, \text{cons}(y, ys)) &\rightarrow d(\text{cons}(1, xs), y, ys) \\
b(xs, 1, \text{cons}(y, ys)) &\rightarrow p(\text{cons}(1, xs), y, ys) \\
b(xs, 2, \text{cons}(y, ys)) &\rightarrow p(\text{cons}(2, xs), y, ys) \\
d(\text{cons}(x, xs), 0, ys) &\rightarrow b(xs, x, \text{cons}(1, ys)) \\
d(\text{cons}(x, xs), 1, ys) &\rightarrow q(xs, x, \text{cons}(1, ys)) \\
d(\text{cons}(x, xs), 2, ys) &\rightarrow q(xs, x, \text{cons}(2, ys)) \\
q(\text{cons}(x, xs), 0, ys) &\rightarrow b(xs, x, \text{cons}(2, ys)) \\
q(\text{cons}(x, xs), 1, ys) &\rightarrow b(xs, x, \text{cons}(0, ys)) \\
q(xs, 2, \text{cons}(y, ys)) &\rightarrow p(\text{cons}(0, xs), y, ys) \\
p(xs, 0, \text{cons}(y, ys)) &\rightarrow d(\text{cons}(2, xs), y, ys) \\
p(xs, 1, \text{cons}(y, ys)) &\rightarrow d(\text{cons}(0, xs), y, ys) \\
p(\text{cons}(x, xs), 2, ys) &\rightarrow q(xs, x, \text{cons}(0, ys))
\end{aligned}$$

seems to be the desired left-linear constructor system. However, it remains to be proven that it does not have a decreasing loop. Finally, ideas from [123]

8.6. CONCLUSION AND FUTURE WORK

might help to further improve the applicability of loop detection for innermost rewriting.

Lower Bounds for Term Rewriting by Induction

In this chapter, we present our second approach to generate lower bounds for $\text{rc}_{cp}(\mathcal{R})$ (by the so-called *induction technique*). To illustrate the idea, consider the following TRS \mathcal{R}_{qs} for *Quicksort*.¹ The auxiliary function $\text{low}(x, xs)$ returns those elements from the list xs that are smaller than x (and high works analogously). To ease readability, we use infix notation for the function symbols \preceq and $++$.

Example 9.1 (TRS \mathcal{R}_{qs} for Quicksort). The following TRS \mathcal{R}_{qs} implements the algorithm *Quicksort*.

```

 $\alpha_0$  :       $\text{qs}(\text{nil}) \rightarrow \text{nil}$ 
 $\alpha_1$  :       $\text{qs}(\text{cons}(x, xs)) \rightarrow \text{qs}(\text{low}(x, xs)) ++ \text{cons}(x, \text{qs}(\text{high}(x, xs)))$ 
 $\alpha_2$  :       $\text{low}(x, \text{nil}) \rightarrow \text{nil}$ 
 $\alpha_3$  :       $\text{low}(x, \text{cons}(y, ys)) \rightarrow \text{ifLow}(x \preceq y, x, \text{cons}(y, ys))$ 
 $\alpha_4$  :       $\text{ifLow}(\text{true}, x, \text{cons}(y, ys)) \rightarrow \text{low}(x, ys)$ 
 $\alpha_5$  :       $\text{ifLow}(\text{false}, x, \text{cons}(y, ys)) \rightarrow \text{cons}(y, \text{low}(x, ys))$ 
 $\alpha_6$  :       $\text{high}(x, \text{nil}) \rightarrow \text{nil}$ 
 $\alpha_7$  :       $\text{high}(x, \text{cons}(y, ys)) \rightarrow \text{ifHigh}(x \preceq y, x, \text{cons}(y, ys))$ 
 $\alpha_8$  :       $\text{ifHigh}(\text{true}, x, \text{cons}(y, ys)) \rightarrow \text{cons}(y, \text{high}(x, ys))$ 
 $\alpha_9$  :       $\text{ifHigh}(\text{false}, x, \text{cons}(y, ys)) \rightarrow \text{high}(x, ys)$ 
 $\alpha_{10}$  :       $\text{zero} \preceq x \rightarrow \text{true}$ 
 $\alpha_{11}$  :       $\text{succ}(x) \preceq \text{zero} \rightarrow \text{false}$ 
 $\alpha_{12}$  :       $\text{succ}(x) \preceq \text{succ}(y) \rightarrow x \preceq y$ 
 $\alpha_{13}$  :       $\text{nil} ++ ys \rightarrow ys$ 
 $\alpha_{14}$  :       $\text{cons}(x, xs) ++ ys \rightarrow \text{cons}(x, xs ++ ys)$ 

```

For any $n \in \mathbb{N}$, let $\text{gen}_{\text{List}}(n)$ be the term $\overbrace{\text{cons}(\text{zero}, \dots, \text{cons}(\text{zero}, \text{nil}) \dots)}^{n \times}$, i.e., the list of length n where all elements have the value zero (we also use the no-

¹This TRS corresponds to “Rubio.04/quick.xml” from the *Termination Problems Data Base* [122] used in the annual *Termination and Complexity Competition* [121].

CHAPTER 9. INDUCTION TECHNIQUE

tation “ $\text{cons}^n(\text{zero}, \text{nil})$ ”). To find lower bounds, we show how to automatically generate *rewrite lemmas* that describe families of rewrite sequences. For example, our induction technique infers the following rewrite lemma automatically.

$$\text{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{3 \cdot n^2 + 2 \cdot n + 1} \text{gen}_{\text{List}}(n) \quad (9.1)$$

The rewrite lemma means that for any $n \in \mathbb{N}$, there is a rewrite sequence of at least cost $3 \cdot n^2 + 2 \cdot n + 1$ that reduces $\text{qs}(\text{cons}^n(\text{zero}, \text{nil}))$ to $\text{cons}^n(\text{zero}, \text{nil})$. From this rewrite lemma, our technique concludes that the runtime of \mathcal{R}_{qs} is at least quadratic. So in contrast to the technique presented in Chapter 8, the induction technique can also prove super-linear polynomial lower bounds.

Up to minor syntactic differences, Example 9.1 is essentially a functional program, i.e., Example 9.1 can easily be transformed to a program written in a functional programming language like Haskell or OCaml. In general, the induction technique can be used to infer lower bounds on the worst-case complexity of (real-world) programs operating on tree-shaped data structures, as such programs can naturally be expressed as term rewrite systems. Consequently, it has important applications in the context of software verification and cybersecurity, as such an analysis can, e.g., be used to detect denial-of-service vulnerabilities. See Section 1.2 for a detailed discussion of the relation between (term) rewriting and real-world programs.

Section 9.1 introduces the concepts of *rewrite lemmas* and *generator symbols* like gen_{List} and other preliminaries. Section 9.2 shows how our implementation automatically speculates conjectures that may result in rewrite lemmas. In Section 9.3, we explain how to verify speculated conjectures automatically by induction. From these induction proofs, one can deduce information on the cost of the rewrite sequences that are represented by a rewrite lemma, cf. Section 9.4. Thus, the use of induction to infer lower runtime bounds is a novel application for automated inductive theorem proving. Afterwards, Section 9.5 shows how rewrite lemmas are used to infer bounds for the complexity of a whole TRS.

Clearly, speculating and proving rewrite lemmas like (9.1) is very challenging. Thus, Section 9.6 introduces *indefinite rewrite lemmas*, i.e., rewrite lemmas with unknown right-hand sides. So whenever the inference of a definite rewrite lemma like (9.1) fails, we can try to prove an indefinite lemma instead. While indefinite lemmas are of limited use for the inference of further rewrite lemmas, the structure of their proofs still allows us to deduce lower bounds.

Up to this point, another drawback of the induction technique is that it can only reason about *homogeneous* data structures like, e.g., lists of zeros in (9.1). Thus, we introduce an *argument filtering* technique in Section 9.7 which allows us to also prove lower bounds for algorithms operating on inhomogeneous data structures in some cases.

After lifting the induction technique to *innermost* rewriting in Section 9.8, we compare it with the loop detection technique from Chapter 8 (Section 9.9) and other related work (Section 9.10) and conclude in Section 9.11.

9.1 From Term Rewriting to Rewrite Lemmas

Our approach is based on rewrite lemmas containing *generator symbols* such as gen_{List} for types like List. Thus, in the first step of our approach we compute suitable types for the TRS \mathcal{R} to be analyzed. Standard TRSs do not have any type annotations or built-in types, but they are defined over untyped signatures Σ . Definition 9.2 extends them with types (see, e.g., [57, 86, 133]), where for simplicity, we restrict ourselves to monomorphic types.

Definition 9.2 (Typing). Let Σ be an (untyped) signature. A many-sorted signature Σ' is a *typed variant* of Σ if it contains the same function symbols as Σ , with the same arities. Similarly, in a typed variant \mathcal{V}' of the variables \mathcal{V} , every variable has a type τ . We always assume that for every type τ , \mathcal{V}' contains infinitely many variables of type τ . Given Σ' and \mathcal{V}' , $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is a *well-typed* term of type τ w.r.t. Σ' and \mathcal{V}' if

- $t \in \mathcal{V}'$ is a variable of type τ or
- $t = f(t_1, \dots, t_m)$ with $m \in \mathbb{N}$, where each t_i is a well-typed term of type τ_i and $f \in \Sigma'$ has the type $\tau_1 \times \dots \times \tau_m \rightarrow \tau$.

A term rewrite rule $\ell \rightarrow r$ is well typed if ℓ and r are well-typed terms of the same type. A TRS is well typed if all of its rules are well typed.² Furthermore, a substitution σ is well typed if $x\sigma$ is a well-typed term of type τ for each $x \in \text{dom}(\sigma)$ of type τ .

For any TRS \mathcal{R} , a standard type inference algorithm (e.g., [100]) can compute a typed variant Σ' such that \mathcal{R} is well typed. Here, we compute typed variants where the set of terms is decomposed into as many types as possible (i.e., where as few terms as possible are considered to be “well typed”).

Example 9.3. \mathcal{R}_{qs} is well typed w.r.t. the many-sorted signature Σ with the following function symbols and types:

| | | | |
|--------------------------|--|-----------------------------------|--|
| <code>nil</code> | : List | <code>qs</code> | : List \rightarrow List |
| <code>cons</code> | : Nat \times List \rightarrow List | <code>++</code> | : List \times List \rightarrow List |
| <code>zero</code> | : Nat | <code>\preceq</code> | : Nat \times Nat \rightarrow Bool |
| <code>succ</code> | : Nat \rightarrow Nat | <code>low, high</code> | : Nat \times List \rightarrow List |
| <code>true, false</code> | : Bool | <code>ifLow, ifHigh</code> | : Bool \times Nat \times List \rightarrow List |

A type τ *depends* on another type τ' (denoted $\tau \sqsubseteq_{\text{dep}} \tau'$) if $\tau = \tau'$ or if there is a $\mathbf{c} \in \Sigma_{\mathbf{c}}(\mathcal{R})$ of type $\tau_1 \times \dots \times \tau_m \rightarrow \tau$ where $\tau_i \sqsubseteq_{\text{dep}} \tau'$ for some $i \in \{1, \dots, m\}$. For example, we have List \sqsubseteq_{dep} Nat in Example 9.3. To ease the presentation, we do not allow mutually recursive types (i.e., if $\tau \sqsubseteq_{\text{dep}} \tau'$ and $\tau' \sqsubseteq_{\text{dep}} \tau$, then

²W.l.o.g., here one may rename the variables in every rule. Then it is not a problem if the variable x is used with type τ_1 in one rule and with type τ_2 in another rule.

CHAPTER 9. INDUCTION TECHNIQUE

$\tau' = \tau$).

To represent families of terms, we now introduce generator symbols gen_τ and generator equations. For any $n \in \mathbb{N}$, $\text{gen}_\tau(n)$ represents a term from $\mathcal{T}(\Sigma_c(\mathcal{R}))$ where a recursive constructor of type τ is nested n times. A constructor

$$\mathbf{c} : \tau_1 \times \dots \times \tau_m \rightarrow \tau$$

is called *recursive* if $\tau_i = \tau$ for some $i \in \{1, \dots, m\}$. For the type Nat above, we have the following generator equations:

$$\text{gen}_{\text{Nat}}(0) = \mathbf{zero} \tag{9.2}$$

$$\text{gen}_{\text{Nat}}(n+1) = \mathbf{succ}(\text{gen}_{\text{Nat}}(n)) \tag{9.3}$$

If a constructor has a non-recursive argument of type τ' , then gen_τ instantiates this argument by $\text{gen}_{\tau'}(0)$. For the type List , we get:

$$\text{gen}_{\text{List}}(0) = \mathbf{nil} \tag{9.4}$$

$$\text{gen}_{\text{List}}(n+1) = \mathbf{cons}(\text{gen}_{\text{Nat}}(0), \text{gen}_{\text{List}}(n)) \tag{9.5}$$

Thus, the set of \mathcal{R}_{qs} 's generator equations is $\mathcal{G}_{\text{qs}} = \{(9.2), (9.3), (9.4), (9.5)\}$. If a constructor has several recursive arguments, then several generator equations are possible. For a type Tree with the constructors $\mathbf{leaf} : \text{Tree}$ and $\mathbf{node} : \text{Tree} \times \text{Tree} \rightarrow \text{Tree}$, we have

$$\begin{array}{ll} \text{gen}_{\text{Tree}}(0) = \mathbf{leaf} & \text{and} \\ \text{gen}_{\text{Tree}}(n+1) = \mathbf{node}(\text{gen}_{\text{Tree}}(n), \mathbf{leaf}) & \text{or} \\ \text{gen}_{\text{Tree}}(n+1) = \mathbf{node}(\mathbf{leaf}, \text{gen}_{\text{Tree}}(n)). \end{array}$$

Similarly, if a type has several non-recursive or recursive constructors, then different generator equations can be obtained by considering all combinations of non-recursive and recursive constructors.

To ease readability, we only consider generator equations for *simply structured* types τ . Such types have exactly two constructors $\mathbf{c}, \mathbf{d} \in \Sigma_c(\mathcal{R})$, where \mathbf{c} is not recursive, \mathbf{d} has exactly one argument of type τ , and each argument type $\tau' \neq \tau$ of \mathbf{c} or \mathbf{d} is simply structured, too. Our approach is easily extended to more complex types by heuristically choosing one of the possible generator equations.³

³ For types with several recursive or non-recursive constructors, our heuristic prefers to use those constructors for the generator equations that occur in the left-hand sides of (preferably recursive) rules of the TRS. For a constructor with several recursive argument positions like \mathbf{node} , we examine how often the TRS contains recursive calls in the respective arguments of \mathbf{node} . If there are more recursive calls in the first arguments of \mathbf{node} than in the second one, then we take the generator equation $\text{gen}_{\text{Tree}}(n+1) = \mathbf{node}(\text{gen}_{\text{Tree}}(n), \mathbf{leaf})$ instead of $\text{gen}_{\text{Tree}}(n+1) = \mathbf{node}(\mathbf{leaf}, \text{gen}_{\text{Tree}}(n))$.

9.1. FROM TERM REWRITING TO REWRITE LEMMAS

Definition 9.4 (Generator Symbols and Equations). Let $\Sigma_{\mathbb{N}} = \{+, \cdot\} \cup \mathbb{N}$ be a many-sorted signature with the types $+, \cdot : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and $n : \mathbb{N}$ for each natural number n . For every type $\tau \neq \mathbb{N}$, let gen_{τ} be a fresh *generator symbol* of type $\mathbb{N} \rightarrow \tau$. Given a TRS \mathcal{R} , the set $\mathcal{G}_{\mathcal{R}}$ consists of the following *generator equations* for every simply structured type τ with the constructors $\text{c} : \tau_1 \times \dots \times \tau_m \rightarrow \tau$ and $\text{d} : \tau'_1 \times \dots \times \tau'_o \rightarrow \tau$, where $\tau'_j = \tau$.

$$\begin{aligned} \text{gen}_{\tau}(0) &= \text{c}(\text{gen}_{\tau_1}(0), \dots, \text{gen}_{\tau_m}(0)) \\ \text{gen}_{\tau}(n+1) &= \text{d}(\text{gen}_{\tau'_1}(0), \dots, \text{gen}_{\tau'_{j-1}}(0), \text{gen}_{\tau}(n), \text{gen}_{\tau'_{j+1}}(0), \dots, \text{gen}_{\tau'_o}(0)) \end{aligned}$$

Given a set of generator equations \mathcal{G} , $\Sigma_{\mathcal{G}}$ denotes the many-sorted signature containing all generator symbols from \mathcal{G} .

Later on, we will use generator symbols and generator equations to infer *rewrite lemmas* for a TRS \mathcal{R} . Such rewrite lemmas are “meta rules” which represent families of \mathcal{R} -sequences. However, in contrast to TRS rules, rewrite lemmas contain natural numbers and generator symbols and they may have non-constant costs. In the following, let \mathcal{A} be the infinite set containing all valid equations in the theory of \mathbb{N} with addition and multiplication and, for any set of equations \mathcal{E} , let $s \equiv_{\mathcal{E}} t$ be a shorthand for $\mathcal{E} \models s = t$, i.e., $s \equiv_{\mathcal{E}} t$ means that the equation $s = t$ is true in all models of \mathcal{E} .

Definition 9.5 (Rewrite Lemmas). We call a many-sorted signature Σ a *standard signature* if it does not use the type \mathbb{N} .

Let \mathcal{G} be a set of generator equations and let Σ be a standard signature. We call $\ell \xrightarrow{\mathcal{G}} r$ a *rewrite lemma* over Σ w.r.t. \mathcal{G} if $\ell \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}} \cup \Sigma_{\mathcal{G}}, \mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}} \cup \Sigma_{\mathcal{G}}, \mathcal{V}(\ell))$ are well-typed terms of the same type, $\text{root}(\ell) \in \Sigma$, $c\theta \in \mathbb{N}$ for each $\theta : \mathcal{V}(\ell) \rightarrow \mathbb{N}$, and c is weakly monotonically increasing.

Let \mathcal{L} be a set of rewrite lemmas and let s and t be well-typed terms. We have $s \xrightarrow{\mathcal{L}} t$ if there is a rule $\ell \xrightarrow{\mathcal{G}} r \in \mathcal{L}$, a context C , and a well-typed substitution σ such that $C[\ell\sigma] = s$, $C[r\sigma] = t$, and $c' \equiv_{\mathcal{A}} c\sigma$.

We define $\Sigma_{\text{d}}(\mathcal{L}) = \{\text{root}(\ell) \mid \ell \xrightarrow{\mathcal{G}} r \in \mathcal{L}\}$ and $\Sigma_{\text{c}}(\mathcal{L}) = \Sigma \setminus \Sigma_{\text{d}}(\mathcal{L})$.

So in particular, every TRS \mathcal{R} over a standard signature is a set of rewrite lemmas w.r.t. $\mathcal{G}_{\mathcal{R}}$. Note that $\rightarrow_{\mathcal{L}}$ is not a weighted relation (cf. Definition 2.10), since $c\sigma$ might still contain variables. However, the restriction of $\rightarrow_{\mathcal{L}}$ to ground terms is a weighted relation.

Example 9.6. \mathcal{R}_{qs} and $\mathcal{L}_{\text{qs}} = \mathcal{R}_{\text{qs}} \cup \{(9.1)\}$ are sets of rewrite lemmas w.r.t. \mathcal{G}_{qs} .

Throughout this section, we assume a set of generator equations \mathcal{G} and \mathcal{L} always denotes a set of rewrite lemmas w.r.t. \mathcal{G} . Note that the definition of $\rightarrow_{\mathcal{L}}$ is purely syntactic, i.e., a term like $\text{f}(\text{gen}_{\tau}(1))$ cannot be rewritten with a rule

CHAPTER 9. INDUCTION TECHNIQUE

$f(\text{gen}_\tau(1+x)) \rightarrow \dots$ as we have $f(\text{gen}_\tau(1+x))\{x/0\} = f(\text{gen}_\tau(1+0))$, which is not syntactically equal to $f(\text{gen}_\tau(1))$. Similarly, a term like $f(\text{zero})$ cannot be rewritten with $f(\text{gen}_{\text{Nat}}(0)) \rightarrow \dots$, as the definition of $\rightarrow_{\mathcal{L}}$ does not take generator equations into account. Thus, we apply rewrite lemmas *modulo* $\mathcal{G} \cup \mathcal{A}$.

Definition 9.7 (Rewriting Modulo). Let \mathcal{L} be a set of rewrite lemmas and let \mathcal{E} be a set of equations. We define $s \xrightarrow{\mathcal{L}/\mathcal{E}} t$ if $s \equiv_{\mathcal{E}} \circ \xrightarrow{\mathcal{L}} \circ \equiv_{\mathcal{E}} t$. For innermost rewriting, we define $s \xrightarrow{i}_{\mathcal{L}/\mathcal{E}} t$ if there is a context C , a rewrite lemma $\ell \xrightarrow{c'} r \in \mathcal{L}$, and a substitution σ such that $s \equiv_{\mathcal{E}} C[\ell\sigma]$, $C[r\sigma] \equiv_{\mathcal{E}} t$, $c'\sigma \equiv_{\mathcal{A}} c$, and all proper subterms of $\ell\sigma$ are normal forms w.r.t. $\rightarrow_{\mathcal{L}/\mathcal{E}}$. Furthermore, we define $s_0 \xrightarrow{\mathcal{L}/\mathcal{E}}^m s_m$ if $s_0 \xrightarrow{c_1}_{\mathcal{L}/\mathcal{E}} \dots \xrightarrow{c_m}_{\mathcal{L}/\mathcal{E}} s_m$ and $c \equiv_{\mathcal{A}} \sum_{i=1}^m c_i$. If the number of steps m is irrelevant, then we write $s_0 \xrightarrow{\mathcal{L}/\mathcal{E}}^* s_m$ (resp. $s_0 \xrightarrow{\mathcal{L}/\mathcal{E}}^+ s_m$ if $m > 0$). Finally, we lift $\xrightarrow{i}_{\mathcal{L}/\mathcal{E}}$ to $\xrightarrow{i}_{\mathcal{L}/\mathcal{E}}$ analogously.

So in contrast to the integer transition relation (Definition 4.6) and the natural transition relation (Definition 5.2) which also rely on an underlying theory (namely integer arithmetic resp. arithmetic on natural numbers), rewriting modulo is more flexible as the underlying theory is defined by an arbitrary set of equations and $\rightarrow_{\mathcal{L}/\mathcal{E}}$ also allows to rewrite non-ground terms. On the other hand, the integer resp. natural transition relation allows constraints like “ $x > y$ ”, which are not supported by rewriting modulo. In other words, rewriting modulo just allows tests for equality (modulo \mathcal{E}) via the condition “ $s \equiv_{\mathcal{E}} C[\ell\sigma]$ ” in Definition 9.7. Thus, the transition relations from Chapter 4 and Chapter 5 are orthogonal to rewriting modulo.

In the following, we use \rightarrow (resp. \xrightarrow{i}) as a shorthand for $\rightarrow_{\mathcal{L}/\mathcal{G} \cup \mathcal{A}}$ (resp. $\xrightarrow{i}_{\mathcal{L}/\mathcal{G} \cup \mathcal{A}}$) if \mathcal{L} and \mathcal{G} are clear from the context.

Example 9.8. Using \mathcal{L}_{qs} from Example 9.6 we have

$$\text{qs}(\text{cons}^n(\text{zero}, \text{nil})) \xrightarrow{3 \cdot n^2 + 2 \cdot n + 1} \text{cons}^n(\text{zero}, \text{nil})$$

using the rewrite lemma (9.1) since $\text{qs}(\text{cons}^n(\text{zero}, \text{nil})) \equiv_{\mathcal{G}_{\text{qs}} \cup \mathcal{A}} \text{qs}(\text{gen}_{\text{List}}(n))$ and $\text{cons}^n(\text{zero}, \text{nil}) \equiv_{\mathcal{G}_{\text{qs}} \cup \mathcal{A}} \text{gen}_{\text{List}}(n)$.

To reason about rewrite lemmas, it is often useful to normalize terms w.r.t. generator equations. Given a set of generator equations \mathcal{G} , each ground term indeed has a unique normal form w.r.t. \mathcal{G} oriented from left to right.

Lemma 9.1 (Properties of \mathcal{G}). Let Σ be a standard signature, let $\mathcal{G}^f = \{\ell \rightarrow r \mid \ell = r \in \mathcal{G}\}$, and let $\mathcal{G}^b = \{r \rightarrow \ell \mid \ell = r \in \mathcal{G}\}$.

- (1) The relations $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$ and $\rightarrow_{\mathcal{G}^b/\mathcal{A}}$ are well founded.
- (2) The relation $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$ is confluent modulo \mathcal{A} .

9.1. FROM TERM REWRITING TO REWRITE LEMMAS

(3) Every well-typed term $t \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}} \cup \Sigma_{\mathcal{G}})$ has a unique normal form w.r.t. $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$, i.e., $t \downarrow_{\mathcal{G}^f/\mathcal{A}}$ exists.

Proof. We prove (1) – (3) individually.

- (1) Well-foundedness of $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$ follows from Definition 9.4, since the argument of gen_{τ} decreases with each application of an equation and since we excluded mutually recursive types. Well-foundedness of $\rightarrow_{\mathcal{G}^b/\mathcal{A}}$ follows from Definition 9.4, as each rewrite step reduces the number of symbols from $\Sigma_c(\mathcal{R})$.
- (2) The relation $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$ is confluent modulo \mathcal{A} since there are no critical pairs.
- (3) If the type of t is \mathbb{N} , then the claim is trivial. Assume that the type of t is not \mathbb{N} . The term t has a normal form t' which is unique *modulo* \mathcal{A} due to (1) and (2). Since t is well typed and ground and Σ is standard, for each $\pi \in \text{pos}(t)$ with $\text{root}(t|_{\pi}) \in \Sigma_{\mathcal{G}}$ we have $t|_{\pi} \equiv_{\mathcal{A}} \text{gen}_{\tau}(n)$ for some type τ and some $n \in \mathbb{N}$. Since every term of the form $\text{gen}_{\tau}(n)$ is a redex w.r.t. $\rightarrow_{\mathcal{G}^f/\mathcal{A}}$, we get $t'|_{\pi} \in \mathcal{T}(\Sigma)$ by definition of \mathcal{G} . Since t is well typed, its type is not \mathbb{N} , and Σ is standard, t and thus t' does not have further subterms of type \mathbb{N} . Hence, $t'' \equiv_{\mathcal{A}} t'$ implies $t'' = t'$, i.e., t' is unique. \square

So the first step of the technique presented in this chapter is to transform a complexity problem over $\rightarrow_{\mathcal{R}}$ into a complexity problem over $\rightarrow_{\mathcal{L}_{\mathcal{R}}/\mathcal{G}_{\mathcal{R}} \cup \mathcal{A}}$. To this end, we lift our definition of basic terms and “size” to terms that contain generator symbols.

Definition 9.9 ($\mathcal{T}_{\text{basic}}$). We have $f(\mathbf{t}) \in \mathcal{T}_{\text{basic}}(\mathcal{L})$ if $f(\mathbf{t})$ is a well-typed term with $f \in \Sigma_d(\mathcal{L})$ and $\mathbf{t} \subseteq \mathcal{T}(\Sigma_c(\mathcal{L}))$. Moreover, we have $f(\mathbf{t}) \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$ if $f(\mathbf{t})$ is a well-typed term with $f \in \Sigma_d(\mathcal{L})$ and $\mathbf{t} \subseteq \mathcal{T}(\Sigma_c(\mathcal{L}) \cup \Sigma_{\mathcal{G}} \cup \mathbb{N})$.

So in contrast to $\mathcal{T}_{\text{basic}}(\mathcal{L})$, $\mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$ also allows generator symbols and natural numbers. Thus, for each $n \in \mathbb{N}$ we have $\text{qs}(\text{gen}_{\text{List}}(n)) \in \mathcal{T}_{\text{basic}}(\mathcal{L}_{\text{qs}}, \mathcal{G}_{\text{qs}})$, but $\text{qs}(\text{gen}_{\text{List}}(n)) \notin \mathcal{T}_{\text{basic}}(\mathcal{L}_{\text{qs}})$. However, we have

$$\text{qs}(\text{gen}_{\text{List}}(n)) \downarrow_{\mathcal{G}_{\text{qs}}^f} = \text{qs}(\text{cons}^n(\text{zero}, \text{nil})) \in \mathcal{T}_{\text{basic}}(\mathcal{L}_{\text{qs}}, \mathcal{G}_{\text{qs}}) \cap \mathcal{T}_{\text{basic}}(\mathcal{L}_{\text{qs}}).$$

This is not a coincidence, as *every* basic term with generator symbols corresponds to a basic term without generator symbols.

Lemma 9.10. We have $t \downarrow_{\mathcal{G}^f/\mathcal{A}} \in \mathcal{T}_{\text{basic}}(\mathcal{L})$ for all $t \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$.

Proof. The term $t \downarrow_{\mathcal{G}^f/\mathcal{A}}$ exists due to Lemma 9.1 (3). Moreover, $t \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$ clearly implies $t \downarrow_{\mathcal{G}^f/\mathcal{A}} \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$. It remains to show that $t \downarrow_{\mathcal{G}^f/\mathcal{A}}$ does not

CHAPTER 9. INDUCTION TECHNIQUE

contain generator symbols or symbols from $\Sigma_{\mathbb{N}}$, which follows as in the proof of Lemma 9.1 (3). Thus, we get $t \downarrow_{\mathcal{G}^f/\mathcal{A}} \in \mathcal{T}_{\text{basic}}(\mathcal{L})$. \square

The size of a basic terms with generator symbols is defined to be the size of the corresponding basic term without generator symbols.

Definition 9.11 ($\|\cdot\|_{\mathcal{G}}$). We define $\|t\|_{\mathcal{G}} = \|t \downarrow_{\mathcal{G}^f/\mathcal{A}}\|_t$ for all $t \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$.

Thus, we have, e.g.,

$$\|\text{qs}(\text{gen}_{\text{List}}(n))\|_{\mathcal{G}_{\text{qs}}} = \|\text{qs}(\text{gen}_{\text{List}}(n)) \downarrow_{\mathcal{G}_{\text{qs}}^f}\|_t = \|\text{qs}(\text{cons}^n(\text{zero}, \text{nil}))\|_t = 2n + 2.$$

Based on these notions of basic terms and size, we can define the canonical complexity problem of a set of rewrite lemmas in a natural way.

Definition 9.12 (Canonical Complexity Problem). The *canonical complexity problem* of \mathcal{L} and \mathcal{G} is $cp(\mathcal{L}, \mathcal{G}) = (\mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G}), \rhd, \|\cdot\|_{\mathcal{G}})$. Similarly, the *innermost canonical complexity problem* of \mathcal{L} and \mathcal{G} is $cp_i(\mathcal{L}, \mathcal{G}) = (\mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G}), \rhd_i, \|\cdot\|_{\mathcal{G}})$.

This gives rise to the first processor of the current chapter.

Theorem 9.13 (From Term Rewriting to Rewrite Lemmas). *Let \mathcal{R} be a well-typed TRS over a standard signature Σ . Then the processor mapping $cp(\mathcal{R})$ to $cp(\mathcal{R}, \mathcal{G}_{\mathcal{R}})$ is sound for lower bounds.*

Proof. Since $\mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$ just contains well-typed ground terms, we only have to regard rewrite sequences on well-typed ground terms. Thus, consider a rewrite step $s \xrightarrow{\ell} t$ where s and t are well typed and ground. We prove $s \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} \xrightarrow{\ell}_{\mathcal{R}} t \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}$. Then the claim follows by Lemma 9.10 and Definition 9.11.

Let σ and $\ell \xrightarrow{\ell} r$ be the substitution and rule which are used for the rewrite step, i.e., we have $C[\ell\sigma] \equiv_{\mathcal{G}_{\mathcal{R}} \cup \mathcal{A}} s$ and $C[r\sigma] \equiv_{\mathcal{G}_{\mathcal{R}} \cup \mathcal{A}} t$ for some context C . Thus, we get $C[\ell\sigma] \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = s \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}$ and $C[r\sigma] \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = t \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}$ by Lemma 9.1 (3). Let $\sigma' = \{x/x\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} \mid x \in \text{dom}(\sigma)\}$. Since ℓ and r do not contain generator symbols, we get $\ell\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = \ell\sigma'$ and $r\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = r\sigma'$. Since Σ is standard and the only argument of generator symbols has type \mathbb{N} , generator symbols cannot occur above symbols from Σ in well-typed terms. So in particular, $\text{root}(\ell) \in \Sigma$ and $C[\ell\sigma] \equiv_{\mathcal{G}_{\mathcal{R}} \cup \mathcal{A}} s$ implies that there are no generator symbols above \square in C . Thus we get

$$\begin{aligned} s \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} &= C[\ell\sigma] \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} \\ &= C \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} [\ell\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}] \text{ as there are no gen. symbols above } \square \text{ in } C \\ &= C \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} [\ell\sigma'] \quad \text{as } \ell\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = \ell\sigma' \end{aligned}$$

9.1. FROM TERM REWRITING TO REWRITE LEMMAS

and

$$\begin{aligned}
 t\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} &= C[r\sigma]\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} \\
 &= C\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}[r\sigma\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}] \text{ as there are no gen. symbols above } \square \text{ in } C \\
 &= C\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}[r\sigma'] \quad \text{as } r\sigma\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} = r\sigma'.
 \end{aligned}$$

Hence, we have $s\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}} \xrightarrow{\mathcal{R}} t\downarrow_{\mathcal{G}_{\mathcal{R}}^f/\mathcal{A}}$, as desired. \square

Note that the processor from Theorem 9.13 is not sound for upper bounds, as the definition of complexity for rewrite lemmas just considers rewrite sequences starting with basic *ground* terms, whereas the complexity of TRSs is defined in terms of arbitrary derivations starting with basic terms. Thus, the runtime complexity of the TRS $\mathcal{R} = \{f(x) \xrightarrow{1} f(x)\}$ is unbounded, whereas we have $\text{rc}_{cp(\mathcal{R}, \mathcal{G}_{\mathcal{R}})}(n) \in \mathcal{O}(1)$ due to the absence of basic ground terms.

9.2 Speculating Conjectures

We now show how to speculate *conjectures* for a set of rewrite lemmas. For a defined symbol f of type $\tau_1 \times \dots \times \tau_m \rightarrow \tau$ with simply structured types τ_1, \dots, τ_m , our goal is to speculate a *conjecture*.

Definition 9.14 (Conjecture). A rule

$$s = f(\text{gen}_{\tau_1}(s_1), \dots, \text{gen}_{\tau_m}(s_m)) \xrightarrow{\circ n}^? t$$

is called a *conjecture* for \mathcal{L} if s and t are well typed, $n \in \mathcal{V}(s)$, $f \in \Sigma_d(\mathcal{L})$, and $t \in \mathcal{T}(\Sigma \cup \Sigma_{\mathbb{N}} \cup \Sigma_{\mathcal{G}}, \mathcal{V}(s))$.

The variable n is called the *induction variable* of the conjecture.

Here, “ $\circ n$ ” indicates that we will try to prove the *validity* of the conjecture (cf. Definition 9.15) by induction on n later on. While lemma speculation was investigated in inductive theorem proving and verification since decades [23], we want to find lemmas of a special form in order to extract suitable lower bounds from their induction proofs.

When speculating conjectures, we take the dependencies between defined symbols into account. If $f \sqsupseteq g$ and $g \not\sqsupseteq f$, then we first generate a conjecture $s = g(\dots) \xrightarrow{\circ n}^? t$ for g . Afterwards, we prove its validity by induction on n in Section 9.3. If this proof attempt succeeds, then we infer a cost function $c \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}(s))$ in Section 9.4. This cost function describes a lower bound for the cost of the corresponding evaluations. Then the analyzed set of rewrite lemmas can be extended by $g(\dots) \xrightarrow{c} t$ and this new rewrite lemma can be used when generating a conjecture for f afterwards.

Definition 9.15 (Validity of Conjectures). A conjecture $s \xrightarrow{\circ n}^? t$ is *valid* for \mathcal{L} if $s\sigma \rightarrow^* t\sigma$ holds for all $\sigma : \mathcal{V}(s) \rightarrow \mathbb{N}$.

For instance, the conjecture $\text{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{\circ n}^? \text{gen}_{\text{List}}(n)$ is valid for \mathcal{R}_{qs} , as $n\sigma = m \in \mathbb{N}$ implies

$$\begin{aligned} \text{qs}(\text{gen}_{\text{List}}(n))\sigma &\equiv_{\mathcal{G}_{\text{qs}} \cup \mathcal{A}} \\ \text{qs}(\text{cons}^m(\text{zero}, \text{nil})) &\xrightarrow{\mathcal{R}_{\text{qs}}}^+ \text{cons}^m(\text{zero}, \text{nil}) \\ &\equiv_{\mathcal{G}_{\text{qs}} \cup \mathcal{A}} \text{gen}_{\text{List}}(n)\sigma, \end{aligned}$$

i.e., $\text{qs}(\text{gen}_{\text{List}}(n))\sigma \rightarrow^* \text{gen}_{\text{List}}(n)\sigma$.

Of course, our algorithm for the speculation of conjectures is just one possible implementation for this task. The soundness of our approach (i.e., the correctness of the theorems and lemmas in Sections 9.3 to 9.5) is independent of the specific implementation that is used for the speculation of conjectures.

To speculate a conjecture for a defined symbol f , we first generate *sample conjectures* that describe the effect of applying f to specific arguments. To

9.2. SPECULATING CONJECTURES

obtain them, we narrow $f(\text{gen}_{\tau_1}(n_1), \dots, \text{gen}_{\tau_k}(n_k))$ where $n_1, \dots, n_k \in \mathcal{V}$, using the available rewrite lemmas. Thereby, we take the generator equations and arithmetic into account. This narrowing corresponds to a case analysis over the possible derivations.

Definition 9.16 (Narrowing Modulo). Let \mathcal{L} be a set of rewrite lemmas and let \mathcal{E} be a set of equations. We have $s \xrightarrow{\mathcal{L}/\mathcal{E}} t$ if σ is a well-typed substitution such that $s\sigma \rightarrow_{\mathcal{L}/\mathcal{E}} t$.

So in contrast to rewriting modulo equations, narrowing modulo equations essentially performs equational unification instead of equational matching when applying rewrite lemmas: First we instantiate variables in s via σ , then we match a left-hand side of a rewrite lemma to a subterm of some $s' \equiv_{\mathcal{E}} s\sigma$ when performing the rewrite step $s\sigma \rightarrow_{\mathcal{L}/\mathcal{E}} t$. However, in contrast to the narrowing relation for term rewriting (Definition 7.5), we do not require the unifier to be an mgu. The reason is that the question whether an mgu exists depends on the set of equations \mathcal{E} and, for our use case, there is no need to restrict narrowing modulo to mgus.

In the following, we use “ $s \rightsquigarrow t$ ” as a shorthand for “ $s \rightsquigarrow_{\mathcal{L}/\mathcal{G} \cup \mathcal{A}} t$ ” if \mathcal{L} and \mathcal{G} are clear from the context. For instance, we have $\text{qs}(\text{gen}_{\text{List}}(n)) \rightsquigarrow \text{gen}_{\text{List}}(0)$ using the rewrite lemmas $\mathcal{L} = \mathcal{R}_{\text{qs}}$, the corresponding generator equations \mathcal{G}_{qs} , the substitution $\sigma = \{n/0\}$, and the rule α_0 since we have

$$\text{qs}(\text{gen}_{\text{List}}(n))\sigma = \text{qs}(\text{gen}_{\text{List}}(0)) \equiv_{\mathcal{G}} \text{lhs}(\alpha_0) \rightarrow_{\alpha_0} \text{rhs}(\alpha_0) \equiv_{\mathcal{G}} \text{gen}_{\text{List}}(0).$$

Although checking $s\sigma \equiv_{\mathcal{G} \cup \mathcal{A}} s'\sigma$ is undecidable for general equations \mathcal{G} , equational unification is decidable in quadratic time for the generator equations of Definition 9.4 [98]. Moreover, arithmetic terms s and t can be unified by rearranging the equality $s = t$ to the form $x = q$ where $x \in \mathcal{V}$ and $q \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V} \setminus \{x\})$. Thus, due to the restricted form of the generator equations in Definition 9.4, the required narrowing works reasonably efficient in practice.

Example 9.17 (Narrowing). In Example 9.1 we have $\text{qs} \sqsupseteq \text{low}$ and $\text{qs} \sqsupseteq \text{high}$. Assume that we obtained \mathcal{L}'_{qs} by extending \mathcal{R}_{qs} by the rewrite lemmas

$$\begin{aligned} \beta_0 : \quad & \text{low}(\text{gen}_{\text{Nat}}(0), \text{gen}_{\text{List}}(n)) \xrightarrow{3 \cdot n + 1} \text{gen}_{\text{List}}(0) \text{ and} \\ \beta_1 : \quad & \text{high}(\text{gen}_{\text{Nat}}(0), \text{gen}_{\text{List}}(n)) \xrightarrow{3 \cdot n + 1} \text{gen}_{\text{List}}(n). \end{aligned}$$

Then the narrowing tree in Figure 9.1 can be generated to find sample conjectures for qs . The arrows are labeled with the used rules and substitutions. For the sake of clarity, some arrows correspond to *several* narrowing steps. The goal is to get representative rewrite sequences, not to cover all reductions. Hence, we stop constructing the tree after some steps and choose suitable narrowings heuristically.⁴

CHAPTER 9. INDUCTION TECHNIQUE

After constructing a narrowing tree with root $s = f(\dots)$, we collect sample conjectures $s\sigma \xrightarrow{\odot_d!} t$. Here, t results from a leaf q of the tree which is in \rightsquigarrow -normal form by normalizing q w.r.t. the generator equations \mathcal{G} applied from right to left (which terminates due to Lemma 9.1). Thus, terms from $\mathcal{T}(\Sigma, \mathcal{V})$ are rewritten to generator symbols with polynomials as arguments. Moreover, σ is the substitution on the path from the root to q , and d is the number of applications of recursive f -rules on the path (the *recursion depth*). A rule $f(\dots) \rightarrow r$ is *recursive* if r contains a symbol g with $g \supseteq f$.

Example 9.18 (Sample Conjectures). In Example 9.17, the set S of sample conjectures contains⁵

$$\begin{aligned} \text{qs}(\text{gen}_{\text{List}}(n_1))\{n_1/0\} &\xrightarrow{\odot_0!} \text{gen}_{\text{List}}(0), \\ \text{qs}(\text{gen}_{\text{List}}(n_1))\{n_1/1\} &\xrightarrow{\odot_1!} \text{gen}_{\text{List}}(1), \text{ and} \\ \text{qs}(\text{gen}_{\text{List}}(n_1))\{n_1/2\} &\xrightarrow{\odot_2!} \text{gen}_{\text{List}}(2). \end{aligned} \quad (9.6)$$

The sequence from $\text{qs}(\text{gen}_{\text{List}}(n_1))$ to nil does not use recursive qs -rules. Hence, its recursion depth is 0 and the \rightsquigarrow -normal form nil rewrites to $\text{gen}_{\text{List}}(0)$ when applying the generator equation (9.2) from right to left. The sequence from $\text{qs}(\text{gen}_{\text{List}}(n_1))$ to $\text{cons}(\text{zero}, \text{nil})$ (resp. $\text{cons}^2(\text{zero}, \text{nil})$) uses the recursive qs -rule α_1 once (resp. twice), i.e., it has recursion depth 1 (resp. 2). Moreover, its \rightsquigarrow -normal form rewrites to $\text{gen}_{\text{List}}(1)$ (resp. $\text{gen}_{\text{List}}(2)$) when applying \mathcal{G}_{qs} from right to left.

Now the goal is to find a maximal subset of these sample conjectures whose elements are suitable for generalization. Then, this subset is used to speculate a general conjecture (whose validity must be proved afterwards).

For a narrowing tree with root s , let S_{\max} be a maximal subset of all sample conjectures such that all $s\sigma \xrightarrow{\odot_d!} t, s\sigma' \xrightarrow{\odot_{d'}!} t' \in S_{\max}$ are identical up to the occurring natural numbers and variable names. For instance, the sample conjectures (9.6) are identical up to the occurring numbers. To obtain a general conjecture, we replace all numbers in the left-hand and right-hand sides of sample conjectures by polynomials. In our example, we want to speculate a conjecture of the form $\text{qs}(\text{gen}_{\text{List}}(\text{pol}^{\text{left}})) \xrightarrow{\odot_n?} \text{gen}_{\text{List}}(\text{pol}^{\text{right}})$. Here, pol^{left} and $\text{pol}^{\text{right}}$ are polynomials in n , where n stands for the recursion depth. This facilitates the proof of the resulting conjecture by induction on n .

For any term q , let $\text{pos}_{\mathbb{N}}(q) = \{\pi \in \text{pos}(q) \mid q|_{\pi} \in \mathbb{N}\}$. Then for every position $\pi \in \text{pos}_{\mathbb{N}}(s\sigma)$ (resp. $\pi \in \text{pos}_{\mathbb{N}}(t)$) with $s\sigma \xrightarrow{\odot_d!} t \in S_{\max}$, we search for a polynomial $\text{pol}_{\pi}^{\text{left}}$ (resp. $\text{pol}_{\pi}^{\text{right}}$). To obtain these polynomials, for every

⁴In our implementation, we construct the narrowing tree breadth-first. Here, we prefer narrowing with rules with non-constant costs (i.e., narrowing with the rules from the initial TRS is only done for those subterms where no “meta rule” is applicable).

⁵We always simplify arithmetic expressions in terms and substitutions, e.g., the substitution $\{n_1/0 + 1\}$ in the second sample conjecture is simplified to $\{n_1/1\}$.

9.2. SPECULATING CONJECTURES

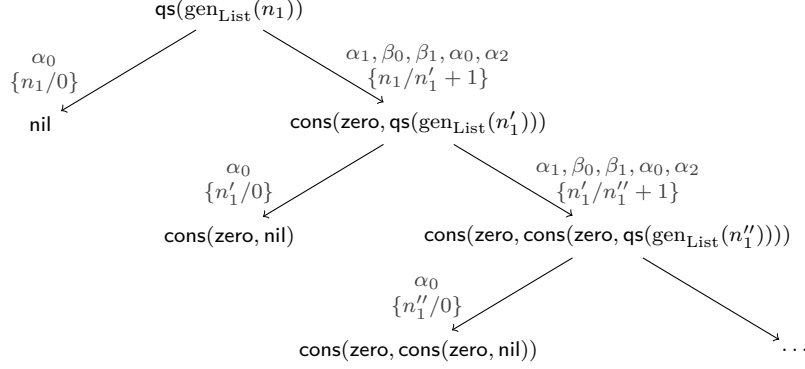


Figure 9.1: Narrowing Tree

$s\sigma \xrightarrow{\odot d}! t \in S_{\max}$ we generate the constraints

$$\begin{aligned} \text{pol}_{\pi}^{\text{left}}(d) &= s\sigma|_{\pi} \quad \text{for all } \pi \in \text{pos}_{\mathbb{N}}(s\sigma) \text{ and} \\ \text{pol}_{\pi}^{\text{right}}(d) &= t|_{\pi} \quad \text{for all } \pi \in \text{pos}_{\mathbb{N}}(t). \end{aligned} \quad (9.7)$$

Here, $\text{pol}_{\pi}^{\text{left}}$ and $\text{pol}_{\pi}^{\text{right}}$ are polynomials with abstract coefficients. If one searches for polynomials of degree e , then the polynomials have the form $c_0 + c_1 \cdot n + \dots + c_e \cdot n^e$ and the constraints in (9.7) are linear diophantine equations over the unknown coefficients $c_i \in \mathbb{N}$.⁶ These equations are easily solved automatically. Finally, the generalized speculated conjecture is obtained from $s\sigma \xrightarrow{\odot d}! t$ by replacing $s\sigma|_{\pi}$ with $\text{pol}_{\pi}^{\text{left}}$ for every $\pi \in \text{pos}_{\mathbb{N}}(s\sigma)$ and by replacing $t|_{\pi}$ with $\text{pol}_{\pi}^{\text{right}}$ for every $\pi \in \text{pos}_{\mathbb{N}}(t)$.

Example 9.19 (Speculating Conjectures). In Example 9.17, we narrowed $s = \text{qs}(\text{gen}_{\text{List}}(n_1))$ and obtained the set S_{\max} with the sample conjectures (9.6), cf. Example 9.18. For each $s\sigma \xrightarrow{\odot d}! t \in S_{\max}$, we have $\text{pos}_{\mathbb{N}}(s\sigma) = \{1.1\}$ and $\text{pos}_{\mathbb{N}}(t) = \{1\}$. Hence, from the sample conjecture

$$\text{qs}(\text{gen}_{\text{List}}(0)) \xrightarrow{\odot 0}! \text{gen}_{\text{List}}(0),$$

we obtain the constraints

$$\begin{aligned} \text{pol}_{1.1}^{\text{left}}(d) &= \text{pol}_{1.1}^{\text{left}}(0) = \text{qs}(\text{gen}_{\text{List}}(0))|_{1.1} = 0 \text{ and} \\ \text{pol}_1^{\text{right}}(d) &= \text{pol}_1^{\text{right}}(0) = \text{gen}_{\text{List}}(0)|_1 = 0. \end{aligned}$$

Similarly, from the two other sample conjectures we get

$$\text{pol}_{1.1}^{\text{left}}(1) = \text{pol}_1^{\text{right}}(1) = 1 \quad \text{and} \quad \text{pol}_{1.1}^{\text{left}}(2) = \text{pol}_1^{\text{right}}(2) = 2.$$

⁶ In the constraints (9.7), n is instantiated by an actual number d . Thus, if $\text{pol}_{\pi}^{\text{left}} = c_0 + c_1 \cdot n + \dots + c_e \cdot n^e$, then $\text{pol}_{\pi}^{\text{left}}(d)$ is a *linear* polynomial over the unknowns c_0, \dots, c_e .

CHAPTER 9. INDUCTION TECHNIQUE

When using $\text{pol}_{1.1}^{\text{left}} = c_0 + c_1 \cdot n + c_2 \cdot n^2$ and $\text{pol}_1^{\text{right}} = d_0 + d_1 \cdot n + d_2 \cdot n^2$ with the abstract coefficients $c_0, \dots, c_2, d_0, \dots, d_2$, the solution $c_0 = c_2 = d_0 = d_2 = 0$, $c_1 = d_1 = 1$ (i.e., $\text{pol}_{1.1}^{\text{left}} = n$ and $\text{pol}_1^{\text{right}} = n$) is easily found automatically. The resulting speculated conjecture is

$$\begin{array}{ccc} \text{qs}(\text{gen}_{\text{List}}(\text{pol}_{1.1}^{\text{left}})) & \xrightarrow{\text{O}_n^?} & \text{gen}_{\text{List}}(\text{pol}_1^{\text{right}}), \text{ i.e.,} \\ \text{qs}(\text{gen}_{\text{List}}(n)) & \xrightarrow{\text{O}_n^?} & \text{gen}_{\text{List}}(n). \end{array}$$

If S_{\max} contains sample conjectures with e different recursion depths, then there are unique polynomials of at most degree $e - 1$ satisfying the constraints (9.7). The reason is that the sample conjectures give rise to e constraints for the unknown coefficients of the polynomial, and a polynomial of degree $e - 1$ has e coefficients.

Example 9.20 (Several Variables in Conjecture). We consider the TRS from Example 8.3 to show how to speculate conjectures with *several* variables. Narrowing $s = \text{add}(\text{gen}_{\text{Nat}}(n_1), \text{gen}_{\text{Nat}}(n_2))$ yields the sample conjectures

$$\begin{array}{ccc} \text{add}(\text{gen}_{\text{Nat}}(n_1), \text{gen}_{\text{Nat}}(n_2))\{n_1/0\} & \xrightarrow{\text{O}_0^!} & \text{gen}_{\text{Nat}}(n_2), \\ \text{add}(\text{gen}_{\text{Nat}}(n_1), \text{gen}_{\text{Nat}}(n_2))\{n_1/1\} & \xrightarrow{\text{O}_1^!} & \text{gen}_{\text{Nat}}(n_2 + 1), \\ \text{add}(\text{gen}_{\text{Nat}}(n_1), \text{gen}_{\text{Nat}}(n_2))\{n_1/2\} & \xrightarrow{\text{O}_2^!} & \text{gen}_{\text{Nat}}(n_2 + 2), \text{ and} \\ \text{add}(\text{gen}_{\text{Nat}}(n_1), \text{gen}_{\text{Nat}}(n_2))\{n_1/3\} & \xrightarrow{\text{O}_3^!} & \text{gen}_{\text{Nat}}(n_2 + 3). \end{array}$$

For the last three sample conjectures $s\sigma \xrightarrow{\text{O}_d^!} t$, the only number in $s\sigma$ is at position 1.1 and the polynomial $\text{pol}_{1.1}^{\text{left}} = n$ satisfies the constraint $\text{pol}_{1.1}^{\text{left}}(d) = s\sigma|_{1.1}$. Moreover, the only number in t is at position 1.2 and the polynomial $\text{pol}_{1.2}^{\text{right}} = n$ satisfies $\text{pol}_{1.2}^{\text{right}}(d) = t|_{1.2}$. Thus, we speculate the conjecture $\text{add}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{\text{O}_n^?} \text{gen}_{\text{Nat}}(n_2 + n)$.

Example 9.21 (Larger Coefficients of Polynomials). The following rules illustrates how we speculate conjectures where the coefficients of the polynomials are larger than 1.

$$\text{half}(\text{zero}) \rightarrow \text{zero} \qquad \text{half}(\text{succ}(\text{succ}(x))) \rightarrow \text{succ}(\text{half}(x))$$

By narrowing $s = \text{half}(\text{gen}_{\text{Nat}}(n_1))$, we obtain the sample conjectures

$$\begin{array}{ccc} \text{half}(\text{gen}_{\text{Nat}}(n_1))\{n_1/0\} & \xrightarrow{\text{O}_0^!} & \text{gen}_{\text{Nat}}(0), \\ \text{half}(\text{gen}_{\text{Nat}}(n_1))\{n_1/2\} & \xrightarrow{\text{O}_1^!} & \text{gen}_{\text{Nat}}(1), \text{ and} \\ \text{half}(\text{gen}_{\text{Nat}}(n_1))\{n_1/4\} & \xrightarrow{\text{O}_2^!} & \text{gen}_{\text{Nat}}(2). \end{array}$$

For these sample conjectures $s\sigma \xrightarrow{\text{O}_d^!} t$, the only numbers in $s\sigma$ (resp. t) are at position 1.1 (resp. at position 1). The polynomial $\text{pol}_{1.1}^{\text{left}} = 2 \cdot n$ satisfies the constraint $\text{pol}_{1.1}^{\text{left}}(d) = s\sigma|_{1.1}$ and the polynomial $\text{pol}_1^{\text{right}} = n$ satisfies $\text{pol}_1^{\text{right}}(d) = t|_1$. Hence, we obtain $\text{half}(\text{gen}_{\text{Nat}}(2 \cdot n)) \xrightarrow{\text{O}_n^?} \text{gen}_{\text{Nat}}(n)$.

9.2. SPECULATING CONJECTURES

Algorithm 4 summarizes our method to speculate conjectures for a set of rewrite lemmas \mathcal{L} . Note that we have $\text{pos}_{\mathbb{N}}(s\sigma) = \text{pos}_{\mathbb{N}}(s\sigma')$ (resp. $\text{pos}_{\mathbb{N}}(t) = \text{pos}_{\mathbb{N}}(t')$) and $s\sigma$ and $s\sigma'$ (resp. t and t') are identical up to variable names and the positions $\text{pos}_{\mathbb{N}}(s\sigma)$ (resp. $\text{pos}_{\mathbb{N}}(t)$) for each $s\sigma \xrightarrow{\odot d} t, s\sigma' \xrightarrow{\odot d'} t' \in S_{\max}$ as the sample conjectures in S_{\max} are suitable for generalization. Hence, the choice in Step 4 does not affect the result of our algorithm. In Step 8, we use the SMT solver Z3 in our experiments (cf. Chapter 12). The latest version of AProVE uses SMTInterpol [37] instead. As both AProVE and SMTInterpol are implemented in Java, invoking SMTInterpol from AProVE causes less overhead than invoking Z3.

Algorithm 4 Speculating Conjectures

1. Let $f \in \Sigma_d(\mathcal{L})$ be a minimal symbol (w.r.t. \sqsubseteq) which has not been analyzed
 2. Compute a narrowing tree for $s = f(\text{gen}_{\tau_1}(n_1), \dots, \text{gen}_{\tau_k}(n_k))$
 3. Obtain a maximal set of sample conjectures S_{\max} which is suitable for generalization from this narrowing tree and let $e = |S_{\max}|$
 4. Choose some $s\sigma \xrightarrow{\odot d} t \in S_{\max}$
 5. For each $\pi \in \text{pos}_{\mathbb{N}}(s\sigma)$ resp. $\pi \in \text{pos}_{\mathbb{N}}(t)$
 Set $\text{pol}_{\pi}^{\text{left}} := c_0^{\pi} + c_1^{\pi} \cdot n + c_2^{\pi} \cdot n^2 + \dots + c_{e-1}^{\pi} \cdot n^{e-1}$
 resp. $\text{pol}_{\pi}^{\text{right}} := d_0^{\pi} + d_1^{\pi} \cdot n + d_2^{\pi} \cdot n^2 + \dots + d_{e-1}^{\pi} \cdot n^{e-1}$
 6. Set $\varphi := \text{true}$
 7. For each $s\sigma' \xrightarrow{\odot d'} t' \in S_{\max}$ and each $\pi \in \text{pos}_{\mathbb{N}}(s\sigma')$ resp. $\pi \in \text{pos}_{\mathbb{N}}(t')$
 Set $\varphi := \varphi \wedge \text{pol}_{\pi}^{\text{left}}(d') = s\sigma'|_{\pi}$
 resp. $\varphi := \varphi \wedge \text{pol}_{\pi}^{\text{right}}(d') = t'|_{\pi}$
 8. Search for a model σ of φ using standard SMT solvers
 9. Let ℓ result from $s\sigma$ by replacing $s\sigma|_{\pi}$ with $\text{pol}_{\pi}^{\text{left}}\sigma$ for each $\pi \in \text{pos}_{\mathbb{N}}(s\sigma)$
 10. Let r result from t by replacing $t|_{\pi}$ with $\text{pol}_{\pi}^{\text{right}}\sigma$ for each $\pi \in \text{pos}_{\mathbb{N}}(t)$
 11. Return $\ell \xrightarrow{\odot n} r$
-

9.3 Proving Conjectures

Now we show how to prove the validity of speculated conjectures, cf. Definition 9.15. To prove validity of a conjecture $s \xrightarrow{\circ n}^? t$ by induction, we use rewriting with \rightarrow . In the induction step, we try to reduce $s\{n/n+1\}$ to $t\{n/n+1\}$, where one may use the rule $\alpha_{\text{ih}} = s \rightarrow t$ as induction hypothesis. Here, the induction variable n must not be instantiated and the remaining variables in α_{ih} may only be instantiated by an increasing substitution. A substitution σ is *increasing* if $\mathcal{A} \models x\sigma \geq x$ holds for all $x \in \text{dom}(\sigma)$. For example, the substitution $\sigma = \{x/x+y\}$ is increasing because $\mathcal{A} \models x+y \geq x$. The restriction to increasing substitutions results in induction proofs that are particularly suitable for inferring runtimes of valid conjectures. More precisely, increasing substitutions are necessary to ensure the soundness of the recurrence equations that we will construct for lower bounds in Section 9.4.

Thus, for any rule $\ell \rightarrow r$ containing only variables of type \mathbb{N} and any $n \in \mathcal{V}$, let $s \mapsto_{\ell \rightarrow r, n} t$ if there exist an increasing substitution σ with $n\sigma = n$ and a context C such that $s \equiv_{\mathcal{G} \cup \mathcal{A}} C[\ell\sigma]$ and $C[r\sigma] \equiv_{\mathcal{G} \cup \mathcal{A}} t$. Moreover, we define $q \xrightarrow{\ell \rightarrow r, n} p$ if $q \xrightarrow{\ell} p$ or $c = 0$ and $q \mapsto_{\ell \rightarrow r, n} p$ (and we lift $\xrightarrow{\ell \rightarrow r, n}$ to $\xrightarrow{\ell \rightarrow r, n}^*$ analogously to $\xrightarrow{\ell}$, cf. Definition 9.7). Theorem 9.22 shows which rewrite sequences are needed to prove a conjecture $s \xrightarrow{\circ n}^? t$ by induction on n .

Theorem 9.22 (Proving Conjectures). *Let $s \xrightarrow{\circ n}^? t$ be a conjecture for \mathcal{L} . If $s\{n/0\} \xrightarrow{i\ell}^* t\{n/0\}$ and $s\{n/n+1\} \xrightarrow{i\delta}^+_{s \rightarrow t, n} t\{n/n+1\}$ for some $i\ell, i\delta \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}(s))$, then the conjecture $s \xrightarrow{\circ n}^? t$ is valid for \mathcal{L} .*

Proof. We prove Theorem 9.22 together with Theorem 9.26 in Section 9.4. \square

Example 9.23 (Proof of Conjecture for **qs**). We continue Example 9.19. To prove the conjecture $\text{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{\circ n}^? \text{gen}_{\text{List}}(n)$, in the induction base we show $\text{qs}(\text{gen}_{\text{List}}(0)) \rightarrow \text{gen}_{\text{List}}(0)$ and in the induction step, we obtain

$$\begin{array}{rcl} & & \text{qs}(\text{gen}_{\text{List}}(n+1)) \\ & \rightarrow^* & \text{nil} ++ \text{cons}(\text{zero}, \text{qs}(\text{gen}_{\text{List}}(n))) \\ \mapsto_{\text{qs}(\text{gen}_{\text{List}}(n)) \rightarrow \text{gen}_{\text{List}}(n), n} & & \text{nil} ++ \text{cons}(\text{zero}, \text{gen}_{\text{List}}(n)) \\ & \rightarrow & \text{gen}_{\text{List}}(n+1). \end{array}$$

The following example illustrates why one may have to instantiate non-induction variables in the induction hypothesis when proving conjectures.

Example 9.24 (Instantiating Non-Induction Variables). Consider the TRS from Example 8.3. Recall that we speculated the conjecture

$$\text{add}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{\circ n}^? \text{gen}_{\text{Nat}}(n_2 + n)$$

9.3. PROVING CONJECTURES

in Example 9.20. To prove this conjecture, in the induction base we have

$$\mathbf{add}(\mathbf{gen}_{\mathbf{Nat}}(0), \mathbf{gen}_{\mathbf{Nat}}(n_2)) \stackrel{1}{\sim} \mathbf{gen}_{\mathbf{Nat}}(n_2).$$

In the induction step, we obtain

$$\mathbf{add}(\mathbf{gen}_{\mathbf{Nat}}(n+1), \mathbf{gen}_{\mathbf{Nat}}(n_2)) \stackrel{1}{\sim} \mathbf{add}(\mathbf{gen}_{\mathbf{Nat}}(n), \mathbf{gen}_{\mathbf{Nat}}(n_2+1)).$$

To apply the induction hypothesis

$$\alpha_{\text{ih}} : \mathbf{add}(\mathbf{gen}_{\mathbf{Nat}}(n), \mathbf{gen}_{\mathbf{Nat}}(n_2)) \rightarrow \mathbf{gen}_{\mathbf{Nat}}(n_2+n),$$

we therefore have to instantiate the non-induction variable n_2 by n_2+1 . Clearly, this is an increasing substitution since $\mathcal{A} \models n_2+1 \geq n_2$. Thus, the proof of the induction step continues with

$$\begin{aligned} \mathbf{add}(\mathbf{gen}_{\mathbf{Nat}}(n), \mathbf{gen}_{\mathbf{Nat}}(n_2+1)) &\mapsto_{\alpha_{\text{ih}}, n} \mathbf{gen}_{\mathbf{Nat}}((n_2+1)+n) \\ &\equiv_{\mathcal{A}} \mathbf{gen}_{\mathbf{Nat}}(n_2+(n+1)). \end{aligned}$$

9.4 Inferring Bounds for Valid Conjectures

For a valid conjecture $s \xrightarrow{\circ n}^? t$, we now show how to infer a lower bound on the cost of the corresponding rewrite sequences, i.e., how to generate a cost function $c \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}(s))$ such that we obtain a rewrite lemma $s \xrightarrow{c} t$. More precisely, we show that one can infer a suitable bound from the induction proof of a conjecture $s \xrightarrow{\circ n}^? t$. Assume that we proved validity of $s \xrightarrow{\circ n}^? t$ as in Theorem 9.22 where the induction hypothesis $s \rightarrow t$ was applied $i\hbar$ times. Then we get the following recurrence equations for c :

$$c\{n/0\} = i\ell \quad \text{and} \quad c\{n/n+1\} = i\hbar \cdot c + i\mathfrak{s} \quad (9.8)$$

Here, the cost of the rewrite sequences which are covered by the induction hypothesis $s \rightarrow t$ are taken into account by the addend $i\hbar \cdot c$. When applying $s \rightarrow t$, s and t are instantiated by an increasing substitution σ . By the induction hypothesis, each rewrite sequence $s\sigma \rightarrow^* t\sigma$ has at least cost $c\sigma$. Since σ is *increasing*, we have $x\sigma \geq x$ for all $x \in \mathcal{V}(s)$. As $c \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}(s))$ is weakly monotonically increasing, this implies $c\sigma \geq c$. Thus, c is a lower bound for the costs of the reduction $s\sigma \rightarrow^* t\sigma$. Hence, the restriction to weakly monotonic functions c and increasing substitutions σ allows us to underapproximate $c\sigma$ by c in (9.8), resulting in recurrence equations that are suitable for automation. By solving the recurrence equations (9.8), we can now compute c explicitly.

Lemma 9.25 (Solving (9.8) Explicitly). *Let c be defined as follows:*

$$c = i\hbar^n \cdot i\ell + \sum_{i=0}^{n-1} i\hbar^{n-1-i} \cdot i\mathfrak{s}\{n/i\}.$$

Then c satisfies (9.8).

Proof. We use induction on n . We obtain $c\{n/0\} = i\hbar^0 \cdot i\ell = i\ell$, as required in (9.8). Similarly,

$$\begin{aligned} c\{n/n+1\} &= i\hbar^{n+1} \cdot i\ell + \sum_{i=0}^n i\hbar^{n-i} \cdot i\mathfrak{s}\{n/i\} \\ &= i\hbar^{n+1} \cdot i\ell + \left(\sum_{i=0}^{n-1} i\hbar^{n-i} \cdot i\mathfrak{s}\{n/i\} \right) + i\mathfrak{s} \\ &= i\hbar^{n+1} \cdot i\ell + i\hbar \cdot \left(\sum_{i=0}^{n-1} i\hbar^{n-1-i} \cdot i\mathfrak{s}\{n/i\} \right) + i\mathfrak{s} \\ &= i\hbar \cdot (i\hbar^n \cdot i\ell + \sum_{i=0}^{n-1} i\hbar^{n-1-i} \cdot i\mathfrak{s}\{n/i\}) + i\mathfrak{s} \\ &= i\hbar \cdot c + i\mathfrak{s}, \end{aligned}$$

as in (9.8). □

Using the explicit form of c from Lemma 9.25, we can obtain a rewrite lemma from a valid conjecture.

9.4. INFERRING BOUNDS FOR VALID CONJECTURES

Theorem 9.26 (Explicit Runtime of Valid Conjectures). *Let $s \xrightarrow{\circ n}^? t$ be a conjecture with an induction proof as in Theorem 9.22. Then the processor mapping $cp(\mathcal{L}, \mathcal{G})$ to $cp(\mathcal{L} \cup \{s \xrightarrow{c} t\}, \mathcal{G})$ is sound for lower bounds.*

To prove Theorem 9.26, we need the following auxiliary lemma, which shows that \rightarrow is closed under instantiation of variables with natural numbers.

Lemma 9.27 (Stability of \rightarrow and $\rightarrow_{\ell \rightarrow r, n}$). *Let ℓ, r, s , and t be well-typed terms where s only contains variables of type \mathbb{N} , and let $\mu : \mathcal{V}(s) \rightarrow \mathbb{N}$. Then we have:*

- (a) $s \xrightarrow{c} t$ implies $s\mu \xrightarrow{c\mu} t\mu$
- (b) $s \xrightarrow{c}_{\ell \rightarrow r, n} t$ implies that there is a substitution $\sigma : \mathcal{V}(\ell) \rightarrow \mathbb{N}$ with $n\sigma = n\mu$ and $m\sigma \geq m\mu$ for all $m \in \mathcal{V}(\ell)$ such that $s\mu \xrightarrow{c\mu}_{\ell \sigma \rightarrow r \sigma, n} t\mu$.

Proof. Since rewriting is closed under substitutions, we immediately have (a). For (b), let $s \xrightarrow{c}_{\ell \rightarrow r, n} t$. If we also have $s \xrightarrow{c} t$, then the claim follows from (a). Otherwise, we have $s \mapsto_{\ell \rightarrow r, n} t$ and $c = 0$. Hence, there is a term s' , an increasing substitution σ' with $n\sigma' = n$, and a context C such that $s \equiv_{\mathcal{G} \cup \mathcal{A}} C[\ell\sigma']$ and $C[r\sigma'] \equiv_{\mathcal{G} \cup \mathcal{A}} t$. Let $\sigma = \sigma'\mu$. Then $s\mu \mapsto_{\ell\sigma \rightarrow r\sigma, n} t\mu$, since $s\mu \equiv_{\mathcal{G} \cup \mathcal{A}} C[\ell\sigma]\mu = C\mu[\ell\sigma]$ and $C\mu[r\sigma] = C[r\sigma']\mu \equiv_{\mathcal{G} \cup \mathcal{A}} t\mu$. Moreover, $n\sigma = n\sigma'\mu = n\mu$ and as σ' is increasing, $m\sigma' \geq m$ implies $m\sigma = m\sigma'\mu \geq m\mu$. \square

Now we can prove Theorems 9.22 and 9.26 together.

Proof of Theorems 9.22 and 9.26. To prove the theorems, it suffices to prove

$$s\mu \xrightarrow{\mathcal{K}}^* t\mu \text{ with } \mathcal{K} \geq c\mu \text{ for any } \mu : \mathcal{V}(s) \rightarrow \mathbb{N}. \quad (9.9)$$

We use induction on $n\mu$. For the induction base, assume $n\mu = 0$. By the prerequisites of the theorem, we have $s\{n/0\} \xrightarrow{i\mathcal{L}}^* t\{n/0\}$ (cf. Theorem 9.22). By Lemma 9.27 (a), \rightarrow is stable and thus we get $s\mu = s\{n/0\}\mu \xrightarrow{(i\mathcal{L})\mu}^* t\{n/0\}\mu = t\mu$. Finally, we have $c\mu = c\{n/0\}\mu = (i\mathcal{L})\mu$.

In the induction step, we have $n\mu > 0$. Let $\mu' : \mathcal{V}(s) \rightarrow \mathbb{N}$ where μ' is like μ for all $\mathcal{V}(s) \setminus \{n\}$ and $n\mu' = n\mu - 1$. We have

$$s\{n/n+1\} = v_1 \xrightarrow{c_1\theta_1}_{s \rightarrow t, n} \dots \xrightarrow{c_o\theta_o}_{s \rightarrow t, n} v_{o+1} = t\{n/n+1\}$$

with $o > 0$ and $i\mathcal{J} = \sum_{i=1}^o c_i\theta_i$ by the prerequisites of the theorem. By Lemma 9.27 (b), we obtain

$$\begin{aligned} & s\{n/n+1\}\mu' \\ = & v_1\mu' \xrightarrow{c_1\theta_1\mu'}_{s\sigma_1 \rightarrow t\sigma_1, n} \dots \xrightarrow{c_o\theta_o\mu'}_{s\sigma_o \rightarrow t\sigma_o, n} v_{o+1}\mu' = t\{n/n+1\}\mu' \end{aligned} \quad (9.10)$$

CHAPTER 9. INDUCTION TECHNIQUE

for substitutions σ_j such that $s\sigma_j$ and $t\sigma_j$ are ground, $n\sigma_j = n\mu'$, and $m\sigma_j \geq m\mu'$ for all $m \in \mathcal{V}(s)$.

For each step $v_j\mu' \mapsto_{s\sigma_j \rightarrow t\sigma_j, n} v_{j+1}\mu'$, we get $s\sigma_j \xrightarrow{\mathcal{K}' \Delta^*} t\sigma_j$ and thus also $v_j\mu' \xrightarrow{\mathcal{K}' \Delta^*} v_{j+1}\mu'$ with $\mathcal{K}' \geq c\sigma_j$ by the induction hypothesis since $n\sigma_j = n\mu' = n\mu - 1$. As c is weakly monotonic and $m\sigma_j \geq m\mu'$ for each $m \in \mathcal{V}(c) \subseteq \mathcal{V}(s)$, this implies $\mathcal{K}' \geq c\mu'$. Since there are $i\mathcal{K}$ many of these steps and each of them has cost 0 in (9.10), we get $s\{n/n+1\}\mu' \xrightarrow{\mathcal{K} \Delta^+} t\{n/n+1\}\mu'$ with

$$\overline{\mathcal{K}} \geq i\mathcal{K} \cdot c\mu' + (i\mathcal{J})\mu' \stackrel{(9.8)}{=} c\{n/n+1\}\mu'.$$

This proves the desired claim, since $s\mu \equiv_{\mathcal{A}} s\{n/n+1\}\mu'$, $t\mu \equiv_{\mathcal{A}} t\{n/n+1\}\mu'$, and $c\mu \equiv_{\mathcal{A}} c\{n/n+1\}\mu'$. \square

The following example shows how Theorem 9.26 can be used to obtain a rewrite lemma for \mathbf{qs} from the valid conjecture $\mathbf{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{\mathcal{O}_n^?} \text{gen}_{\text{List}}(n)$ from Example 9.23.

Example 9.28 (Computing c for \mathbf{qs}). Reconsider the induction proof of the conjecture $\mathbf{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{\mathcal{O}_n^?} \text{gen}_{\text{List}}(n)$ in Example 9.23. The proof of the induction base is $\mathbf{qs}(\text{gen}_{\text{List}}(0)) \equiv_{\mathcal{G}_{\mathbf{qs}}} \mathbf{qs}(\text{nil}) \xrightarrow{1}_{\mathcal{L}'_{\mathbf{qs}}} \text{nil} \equiv_{\mathcal{G}_{\mathbf{qs}}} \text{gen}_{\text{List}}(0)$. Hence, $i\mathcal{C} = 1$. The proof of the induction step is as follows.

$$\begin{aligned} & \mathbf{qs}(\text{gen}_{\text{List}}(n+1)) \equiv_{\mathcal{G}_{\mathbf{qs}}} \\ & \mathbf{qs}(\text{cons}(\text{zero}, \text{gen}_{\text{List}}(n))) \xrightarrow{1}_{\mathcal{L}'_{\mathbf{qs}}} \\ & \mathbf{qs}(\text{low}(\text{zero}, \text{gen}_{\text{List}}(n))) ++ \text{cons}(\text{zero}, \mathbf{qs}(\text{high}(\text{zero}, \text{gen}_{\text{List}}(n)))) \xrightarrow{3 \cdot n + 1}_{\Delta} \\ & \mathbf{qs}(\text{nil}) ++ \text{cons}(\text{zero}, \mathbf{qs}(\text{high}(\text{zero}, \text{gen}_{\text{List}}(n)))) \xrightarrow{3 \cdot n + 1}_{\Delta} \\ & \mathbf{qs}(\text{nil}) ++ \text{cons}(\text{zero}, \mathbf{qs}(\text{gen}_{\text{List}}(n))) \xrightarrow{1}_{\mathcal{L}'_{\mathbf{qs}}} \\ & \text{nil} ++ \text{cons}(\text{zero}, \mathbf{qs}(\text{gen}_{\text{List}}(n))) \mapsto_{\alpha_{\text{ih}, n}} \\ & \text{nil} ++ \text{cons}(\text{zero}, \text{gen}_{\text{List}}(n)) \xrightarrow{1}_{\mathcal{L}'_{\mathbf{qs}}} \\ & \text{cons}(\text{zero}, \text{gen}_{\text{List}}(n)) \equiv_{\mathcal{G}_{\mathbf{qs}}} \\ & \text{gen}_{\text{List}}(n+1) \end{aligned}$$

Thus, we have $i\mathcal{J} = 6 \cdot n + 5$ and $i\mathcal{K} = 1$. Now Theorem 9.26 implies

$$c = i\mathcal{C} + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\} = 1 + \sum_{i=0}^{n-1} (6 \cdot i + 5) = 3 \cdot n^2 + 2 \cdot n + 1.$$

Hence, we get the rewrite lemma

$$\mathbf{qs}(\text{gen}_{\text{List}}(n)) \xrightarrow{3 \cdot n^2 + 2 \cdot n + 1} \text{gen}_{\text{List}}(n). \quad (9.11)$$

In general, the recurrence equations (9.8) do not describe the exact cost of the corresponding rewrite sequence. The reason is that when proving a conjecture

9.4. INFERRING BOUNDS FOR VALID CONJECTURES

$s \xrightarrow{\mathcal{O}_n} t$ by induction, one may instantiate non-induction variables in the induction hypothesis, but this instantiation is ignored in the recurrence equations (9.8). Hence, in general c in Theorem 9.26 is only a lower bound for the runtime. However, even if the non-induction variables in the induction hypothesis are instantiated in the proof of a conjecture, c may still be exact.

Example 9.29 (Exact Bounds). The proof of

$$\text{add}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{\mathcal{O}_n} \text{gen}_{\text{Nat}}(n_2 + n)$$

in Example 9.24 used one rewrite step for the induction base and one for the induction step (i.e., $i\ell = 1$ and $i\mathcal{J} = 1$). The induction hypothesis was applied once (i.e., $i\mathcal{K} = 1$), where n_2 was instantiated with $n_2 + 1$. Thus, Theorem 9.26 results in the exact cost function

$$c = i\ell + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\} = 1 + n.$$

This yields the lemma

$$\text{add}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{1+n} \text{gen}_{\text{Nat}}(n_2 + n),$$

which results in a linear lower bound for the runtime complexity of the whole TRS.

In this example, the bound $1 + n$ for the runtime of the rewrite lemma is exact, because $i\ell$ and $i\mathcal{J}$ do not depend on n_2 . But the following modification of the `add`-TRS illustrates why our approach might fail to compute exact bounds.

Example 9.30 (Non-Exact Bounds). In the following TRS, `addDouble`(x, y) corresponds to a subsequent application of `add` and `double`, i.e., it first computes the addition of x and y , and then it doubles the result.

$$\begin{aligned} \text{addDouble}(\text{zero}, y) &\rightarrow \text{double}(y) \\ \text{addDouble}(\text{succ}(x), y) &\rightarrow \text{addDouble}(x, \text{succ}(y)) \\ \text{double}(\text{zero}) &\rightarrow \text{zero} \\ \text{double}(\text{succ}(x)) &\rightarrow \text{succ}(\text{succ}(\text{double}(x))) \end{aligned}$$

For `double`, we infer the rewrite lemma

$$\text{double}(\text{gen}_{\text{Nat}}(n)) \xrightarrow{1+n} \text{gen}_{\text{Nat}}(2 \cdot n).$$

For `addDouble`, the technique of Section 9.2 speculates the conjecture

$$\text{addDouble}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{\mathcal{O}_n} \text{gen}_{\text{Nat}}(2 \cdot n_2 + 2 \cdot n),$$

CHAPTER 9. INDUCTION TECHNIQUE

which is proved by induction. In the induction base, we have

$$\text{addDouble}(\text{gen}_{\text{Nat}}(0), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow[1+n_2]{1} \text{double}(\text{gen}_{\text{Nat}}(n_2)) \\ \text{gen}_{\text{Nat}}(2 \cdot n_2)$$

which yields $i\ell = 2 + n_2$. In the induction step, we get $i\mathcal{J} = 1$ and $i\mathcal{K} = 1$. Now Theorem 9.26 yields $c = i\ell + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\} = 2 + n_2 + n$, resulting in the rewrite lemma

$$\text{addDouble}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)) \xrightarrow{2+n_2+n} \text{gen}_{\text{Nat}}(2 \cdot n_2 + 2 \cdot n).$$

However, $2 + n_2 + n$ is only a lower bound on the cost of this rewrite sequence: The non-induction variable n_2 in **addDouble**'s second argument increases in each application of **addDouble**'s recursive rule. Therefore finally, **double**($\text{gen}_{\text{Nat}}(n_2 + n)$) has to be evaluated. Therefore, rewriting **addDouble**($\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n_2)$) has cost $2 + n_2 + 2 \cdot n$. However, the increase of n_2 is ignored in the recurrence equations (9.8) and in Theorem 9.26.

Unfortunately, the costs of the rewrite lemmas generated by Theorem 9.22 contain the operator \sum and hence they do not provide an intuitive understanding of the complexity of the corresponding rewrite sequences. Moreover, they do not immediately give rise to asymptotic bounds. To overcome this problem, one could use recurrence solving to transform \sum -expressions into algebraic expressions, i.e., expressions which only contain the operations (binary) addition, subtraction, multiplication, division, and exponentiation. However, if one is mainly interested in *asymptotic* instead of explicit bounds, then one can find suitable algebraic expressions without solving recurrence equations. The reason is that, for asymptotic bounds, constant factors in the costs of rewrite lemmas can be neglected.

Let $s \xrightarrow{\mathcal{O}_n} t$ be a valid conjecture as in Theorem 9.22. If the induction hypothesis was used once in the proof (i.e., $i\mathcal{K} = 1$), then Lemma 9.25 implies $c = i\ell + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\}$. We now show that

if $i\mathcal{J}$ is a polynomial, then we have

$$c \geq k \cdot (i\ell + n \cdot i\mathcal{J}\{n/n - 1\}) \text{ for some constant } 0 < k \leq 1 \quad (9.12)$$

and hence approximating c with $i\ell + n \cdot i\mathcal{J}\{n/n - 1\}$ is asymptotically sound. Therefore, note that $i\ell$ and $i\mathcal{J}$ are non-negative when instantiated with natural numbers. The reason is that $i\ell$ and $i\mathcal{J}$ are sums of costs of rewrite lemmas, which are non-negative by Definition 9.5. For $n = 0$, we have

$$\begin{aligned} c\{n/0\} &= \left(i\ell + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\} \right) \{n/0\} \\ &= i\ell\{n/0\} \\ &\geq k \cdot i\ell\{n/0\} && \text{as } i\ell \text{ is non-negative} \\ &= k \cdot (i\ell + n \cdot i\mathcal{J}\{n/n - 1\})\{n/0\} \end{aligned}$$

9.4. INFERRING BOUNDS FOR VALID CONJECTURES

for all $0 < k \leq 1$. For $n = 1$, we have

$$\begin{aligned} c\{n/1\} &= \left(i\ell + \sum_{i=0}^{n-1} i\mathcal{J}\{n/i\} \right) \{n/1\} \\ &= i\ell\{n/1\} + i\mathcal{J}\{n/0\} \\ &\geq k \cdot (i\ell\{n/1\} + i\mathcal{J}\{n/0\}) \quad \text{as } i\ell \text{ and } i\mathcal{J} \text{ are non-negative} \\ &= k \cdot (i\ell + n \cdot i\mathcal{J}\{n/n-1\}) \{n/1\} \end{aligned}$$

for all $0 < k \leq 1$. For the case $n > 1$, let $d_{i\mathcal{J}}$ be the degree of $i\mathcal{J}$ w.r.t. n . Then we have

$$i\mathcal{J} = \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot n^m \quad (9.13)$$

where each t_m is a polynomial that does not contain n . Hence,

$$c = i\ell + \sum_{i=0}^{n-1} \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot i^m = i\ell + \sum_{m=0}^{d_{i\mathcal{J}}} \sum_{i=0}^{n-1} t_m \cdot i^m = i\ell + \sum_{m=0}^{d_{i\mathcal{J}}} \left(t_m \cdot \sum_{i=0}^{n-1} i^m \right) \quad (9.14)$$

By Faulhaber's formula [93],

$$s_m = \sum_{i=0}^{n-1} i^m \text{ is a polynomial over } n \text{ with degree } m+1 \quad (9.15)$$

for any $m \in \mathbb{N}$. For example, $\sum_{i=0}^{n-1} i^1 = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n$ is a polynomial with degree 2. Thus, (9.14) implies

$$c = i\ell + \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot s_m. \quad (9.16)$$

Note that

$$s_m\{n/0\} = s_m\{n/1\} = 0, \quad (9.17)$$

$$s_m \text{ is strictly monotonically increasing for } n \geq 1, \text{ and} \quad (9.18)$$

$$s_m \geq 1 \text{ for all } n > 1 \quad (9.19)$$

hold for all $m \in \mathbb{N}$.

To finish the poof of (9.12), we first prove that

$$\text{for each } s_m, n > 1 \text{ implies } s_m \geq k_m \cdot n^{m+1} \text{ for some } 0 < k_m \leq 1. \quad (9.20)$$

Let $s_m = \sum_{i=0}^{m+1} c_i \cdot n^i$. Then (9.17) implies $c_0 = 0$. Thus, (9.20) holds if and only if we have

$$\sum_{i=1}^m c_i \cdot n^i \geq (k_m - c_{m+1}) \cdot n^{m+1} \quad (9.21)$$

for all $n > 1$. As, according to (9.15), the degree of s_m is $m+1$, (9.18) implies

CHAPTER 9. INDUCTION TECHNIQUE

$c_{m+1} > 0$. Thus, for $n \geq 0$

$$(k_m - c_{m+1}) \cdot n^{m+1} \text{ is a strictly monotonically decreasing polynomial} \\ \text{with degree } m+1 \text{ for any } 0 < k_m < \min(1, c_{m+1}). \quad (9.22)$$

Let $0 < k_{\max} < \min(c_{m+1}, 1)$. If (9.21) holds for all $n > 1$ and $k_m = k_{\max}$, then the claim (9.20) follows immediately. Thus, assume that there is an $n > 1$ such that (9.21) does not hold for $k_m = k_{\max}$. Let $n_0 > 1$ be the maximal natural number such that (9.21) is violated for $k_m = k_{\max}$, i.e., (9.21) holds for all $n > n_0$. Note that n_0 exists due to (9.22), as the degree of $\sum_{i=1}^m c_i \cdot n^i$ is at most m . Then clearly

$$(9.21) \text{ holds for all } n > n_0 \text{ and all } 0 < k_m \leq k_{\max}. \quad (9.23)$$

Let $k_m = \min\left(k_{\max}, \frac{1}{n_0^{m+1}}\right)$. Then we get:

$$\begin{aligned} & \sum_{i=1}^m c_i \cdot n^i \\ &= s_m - c_{m+1} \cdot n^{m+1} && \text{as } s_m = \sum_{i=1}^{m+1} c_i \cdot n^i \\ &\geq 1 - c_{m+1} \cdot n^{m+1} && \text{for all } n > 1 \text{ by (9.19)} \\ &\geq \frac{n^{m+1}}{n_0^{m+1}} - c_{m+1} \cdot n^{m+1} && \text{for all } n \leq n_0 \\ &\geq \min\left(k_{\max} \cdot n^{m+1}, \frac{n^{m+1}}{n_0^{m+1}}\right) - c_{m+1} \cdot n^{m+1} \\ &= (k_m - c_{m+1}) \cdot n^{m+1}, \end{aligned}$$

i.e., (9.21) holds for all $1 < n \leq n_0$. Thus, as we have

$$k_m = \min\left(k_{\max}, \frac{1}{n_0^{m+1}}\right) \leq k_{\max},$$

(9.21) holds for all $n > 1$ due to (9.23), which finishes the proof of (9.20).

Hence, for all $n > 1$ we have

$$\begin{aligned} c &= i\ell + \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot s_m && \text{by (9.16)} \\ &\geq i\ell + \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot (k_i \cdot n^{m+1}) && \text{by (9.20)} \\ &\geq i\ell + k \cdot \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot n^{m+1} && \text{where } k = \min\{k_1, \dots, k_{d_{i\mathcal{J}}}\} \\ &= i\ell + k \cdot n \cdot \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot n^m \\ &\geq k \cdot (i\ell + n \cdot \sum_{m=0}^{d_{i\mathcal{J}}} t_m \cdot n^m) && \text{as } 0 < k \leq 1 \text{ and } i\ell \text{ is non-negative} \\ &= k \cdot (i\ell + n \cdot i\mathcal{J}) && \text{by (9.13)} \\ &\geq k \cdot (i\ell + n \cdot i\mathcal{J}\{n/n-1\}) && \text{due to monotonicity of } i\mathcal{J}. \end{aligned}$$

Note that $i\mathcal{J}$ is weakly monotonically increasing as it is a sum of costs of rewrite lemmas, which are weakly monotonically increasing by Definition 9.5. This finishes the proof of (9.12).

Now we consider the case where the induction hypothesis was used several times, i.e., $i\ell > 1$. In this case, the valid conjecture corresponds to a family of

9.4. INFERRING BOUNDS FOR VALID CONJECTURES

rewrite sequences with exponential costs. More precisely, Lemma 9.25 implies

$$c \geq i\hbar^n \cdot i\ell \quad (9.24)$$

$$\begin{aligned} \text{and, if } \mathcal{A} \models i\mathcal{s} \geq 1, \quad c &= i\hbar^n \cdot i\ell + \sum_{i=0}^{n-1} i\hbar^{n-1-i} \cdot i\mathcal{s}\{n/i\} \\ &\geq i\hbar^n \cdot i\ell + \sum_{i=0}^{n-1} i\hbar^{n-1-i} \\ &= i\hbar^n \cdot i\ell + \sum_{j=0}^{n-1} i\hbar^j \\ &= i\hbar^n \cdot i\ell + \frac{i\hbar^n - 1}{i\hbar - 1}. \end{aligned} \quad (9.25)$$

Theorem 9.31 (Asymptotic Runtime of Valid Conjectures). *Let $s \xrightarrow{\text{On}_?} t$ be a conjecture with an induction proof as in Theorem 9.22 where the induction hypothesis $s \rightarrow t$ was applied $i\hbar$ times. Let proc_i be the processor mapping $cp(\mathcal{L}, \mathcal{G})$ to $cp(\mathcal{L} \cup \{s \xrightarrow{c_i} t\}, \mathcal{G})$ where*

- (1) $c_1 = i\ell + n \cdot i\mathcal{s}\{n/n - 1\}$ if $i\mathcal{s}$ is a polynomial and $i\hbar = 1$,
- (2) $c_2 = i\hbar^n \cdot i\ell + \frac{i\hbar^n - 1}{i\hbar - 1}$ if $i\hbar > 1$ and $\mathcal{A} \models i\mathcal{s} \geq 1$, and
- (3) $c_3 = i\hbar^n \cdot i\ell$ if $i\hbar > 1$.

Then proc_1 is asymptotically sound for lower bounds and proc_2 and proc_3 are sound for lower bounds.

Proof. According to Theorem 9.26, the processor mapping $cp(\mathcal{L}, \mathcal{G})$ to $cp(\mathcal{L} \cup \{s \xrightarrow{c_i} t\}, \mathcal{G})$ is sound for lower bounds. Thus, proc_1 is asymptotically sound due to (9.12) and proc_2 and proc_3 are sound due to (9.25) and (9.24). \square

Example 9.32 (Exponential Runtime). To illustrate Theorem 9.31, let

$$\mathcal{L}_{\text{exp}} = \{f(\text{succ}(x), \text{succ}(x)) \rightarrow f(f(x, x), f(x, x)), f(\text{zero}, \text{zero}) \rightarrow \text{zero}\}.$$

Our approach speculates and proves the conjecture

$$f(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n)) \xrightarrow{\text{On}_?} \text{zero}.$$

The induction base is $f(\text{gen}_{\text{Nat}}(0), \text{gen}_{\text{Nat}}(0)) \equiv_{\mathcal{G}} f(\text{zero}, \text{zero}) \rightarrow_{\mathcal{L}_{\text{exp}}} \text{zero}$, i.e., $i\ell = 1$. The induction step is:

$$\begin{aligned} f(\text{gen}_{\text{Nat}}(n+1), \text{gen}_{\text{Nat}}(n+1)) &\equiv_{\mathcal{G}} \\ f(\text{succ}(\text{gen}_{\text{Nat}}(n)), \text{succ}(\text{gen}_{\text{Nat}}(n))) &\xrightarrow{1}_{\mathcal{L}_{\text{exp}}} \\ f(f(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n)), f(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n))) &\xrightarrow{2}_{\alpha_{\text{ih}}} \\ f(\text{zero}, \text{zero}) &\xrightarrow{1}_{\mathcal{L}_{\text{exp}}} \\ \text{zero} & \end{aligned}$$

CHAPTER 9. INDUCTION TECHNIQUE

Thus, $i\mathcal{R} = 2$ and $i\mathcal{J} = 2$. Hence, by Theorem 9.31 (2), adding the rewrite lemma $f(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(n)) \xrightarrow{2^{n+1}-1} \text{zero}$ to \mathcal{L}_{exp} is sound for lower bounds. Indeed, Theorem 9.26 implies

$$c = 2^n + \sum_{i=0}^{n-1} 2^{n-1-i} \cdot 2 = 2^{n+1} + 2^n - 2 \geq 2^{n+1} - 1.$$

9.5 Inferring Bounds for TRSs

We now use rewrite lemmas to infer lower bounds on $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$. While we already know the cost function c of each rewrite lemma $s \xrightarrow{c} t \in \mathcal{L}$, $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$ is defined w.r.t. the size of the start term of a rewrite sequence. To obtain a lower bound for $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$ from c , for any substitution σ one has to take the relation between $c\sigma$ and the size of the start term $s\sigma$ into account. Our approach in Section 9.2 only speculates lemmas where $s = f(\text{gen}_{\tau_1}(s_1), \dots, \text{gen}_{\tau_m}(s_m))$ for some $f \in \Sigma_d(\mathcal{L})$, polynomials s_1, \dots, s_m , and simply structured types τ_1, \dots, τ_m . For any τ_i , let $d_{\tau_i} : \tau'_1 \times \dots \times \tau'_b \rightarrow \tau$ be τ_i 's recursive constructor. Then for any $n \in \mathbb{N}$, Definition 9.4 implies

$$\|\text{gen}_{\tau_i}(n)\|_{\mathcal{G}} = \|\text{gen}_{\tau_i}(0)\|_{\mathcal{G}} + n \cdot \left(1 + \sum_{i=1}^b \|\text{gen}_{\tau'_i}(0)\|_{\mathcal{G}} - \|\text{gen}_{\tau_i}(0)\|_{\mathcal{G}}\right).$$

The reason is that the term $\text{gen}_{\tau_i}(n)$ contains n occurrences of d_{τ_i} and of each $\text{gen}_{\tau'_1}(0), \dots, \text{gen}_{\tau'_b}(0)$ except for $\text{gen}_{\tau_i}(0)$, and just one occurrence of $\text{gen}_{\tau_i}(0)$. For instance, we have:

$$\begin{aligned} \|\text{gen}_{\text{Nat}}(n)\|_{\mathcal{G}} &= \|\text{gen}_{\text{Nat}}(0)\|_{\mathcal{G}} + n \cdot (1 + \|\text{gen}_{\text{Nat}}(0)\|_{\mathcal{G}} - \|\text{gen}_{\text{Nat}}(0)\|_{\mathcal{G}}) \\ &= \|\text{zero}\|_t + n \\ &= 1 + n \\ \|\text{gen}_{\text{List}}(n)\|_{\mathcal{G}} &= \|\text{gen}_{\text{List}}(0)\|_{\mathcal{G}} + n \cdot (1 + \|\text{gen}_{\text{Nat}}(0)\|_{\mathcal{G}}) \\ &= \|\text{nil}\|_t + n \cdot (1 + \|\text{zero}\|_t) \\ &= 1 + n \cdot 2 \end{aligned}$$

Thus $\|s\|_{\mathcal{G}} = \|f(\text{gen}_{\tau_1}(s_1), \dots, \text{gen}_{\tau_m}(s_m))\|_{\mathcal{G}}$ with $\mathcal{V}(s) = \mathbf{n}$ is given by

$$sz(\mathbf{n}) = 1 + \|\text{gen}_{\tau_1}(s_1)\|_{\mathcal{G}} + \dots + \|\text{gen}_{\tau_m}(s_m)\|_{\mathcal{G}}.$$

For instance, $\text{qs}(\text{gen}_{\text{List}}(n)) \equiv_{\mathcal{G} \cup \mathcal{A}} \text{qs}(\text{cons}^n(\text{zero}, \text{nil}))$ has the size $sz(n) = 1 + \|\text{gen}_{\text{List}}(n)\|_{\mathcal{G}} = 2 \cdot n + 2$. Since $\|\text{gen}_{\tau}(0)\|_{\mathcal{G}}$ is a constant for each type τ , sz is a polynomial whose degree is the maximal degree of the polynomials s_1, \dots, s_k . Hence, the rewrite lemma (9.1) for qs states that there are terms of size $sz(n) = 2 \cdot n + 2$ with reductions whose costs are at least $c = 3 \cdot n^2 + 2 \cdot n + 1$. To determine a lower bound for $\text{rc}_{cp(\mathcal{L}_{\text{qs}}, \mathcal{G}_{\text{qs}})}$, we construct an inverse function sz^{-1} with $(sz \circ sz^{-1})(n) = n$. In our example where $sz(n) = 2 \cdot n + 2$, we have $sz^{-1}(n) = \frac{n-2}{2}$ if n is even. Thus, for all even n there are terms of size n with reductions of length $c\{n/sz^{-1}(n)\} = c\{n/\frac{n-2}{2}\} = \frac{3}{4} \cdot n^2 - 2 \cdot n + 2$. Since multivariate polynomials $sz(\mathbf{n})$ cannot be inverted, we invert the unary function $sz_{\mathbb{N}} : \mathbb{N} \rightarrow \mathbb{N}$ with $sz_{\mathbb{N}}(n) = sz(n, \dots, n)$ instead.

Of course, inverting $sz_{\mathbb{N}}$ fails if $sz_{\mathbb{N}}$ is not injective. However, the conjectures speculated in Section 9.2 only contain polynomials with natural coefficients. Then, $sz_{\mathbb{N}}$ is always strictly monotonically increasing. Hence, we only proceed

CHAPTER 9. INDUCTION TECHNIQUE

if there is an $sz_{\mathbb{N}}^{-1} : \text{img}(sz_{\mathbb{N}}) \rightarrow \mathbb{N}$ where $(sz_{\mathbb{N}} \circ sz_{\mathbb{N}}^{-1})(n) = n$ holds for all $n \in \text{img}(sz_{\mathbb{N}})$. To extend $sz_{\mathbb{N}}^{-1}$ to a function on $\{n \in \mathbb{N} \mid n \geq \min(\text{img}(sz_{\mathbb{N}}))\}$, we define

$$\lfloor sz_{\mathbb{N}}^{-1} \rfloor(m) = sz_{\mathbb{N}}^{-1}(\max\{m' \in \text{img}(sz_{\mathbb{N}}) \mid m' \leq m\}).$$

Then Theorem 9.33 states how we can derive lower bounds for $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$.

Theorem 9.33 (Explicit Bounds for $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$). *Let $s \xrightarrow{c} t \in \mathcal{L}$ where $s \in \mathcal{T}_{\text{basic}}(\mathcal{L}, \mathcal{G})$ and all variables from $\mathcal{V}(s) = \mathbf{n}$ have type \mathbb{N} , let $sz : \mathbb{N}^{\text{len}(\mathbf{n})} \rightarrow \mathbb{N}$ be defined by $sz(\mathbf{c}) = \|s\{\mathbf{n}/\mathbf{c}\}\|_{\mathcal{G}}$, and let $sz_{\mathbb{N}}$ be injective, i.e., $sz_{\mathbb{N}}^{-1}$ exists. Then for all $n \in \mathbb{N}$ with $n \geq \min(\text{img}(sz_{\mathbb{N}}))$, we have*

$$\text{rc}_{cp(\mathcal{L}, \mathcal{G})}(n) \geq c\{x / \lfloor sz_{\mathbb{N}}^{-1} \rfloor(n) \mid x \in \mathcal{V}(s)\}.$$

Proof. Let $\text{len}(\mathbf{n}) = m$. If $n \geq \min(\text{img}(sz_{\mathbb{N}}))$, then there is a maximal $n' \leq n$ such that $n' \in \text{img}(sz_{\mathbb{N}})$. Thus, $\lfloor sz_{\mathbb{N}}^{-1} \rfloor(n) = sz_{\mathbb{N}}^{-1}(n')$. Moreover, we have

$$\begin{aligned} \|s\{\mathbf{n}|_1 / sz_{\mathbb{N}}^{-1}(n'), \dots, \mathbf{n}|_m / sz_{\mathbb{N}}^{-1}(n')\}\|_{\mathcal{G}} &= sz(sz_{\mathbb{N}}^{-1}(n'), \dots, sz_{\mathbb{N}}^{-1}(n')) \\ &= sz_{\mathbb{N}}(sz_{\mathbb{N}}^{-1}(n')) \\ &= n'. \end{aligned}$$

Thus, we have $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}(n') \geq c\{x / \lfloor sz_{\mathbb{N}}^{-1} \rfloor(n) \mid x \in \mathcal{V}(s)\}$. Since $n \geq n'$, we get $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}(n) \geq c\{x / \lfloor sz_{\mathbb{N}}^{-1} \rfloor(n) \mid x \in \mathcal{V}(s)\}$. \square

In the rewrite lemma (9.1) for \mathbf{qs} where $sz_{\mathbb{N}}(n) = 2 \cdot n + 2$, we have $\lfloor sz_{\mathbb{N}}^{-1} \rfloor(n) = \lfloor \frac{n-2}{2} \rfloor \geq \frac{n-3}{2}$ and

$$\text{rc}_{cp(\mathcal{L}_{\mathbf{qs}}, \mathcal{G}_{\mathbf{qs}})}(n) \geq c\{n / \lfloor sz_{\mathbb{N}}^{-1} \rfloor(n)\} \geq c\left\{n / \frac{n-3}{2}\right\} = \frac{3}{4} \cdot n^2 - \frac{7}{2} \cdot n + \frac{19}{4}$$

for all $n \geq 2$.

However, even if $sz_{\mathbb{N}}^{-1}$ exists, finding resp. approximating $sz_{\mathbb{N}}^{-1}$ automatically can be non-trivial in general. Therefore, we now show how to obtain an asymptotic lower bound for $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$ directly from a rewrite lemma

$$\mathbf{f}(\text{gen}_{\tau_1}(s_1), \dots, \text{gen}_{\tau_m}(s_m)) \xrightarrow{c} t$$

as in Theorem 9.31 without constructing $sz_{\mathbb{N}}^{-1}$. As mentioned, if e is the maximal degree of the polynomials s_1, \dots, s_k , then sz is also a polynomial of degree e and thus, $sz_{\mathbb{N}}(n) \in \mathcal{O}(n^e)$. Moreover, Theorem 9.31 (1) – (3) immediately give rise to a polynomial or exponential asymptotic complexity class for $c_{\mathbb{N}} = c\{m/n \mid m \in \mathcal{V}(c)\}$. Thus, as in Chapter 4, we can use Lemma 4.41 to infer an asymptotic lower bound on $\text{rc}_{cp(\mathcal{L}, \mathcal{G})}$.

9.5. INFERRING BOUNDS FOR TRSS

Lemma (Bounds for Function Composition (Repetition of Lemma 4.41)).

Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ where $g(m) \in \mathcal{O}(m^d)$ for some $d \in \mathbb{N}$ with $d > 0$. Moreover, let $f(m)$ be weakly and let $g(m)$ be strictly monotonically increasing for large enough m .

- If $f(g(m)) \in \Omega(m^k)$ with $k \in \mathbb{N}$, then $f(m) \in \Omega(m^{\frac{k}{d}})$.
- If $f(g(m)) \in \Omega(k^m)$ with $k > 1$, then $f(m) \in \Omega(b^{\sqrt[d]{m}})$ for some $b > 1$.

To apply the lemma, we set $f(m) = c_{\mathbb{N}}\{m/sz_{\mathbb{N}}^{-1}(m)\}$ and $g(m) = sz_{\mathbb{N}}(m)$. Then we get $f(g(m)) = c_{\mathbb{N}}\{m/sz_{\mathbb{N}}^{-1}(sz_{\mathbb{N}}(m))\} = c_{\mathbb{N}}$. Note that for all rewrite lemmas that were inferred by the techniques presented in Sections 9.2 and 9.3, the function $g(m) = sz_{\mathbb{N}}(m)$ and thus also $sz_{\mathbb{N}}^{-1}(m)$ is strictly monotonically increasing by construction. Thus, since we restricted ourselves to weakly monotonically increasing cost functions, $f(m) = c_{\mathbb{N}}\{m/sz_{\mathbb{N}}^{-1}(m)\}$ is weakly monotonically increasing, i.e., Lemma 4.41 is indeed applicable.

To illustrate the application of Lemma 4.41, assume that $sz_{\mathbb{N}}$ is a polynomial of degree d . If $c_{\mathbb{N}}$ is a polynomial of degree k , then Lemma 4.41 allows us to deduce $f(n) = c_{\mathbb{N}}\{n/sz_{\mathbb{N}}^{-1}(n)\} \in \Omega(n^{\frac{k}{d}})$ and thus $rc_{cp(\mathcal{L}, \mathcal{G})}(n) \in \Omega(n^{\frac{k}{d}})$. Similarly, if $c_{\mathbb{N}}$ is an exponential function, then Lemma 4.41 yields $f(n) = c_{\mathbb{N}}\{n/sz_{\mathbb{N}}^{-1}(n)\} \in \Omega(b^{\sqrt[d]{n}})$ and thus $rc_{cp(\mathcal{L}, \mathcal{G})}(n) \in \Omega(b^{\sqrt[d]{n}})$ for some $b > 1$.

So for the rewrite lemma $qs(\text{gen}_{\text{List}}(n)) \xrightarrow{c} \text{gen}_{\text{List}}(n)$ where $c_{\mathbb{N}} = c$ and $sz_{\mathbb{N}} = sz$, we only need the asymptotic bounds $sz_{\mathbb{N}}(n) \in \mathcal{O}(n)$ and $c_{\mathbb{N}} \in \Omega(n^2)$ to infer that Quicksort has at least quadratic complexity, i.e., $rc_{cp(\mathcal{L}_{qs}, \mathcal{G}_{qs})}(n) \in \Omega(n^{\frac{2}{1}}) = \Omega(n^2)$.

Thus, we obtain the following corollary.

Corollary 9.34 (From Rewrite Lemmas to rc). *Let $s \xrightarrow{c} t \in \mathcal{L}$ be a rewrite lemma where all variables in $\mathcal{V}(s)$ have type \mathbb{N} and $sz_{\mathbb{N}}$ is a polynomial of degree d .*

If $c_{\mathbb{N}}(n) \in \Theta(n^k)$, then $rc_{cp(\mathcal{L}, \mathcal{G})}(n) \in \Omega(n^{\frac{k}{d}})$.

If $c_{\mathbb{N}}(n) \in \Theta(k^n)$ with $k > 1$, then $rc_{cp(\mathcal{L}, \mathcal{G})}(n) \in \Omega(b^{\sqrt[d]{n}})$ for some $b > 1$.

Proof. Immediate consequence of Lemma 4.41. □

9.6 Indefinite Rewrite Lemmas

Our technique often fails if the analyzed set of rewrite lemmas is not completely defined, i.e., if there are well-typed ground normal forms containing defined symbols. As an example, the runtime complexity of

$$\mathcal{L}_{\text{in}} = \{f(\text{succ}(x)) \xrightarrow{1} \text{succ}(f(x))\}$$

is linear due to the rewrite sequences $f(\text{succ}^n(\text{zero})) \xrightarrow{n \cdot * } \text{succ}^n(f(\text{zero}))$. However, the term $\text{succ}^n(f(\text{zero}))$ on the right-hand side contains f and thus it cannot be represented in a rewrite lemma. Therefore, we now also allow *indefinite* conjectures and rewrite lemmas with unknown right-hand sides. Then for our example, we could speculate the indefinite conjecture $f(\text{gen}_{\text{Nat}}(n)) \xrightarrow{\odot n ? } \star$, which gives rise to the indefinite rewrite lemma $f(\text{gen}_{\text{Nat}}(n)) \xrightarrow{n ? } \star$, where \star is a fresh constant that represents an arbitrary unknown term.

Recall that when speculating conjectures in Section 9.2, we built a narrowing tree for a term $s = f(\dots)$ and obtained a sample conjecture $s\sigma \xrightarrow{\odot d ! } t$ whenever we reached a normal form t . When speculating indefinite conjectures, we do not narrow in order to reach normal forms, but we create a sample conjecture $s\sigma \xrightarrow{\odot d ! } \star$ after each application of a recursive f -rule. Here, σ is again the substitution and d is the recursion depth of the corresponding path in the narrowing tree. Note that we do not use previous indefinite rewrite lemmas during narrowing, since they do not yield any information on the terms resulting from rewriting.

Example 9.35 (Speculating Indefinite Conjectures). For \mathcal{L}_{in} , we narrow the term $s = f(\text{gen}_{\text{Nat}}(x))$. We get $f(\text{gen}_{\text{Nat}}(x)) \rightsquigarrow \text{succ}(f(\text{gen}_{\text{Nat}}(x')))$ with the substitution $\sigma_1 = \{x/x' + 1\}$. Since we applied a recursive f -rule once, we construct the sample conjecture $s\sigma_1 \xrightarrow{\odot 1 ! } \star$. We continue narrowing and get $\text{succ}(f(\text{gen}_{\text{Nat}}(x')))) \rightsquigarrow \text{succ}(\text{succ}(f(\text{gen}_{\text{Nat}}(x''))))$ with the substitution $\sigma_2 = \{x'/x'' + 1\}$ and recursion depth 2. Since $\sigma_1 \diamond \sigma_2$ corresponds to $\{x/x'' + 2\}$, this yields the sample conjecture $s\{x/x'' + 2\} \xrightarrow{\odot 2 ! } \star$. Another narrowing step results in the sample conjecture $s\{x/x''' + 3\} \xrightarrow{\odot 3 ! } \star$.

These sample conjectures are identical up to the occurring numbers and variable names. Thus, they are suitable for generalization. As in Section 9.2, we replace the numbers in the sample conjectures by polynomials pol in one variable n that represents the recursion depth. This leads to the conjecture $f(\text{gen}_{\text{Nat}}(x + \text{pol})) \xrightarrow{\odot n ? } \star$ and the constraints $\text{pol}(1) = 1, \text{pol}(2) = 2, \text{pol}(3) = 3$. These constraints have the solution $\text{pol}(n) = n$. Thus, we speculate the indefinite conjecture $f(\text{gen}_{\text{Nat}}(x + n)) \xrightarrow{\odot n ? } \star$.

In principle, proving indefinite conjectures $s \rightarrow^* \star$ is not necessary, since adding $s \xrightarrow{0} \star$ to \mathcal{L} is always sound. However, to derive useful lower complexity bounds, we need rewrite lemmas $s \xrightarrow{c} \star$ with non-trivial cost functions c . Theorem 9.36 shows that the approaches for proving lemmas from Section 9.3 and for deriving

CHAPTER 9. INDUCTION TECHNIQUE

for substitutions $\sigma_i : \mathcal{V}(s) \rightarrow \mathbb{N}$ such that $s\sigma_i$ is ground, $n\sigma_i = n\mu'$, and $m\sigma_i \geq m\mu'$ for all $m \in \mathcal{V}(s)$.

For each step $v_i\mu' \mapsto_{s\sigma_i \rightarrow \star, n} v_{i+1}\mu'$, there is a term t_i such that

$$s\sigma_i \xrightarrow{\ell_i \searrow \star} t_i \text{ with } \ell_i \geq c\sigma_i \quad (9.27)$$

by the induction hypothesis since $n\sigma_i = n\mu' = n\mu - 1$. As c is weakly monotonic and $m\sigma_i \geq m\mu'$ for each $m \in \mathcal{V}(c) \subseteq \mathcal{V}(s)$, this implies $\ell_i \geq c\mu'$. From (9.27) we get $v_i\mu' \xrightarrow{\ell_i \searrow \star} v'_{i+1}$ with $v_{i+1}\mu' \succ_\star v'_{i+1}$. Since \rightarrow is closed under \succ_\star , we get

$$\begin{array}{ccccccc} v_1\mu' & \xrightarrow{c_1\theta_1\mu'}_{s\sigma_1 \rightarrow \star, n} & v_2\mu' & \xrightarrow{c_2\theta_2\mu'}_{s\sigma_2 \rightarrow \star, n} & \cdots & \xrightarrow{c_o\theta_o\mu'}_{s\sigma_o \rightarrow \star, n} & v_{o+1}\mu' \\ \Uparrow & & \Uparrow & & \cdots & & \Uparrow \\ \star & & \star & & \cdots & & \star \\ v_1\mu' & \xrightarrow{\ell_1 \searrow \star} & v'_2 & \xrightarrow{\ell_2 \searrow \star} & \cdots & \xrightarrow{\ell_o \searrow \star} & v'_{o+1} \end{array}$$

where $\ell_i = c_i\theta_i\mu'$ if $v_i\mu' \rightarrow v_{i+1}\mu'$ and $\ell_i \geq c\mu'$ if $v_i\mu' \mapsto_{s\sigma_i \rightarrow \star, n} v_{i+1}\mu'$, i.e.,

$$s\{n/n+1\}\mu' = v_1\mu' \xrightarrow{\ell_1 + \dots + \ell_o \searrow \star} v'_{o+1}.$$

Since there are $i\ell$ many \mapsto -steps (which have cost 0 in (9.26)), we get

$$\ell_1 + \dots + \ell_o \geq i\ell \cdot c\mu' + \sum_{i=1}^o c_i\theta_i\mu' = i\ell \cdot c\mu' + (i\mathcal{J})\mu' \stackrel{(9.8)}{=} c\{n/n+1\}\mu'.$$

This proves the claim, since $s\mu \equiv_{\mathcal{A}} s\{n/n+1\}\mu'$ and $c\mu \equiv_{\mathcal{A}} c\{n/n+1\}\mu'$. \square

Thus, the costs of indefinite rewrite lemmas can be computed analogously to the costs of definite rewrite lemmas. To illustrate Theorem 9.36, we continue Example 9.35.

Example 9.37 (Complexity of Indefinite Rewrite Lemmas). We now infer the runtime function c of the rewrite lemma $\alpha = \mathbf{f}(\text{gen}_{\text{Nat}}(x+n)) \xrightarrow{c} \star$. We have $i\ell = 0$, since $\mathbf{f}(\text{gen}_{\text{Nat}}(x+0))$ is already in normal form. In the induction step, we obtain

$$\mathbf{f}(\text{gen}_{\text{Nat}}(x+n+1)) \xrightarrow{1} \text{succ}(\mathbf{f}(\text{gen}_{\text{Nat}}(x+n))) \mapsto_{\alpha, n} \text{succ}(\star).$$

Thus, the induction hypothesis is applied $i\ell = 1$ time and we have $i\mathcal{J} = 1$. According to Theorem 9.26, we have

$$c = i\ell^n \cdot i\ell + \sum_{i=0}^{n-1} i\ell^{n-1-i} \cdot i\mathcal{J}\{n/i\} = 1^n \cdot 0 + \sum_{i=0}^{n-1} 1^{n-1-i} \cdot 1 = n.$$

Alternatively, Theorem 9.31 yields $c = i\ell + n \cdot i\mathcal{J}\{n/n-1\} = n$. Thus, every lower bound for $\mathcal{L}' = \mathcal{L}_{\text{in}} \cup \{\mathbf{f}(\text{gen}_{\text{Nat}}(x+n)) \xrightarrow{n} \star\}$ is also valid for \mathcal{L}_{in} . To obtain a bound on the runtime complexity of \mathcal{L}' , we compute $sz = \|\mathbf{f}(\text{gen}_{\text{Nat}}(x+n))\|_{\mathcal{G}} = x+n+2$. Since $sz_{\mathbb{N}} = 2 \cdot n + 2$ is linear, Corollary 9.34 implies $\text{rc}_{cp(\mathcal{L}', \mathcal{G})}(n) \in \Omega(n)$. Thus we also have $\text{rc}_{cp(\mathcal{L}_{\text{in}}, \mathcal{G})}(n) \in \Omega(n)$.

9.7 Argument Filtering

A drawback of our approach is that generator symbols only represent homogeneous data objects (e.g., lists or trees where all elements have the same value **zero**). To prove lower complexity bounds also in cases where one needs other forms of rewrite lemmas, we use *argument filtering* [12] to remove certain argument positions of function symbols.

Example 9.38 (Argument Filtering). Consider the following TRS $\mathcal{R}_{\text{intlist}}$:

$$\text{intlist}(\text{zero}) \xrightarrow{1} \text{nil} \qquad \text{intlist}(\text{succ}(x)) \xrightarrow{1} \text{cons}(x, \text{intlist}(x))$$

For all $n \in \mathbb{N}$ we have:

$$\text{intlist}(\text{succ}^n(\text{zero})) \xrightarrow{n+1} \text{cons}(\text{succ}^{n-1}(\text{zero}), \dots \text{cons}(\text{succ}(\text{zero}), \text{cons}(\text{zero}, \text{nil})))$$

However, the inhomogeneous list on the right cannot be expressed using generator symbols. Filtering the first argument of **cons** yields $(\mathcal{R}_{\text{intlist}})_{\setminus(\text{cons}, 1)}$:

$$\text{intlist}(\text{zero}) \xrightarrow{1} \text{nil} \qquad \text{intlist}(\text{succ}(x)) \xrightarrow{1} \text{cons}(\text{intlist}(x))$$

For this TRS, our approach can speculate and prove the rewrite lemma

$$\begin{array}{ccc} \text{intlist}(\text{gen}_{\text{Nat}}(n)) & \xrightarrow{n+1} & \text{gen}_{\text{List}}(n), \text{ i.e.,} \\ \text{intlist}(\text{succ}^n(\text{zero})) & \xrightarrow{n+1} & \text{cons}^n(\text{nil}). \end{array}$$

From this rewrite lemma, one can infer $\text{rc}_{\text{cp}((\mathcal{R}_{\text{intlist}})_{\setminus(\text{cons}, 1)}, \mathcal{G})}(n) \in \Omega(n)$.

Definition 9.39 introduces the concept of argument filtering for terms and TRSs formally.

Definition 9.39 (Argument Filtering). Let Σ be a signature with $f \in \Sigma$, $\text{ar}_{\Sigma}(f) = n$, and let $i \in \{1, \dots, n\}$. Let $\Sigma_{\setminus(f, i)}$ be like Σ , but with $\text{ar}_{\Sigma_{\setminus(f, i)}}(f) = n - 1$. For any term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, we define the term $t_{\setminus(f, i)} \in \mathcal{T}(\Sigma_{\setminus(f, i)}, \mathcal{V})$ resulting from *filtering* the i^{th} argument of f . If $t \in \mathcal{V}$, then we have $t_{\setminus(f, i)} = t$. Otherwise, we have $t = g(t_1, \dots, t_b)$ and:

$$t_{\setminus(f, i)} = \begin{cases} g((t_1)_{\setminus(f, i)}, \dots, (t_{i-1})_{\setminus(f, i)}, (t_{i+1})_{\setminus(f, i)}, \dots, (t_b)_{\setminus(f, i)}) & \text{if } f = g \\ g((t_1)_{\setminus(f, i)}, \dots, (t_b)_{\setminus(f, i)}) & \text{if } f \neq g \end{cases}$$

Let \mathcal{R} be a TRS over Σ . Then we define

$$\mathcal{R}_{\setminus(f, i)} = \{\ell_{\setminus(f, i)} \xrightarrow{\mathcal{R}} r_{\setminus(f, i)} \mid \ell \xrightarrow{\mathcal{R}} r \in \mathcal{R}\}.$$

However, a lower bound for the runtime of $\mathcal{R}_{\setminus(f, i)}$ does not imply a lower bound for \mathcal{R} if the argument that is filtered away influences the control flow of the evaluation. Thus, several conditions have to be imposed to ensure that argument filtering is sound for lower bounds:

CHAPTER 9. INDUCTION TECHNIQUE

(1) **Argument filtering must not remove function symbols on left-hand sides of rules.**

An argument may not be filtered away if it is used for non-trivial pattern matching (i.e., if there is a left-hand side of a rule where the i^{th} argument of f is not a variable). As an example, consider

$$\mathcal{R} = \{f(\text{cons}(\text{true}, xs)) \rightarrow f(\text{cons}(\text{false}, xs))\}$$

where $\text{rc}_{cp(\mathcal{R})}(n) \leq 1$ for all n . But if one filters away the first argument of cons , then one obtains the non-terminating rule $f(\text{cons}(xs)) \rightarrow f(\text{cons}(xs))$, i.e., $\text{rc}_{cp(\mathcal{R}_{\setminus(\text{cons}, 1)}}(n) = \omega$ for $n \geq 3$.

(2) **The TRS must be left-linear.** To illustrate this, consider

$$\mathcal{R} = \{f(xs, xs) \rightarrow f(\text{cons}(\text{true}, xs), \text{cons}(\text{false}, xs))\},$$

where again $\text{rc}_{cp(\mathcal{R})}(n) \leq 1$. But filtering away the first argument of cons yields the non-terminating rule $f(xs, xs) \rightarrow f(\text{cons}(xs), \text{cons}(xs))$, i.e., $\text{rc}_{cp(\mathcal{R}_{\setminus(\text{cons}, 1)}}(n) = \omega$ for $n \geq 3$.

(3) **Argument filtering must not result in free variables on right-hand sides of rules.**

The reason is that, otherwise, the resulting system is not a valid TRS since, by definition, $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ must hold for each TRS rule $\ell \rightarrow r$.

Theorem 9.40 states that (1) – (3) are indeed sufficient for the soundness of argument filtering. To infer a lower bound for $\text{rc}_{cp(\mathcal{R})}$ from a bound for $\text{rc}_{cp(\mathcal{R}_{\setminus(f, i)})}$, we have to take into account that filtering changes the size of terms. As an example, consider $\mathcal{R} = \{f(x) \xrightarrow{1} a\}$. Here, we have $\text{rc}_{cp(\mathcal{R}_{\setminus(f, 1)}}(1) = 1$ due to the rewrite sequence $f \xrightarrow{1_{\mathcal{R}_{\setminus(f, 1)}}} a$. The corresponding rewrite sequence in the original TRS \mathcal{R} is $f(x) \xrightarrow{1_{\mathcal{R}}} a$. Thus, $\text{rc}_{cp(\mathcal{R})}(2) = 1$, but all terms of size 1 are normal forms of \mathcal{R} , i.e., $\text{rc}_{cp(\mathcal{R})}(1) = 0$. So $\text{rc}_{cp(\mathcal{R}_{\setminus(f, i)})}(n) \leq \text{rc}_{cp(\mathcal{R})}(n)$ does not hold in general. Nevertheless, for any rewrite sequence of $\mathcal{R}_{\setminus(f, i)}$ starting with a term t , there is a corresponding rewrite sequence of \mathcal{R} starting with a term⁸ s where $|s| \leq 2 \cdot |t|$. Thus, if we have derived a lower bound p for $\text{rc}_{cp(\mathcal{R}_{\setminus(f, i)})}(n)$, we can use $p\{n/\frac{n}{2}\}$ as a lower bound for $\text{rc}_{cp(\mathcal{R})}(n)$. In Example 9.38, we have $\|\text{intlist}(\text{succ}^n(\text{zero}))\|_t = n + 2$ and thus we obtain $\text{rc}_{cp((\mathcal{R}_{\text{intlist}})_{\setminus(\text{cons}, 1)}}(n) \geq n - 1$ by Theorem 9.33. Hence, we get $(n - 1)\{n/\frac{n}{2}\} = \frac{n}{2} - 1 \leq \text{rc}_{cp(\mathcal{R}_{\text{intlist}})}(n)$ for all $n \geq 4$ resp. $\text{rc}_{cp(\mathcal{R}_{\text{intlist}})}(n) \in \Omega(n)$.

⁸The term s can be obtained from t by adding a variable as the i^{th} argument for any f occurring in t .

9.7. ARGUMENT FILTERING

Theorem 9.40 (Soundness of Argument Filtering). *Let $f \in \Sigma$, $\text{ar}_\Sigma(f) = n$, and $i \in \{1, \dots, n\}$. Moreover, let \mathcal{R} be a left-linear TRS over Σ where the following conditions hold for all rules $\ell \rightarrow r \in \mathcal{R}$:*

(1) *If $f(t_1, \dots, t_k)$ is a subterm of ℓ , then $t_i \in \mathcal{V}$.*

(2) $\mathcal{V}(r_{\setminus(f,i)}) \subseteq \mathcal{V}(\ell_{\setminus(f,i)})$.

Then the processor mapping $cp(\mathcal{R})$ to $cp(\mathcal{R}_{\setminus(f,i)})$ is asymptotically sound for lower bounds.

Proof. For each substitution σ , we define $\sigma_{\setminus(f,i)} = \{x/t_{\setminus(f,i)} \mid x\sigma = t\}$. Note that we clearly have

$$(s\sigma)_{\setminus(f,i)} = s_{\setminus(f,i)}\sigma_{\setminus(f,i)} \quad \text{and} \quad (9.28)$$

$$C[s]_{\setminus(f,i)} = C_{\setminus(f,i)}[s_{\setminus(f,i)}] \quad \text{if } \square \text{ occurs in } C_{\setminus(f,i)}. \quad (9.29)$$

We first prove that the following holds for every linear term s where the i^{th} argument of every occurrence of f is a variable:

$$\begin{aligned} \text{If } s_{\setminus(f,i)}\theta &= t_{\setminus(f,i)}, \\ \text{then } s\sigma &= t \text{ for some } \sigma \text{ with } \sigma_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} = \theta|_{\mathcal{V}(s_{\setminus(f,i)})}. \end{aligned} \quad (9.30)$$

We use induction on s . If s is a variable, then we have $s = s_{\setminus(f,i)}$, $\sigma = \{s/t\}$, and $\sigma_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} = \{s/t_{\setminus(f,i)}\} = \theta|_{\mathcal{V}(s_{\setminus(f,i)})}$. Otherwise, we have $\text{root}(s_{\setminus(f,i)}) = \text{root}(t_{\setminus(f,i)})$ since $s_{\setminus(f,i)}$ matches $t_{\setminus(f,i)}$. Moreover, we clearly have $\text{root}(s_{\setminus(f,i)}) = \text{root}(s)$ and $\text{root}(t_{\setminus(f,i)}) = \text{root}(t)$. Thus, we have $\text{root}(s) = \text{root}(t)$, i.e., $s = g(s_1, \dots, s_m)$ and $t = g(t_1, \dots, t_m)$. By the induction hypothesis, we know:

$$\begin{aligned} \text{If } (s_j)_{\setminus(f,i)}\theta_j &= (t_j)_{\setminus(f,i)}, \\ \text{then } s_j\sigma_j &= t_j \text{ for some } \sigma_j \text{ with } (\sigma_j)_{\setminus(f,i)}|_{\mathcal{V}((s_j)_{\setminus(f,i)})} = \theta_j|_{\mathcal{V}((s_j)_{\setminus(f,i)})}. \end{aligned}$$

If $g \neq f$, then we have $(s_j)_{\setminus(f,i)}\theta = (t_j)_{\setminus(f,i)}$ for each $j \in \{1, \dots, m\}$. Thus, there are substitutions σ_j such that $s_j\sigma_j = t_j$ and $(\sigma_j)_{\setminus(f,i)}|_{\mathcal{V}((s_j)_{\setminus(f,i)})} = \theta|_{\mathcal{V}((s_j)_{\setminus(f,i)})}$ for all $j \in \{1, \dots, m\}$. W.l.o.g., assume $\text{dom}(\sigma_j) \subseteq \mathcal{V}(s_j)$ for each $j \in \{1, \dots, m\}$. Then we get $s\sigma = t$ where $\sigma = \sigma_1 \diamond \dots \diamond \sigma_m$ since s is linear. Moreover, we have

$$\begin{aligned} & \sigma_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} \\ &= (\sigma_1)_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} \diamond \dots \diamond (\sigma_m)_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} \quad \text{by definition of } \sigma \\ &= (\sigma_1)_{\setminus(f,i)}|_{\mathcal{V}((s_1)_{\setminus(f,i)})} \diamond \dots \diamond (\sigma_m)_{\setminus(f,i)}|_{\mathcal{V}((s_m)_{\setminus(f,i)})} \quad \text{as } \text{dom}(\sigma_j) \subseteq \mathcal{V}(s_j) \\ &= \theta|_{\mathcal{V}((s_1)_{\setminus(f,i)})} \diamond \dots \diamond \theta|_{\mathcal{V}((s_m)_{\setminus(f,i)})} \\ &= \theta|_{\mathcal{V}(s_{\setminus(f,i)})} \quad \text{as } s \text{ is linear.} \end{aligned}$$

If $g = f$, then we have $(s_j)_{\setminus(f,i)}\theta = (t_j)_{\setminus(f,i)}$ for each $j \in \{1, \dots, m\} \setminus \{i\}$.

CHAPTER 9. INDUCTION TECHNIQUE

Thus, there are substitutions σ_j such that $s_j\sigma_j = t_j$ and $(\sigma_j)_{\setminus(f,i)}|_{\mathcal{V}((s_j)_{\setminus(f,i)})} = \theta|_{\mathcal{V}((s_j)_{\setminus(f,i)})}$ for all $j \in \{1, \dots, m\} \setminus \{i\}$. W.l.o.g., assume $\text{dom}(\sigma_j) \subseteq \mathcal{V}(s_j)$ for each $j \in \{1, \dots, m\} \setminus \{i\}$. Since s_i is a variable and s is linear, we get $s\sigma = t$ with $\sigma = \sigma_1 \diamond \dots \diamond \sigma_m$ where $\sigma_i = \{s_i/t_i\}$. Let $\sigma'_j = (\sigma_j)_{\setminus(f,i)}$. Then we have

$$\begin{aligned}
 & \sigma_{\setminus(f,i)}|_{\mathcal{V}(s_{\setminus(f,i)})} \\
 &= \sigma'_1|_{\mathcal{V}(s_{\setminus(f,i)})} \diamond \dots \diamond \sigma'_m|_{\mathcal{V}(s_{\setminus(f,i)})} && \text{by definition of } \sigma \\
 &= \sigma'_1|_{\mathcal{V}(s_{\setminus(f,i)})} \diamond \dots \diamond \sigma'_{i-1}|_{\mathcal{V}(s_{\setminus(f,i)})} \diamond \sigma'_{i+1}|_{\mathcal{V}(s_{\setminus(f,i)})} \diamond \dots \diamond \sigma'_m|_{\mathcal{V}(s_{\setminus(f,i)})} \\
 & && \text{as } s_i \notin \mathcal{V}(s_{\setminus(f,i)}) \text{ since } s \text{ is linear} \\
 &= \sigma'_1|_{\mathcal{V}((s_1)_{\setminus(f,i)})} \diamond \dots \diamond \sigma'_{i-1}|_{\mathcal{V}((s_{i-1})_{\setminus(f,i)})} \diamond \sigma'_{i+1}|_{\mathcal{V}((s_{i+1})_{\setminus(f,i)})} \diamond \dots \diamond \sigma'_m|_{\mathcal{V}((s_m)_{\setminus(f,i)})} \\
 & && \text{as } \text{dom}(\sigma_j) \subseteq \mathcal{V}(s_j) \\
 &= \theta|_{\mathcal{V}((s_1)_{\setminus(f,i)})} \diamond \dots \diamond \theta|_{\mathcal{V}((s_{i-1})_{\setminus(f,i)})} \diamond \theta|_{\mathcal{V}((s_{i+1})_{\setminus(f,i)})} \diamond \dots \diamond \theta|_{\mathcal{V}((s_m)_{\setminus(f,i)})} \\
 & && \text{as } \sigma'_j|_{\mathcal{V}((s_j)_{\setminus(f,i)})} = (\sigma_j)_{\setminus(f,i)}|_{\mathcal{V}((s_j)_{\setminus(f,i)})} = \theta|_{\mathcal{V}((s_j)_{\setminus(f,i)})} \\
 &= \theta|_{\mathcal{V}(s_{\setminus(f,i)})} && \text{as } s \text{ is linear.}
 \end{aligned}$$

This finishes the proof of (9.30).

Now we prove that

$$s' \xrightarrow{\mathcal{R}_{\setminus(f,i)}} t' \text{ and } s' = s_{\setminus(f,i)} \text{ implies } s \xrightarrow{\mathcal{R}} t \text{ for some } t \text{ with } t' = t_{\setminus(f,i)}.$$

Then $s'_0 \xrightarrow{\mathcal{R}_{\setminus(f,i)}^*} s'_m$ with $s'_0 \in \mathcal{T}_{\text{basic}}(\mathcal{R}_{\setminus(f,i)})$ implies that there is an s_m such that $s_0 \xrightarrow{\mathcal{R}} s_m$ where $s_0 \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ results from s'_0 by adding a fresh variable as i^{th} argument to every occurrence of f . Thus, as observed above, we have $|s_0| \leq 2 \cdot |s'_0|$, which suffices to prove the theorem.

Let $\ell_{\setminus(f,i)} \xrightarrow{\mathcal{R}} r_{\setminus(f,i)}$ and θ be the rule and the substitution of the rewrite step $s' \xrightarrow{\mathcal{R}_{\setminus(f,i)}} t'$, i.e., we have $s' = C[\ell_{\setminus(f,i)}\theta] \xrightarrow{\mathcal{R}_{\setminus(f,i)}} C[r_{\setminus(f,i)}\theta] = t'$ for some context C . Then $s' = C[\ell_{\setminus(f,i)}\theta] = s_{\setminus(f,i)}$ implies that s is of the form $D[p]$ where $D_{\setminus(f,i)} = C$ and $p_{\setminus(f,i)} = \ell_{\setminus(f,i)}\theta$. Thus, (9.30) implies that there is a substitution σ such that $\ell\sigma = p$ and $\sigma_{\setminus(f,i)}|_{\mathcal{V}(\ell_{\setminus(f,i)})} = \theta|_{\mathcal{V}(\ell_{\setminus(f,i)})}$. Hence, we have $s = D[p] = D[\ell\sigma] \xrightarrow{\mathcal{R}} D[r\sigma] = t$. It remains to show $t_{\setminus(f,i)} = t'$. We have

$$\begin{aligned}
 t_{\setminus(f,i)} &= D[r\sigma]_{\setminus(f,i)} \\
 &= D_{\setminus(f,i)}[(r\sigma)_{\setminus(f,i)}] && \text{by (9.29)} \\
 &= C[(r\sigma)_{\setminus(f,i)}] && \text{as } D_{\setminus(f,i)} = C \\
 &= C[r_{\setminus(f,i)}\sigma_{\setminus(f,i)}] && \text{by (9.28)} \\
 &= C[r_{\setminus(f,i)}(\sigma_{\setminus(f,i)}|_{\mathcal{V}(\ell_{\setminus(f,i)})})] && \text{as } \mathcal{V}(r_{\setminus(f,i)}) \subseteq \mathcal{V}(\ell_{\setminus(f,i)}) \\
 &= C[r_{\setminus(f,i)}\theta|_{\mathcal{V}(\ell_{\setminus(f,i)})}] && \text{as } \sigma_{\setminus(f,i)}|_{\mathcal{V}(\ell_{\setminus(f,i)})} = \theta|_{\mathcal{V}(\ell_{\setminus(f,i)})} \\
 &= C[r_{\setminus(f,i)}\theta] && \text{as } \mathcal{V}(r_{\setminus(f,i)}) \subseteq \mathcal{V}(\ell_{\setminus(f,i)}) \\
 &= t'.
 \end{aligned}$$

□

Clearly, the processor from Theorem 9.40 can also be used in combination with other techniques for the inference of lower bounds like the loop detection technique from Chapter 8. However, our experimental evaluation shows that loop detection does not benefit from our argument filtering technique, cf. Chapter 12.

In our implementation, as a heuristic we always perform argument filtering

9.7. ARGUMENT FILTERING

prior to the induction technique if it is permitted by Theorem 9.40, except for cases where filtering removes defined function symbols on right-hand sides of rules. As an example, consider $\mathcal{R} = \{\mathbf{a} \rightarrow \mathbf{f}(\mathbf{a}, \mathbf{b})\}$ where $\text{rc}_{cp(\mathcal{R})}(n) = \omega$ for $n \geq 1$. If one filters away \mathbf{f} 's first argument, then one obtains $\mathbf{a} \rightarrow \mathbf{f}(\mathbf{b})$ and thus, $\text{rc}_{cp(\mathcal{R}_{\setminus \{f,1\}})}(n) = 1$ for $n \geq 1$. So here, argument filtering is sound, but it results in significantly worse lower bounds.

9.8 Proving Innermost Conjectures

Like loop detection, the induction technique can easily be adapted to innermost rewriting. To do so, we first show that the step from innermost rewriting to innermost *equational* rewriting is sound.

Theorem 9.41 (From Term Rewriting to Rewrite Lemmas). *Let \mathcal{R} be a well-typed TRS over a standard signature Σ . Then the processor mapping $cp_i(\mathcal{R})$ to $cp_i(\mathcal{R}, \mathcal{G}_{\mathcal{R}})$ is sound for lower bounds.*

Proof. As in the proof of Theorem 9.13, $s \xrightarrow{i} t$ where s and t are ground implies $s \downarrow_{\mathcal{G}_{\mathcal{R}}^f/A} \xrightarrow{\mathcal{R}} t \downarrow_{\mathcal{G}_{\mathcal{R}}^f/A}$ where both rewrite steps use the rule $\ell \xrightarrow{\mathcal{R}} r$, the former rewrite step uses the substitution σ , and the latter rewrite step uses the substitution $\sigma' = \{x/x\sigma \downarrow_{\mathcal{G}_{\mathcal{R}}^f/A} \mid x \in \text{dom}(\sigma)\}$. By definition of \xrightarrow{i} , all proper subterms of $\ell\sigma$ are in \rightarrow -normal form. Since we have $\ell\sigma \equiv_{\mathcal{G} \cup \mathcal{A}} \ell\sigma'$ by definition of σ' , this implies that all proper subterms of $\ell\sigma'$ are in \rightarrow -normal form. Since $\xrightarrow{i} \mathcal{R} \subseteq \rightarrow$, this means that all proper subterms of $\ell\sigma'$ are in $\xrightarrow{i} \mathcal{R}$ -normal form. Thus, we get $s \downarrow_{\mathcal{G}_{\mathcal{R}}^f/A} \xrightarrow{i} t \downarrow_{\mathcal{G}_{\mathcal{R}}^f/A}$. \square

Moreover, we have to adapt our notion of valid conjectures.

Definition 9.42 (Validity of Innermost Conjectures). A conjecture $s \xrightarrow{i}^{n,?} t$ is *valid* for \mathcal{L} if $s\sigma \xrightarrow{i}^* t\sigma$ holds for all $\sigma : \mathcal{V}(s) \rightarrow \mathbb{N}$.

Finally, we have to use *non-overlapping rewriting modulo $\mathcal{G} \cup \mathcal{A}$* to prove the validity of conjectures (cf. Example 9.46, which illustrates why this is crucial for the correctness of our approach for innermost rewriting).

Definition 9.43 (Non-Overlapping Rewriting Modulo). Let \mathcal{L} be a set of rewrite lemmas and let \mathcal{E} be a set of equations. We define $s \xrightarrow{n}^{c'}_{\mathcal{L}/\mathcal{E}} t$ if there is a context C , a rewrite lemma $\ell \xrightarrow{\mathcal{L}} r \in \mathcal{L}$, and a substitution σ such that $s \equiv_{\mathcal{E}} C[\ell\sigma]$, $C[r\sigma] \equiv_{\mathcal{E}} t$, $c \equiv_{\mathcal{A}} c\sigma$, and $\ell\sigma \triangleright t \notin \mathcal{V}$ implies $t\theta \not\equiv_{\mathcal{E}} \ell'\theta$ for each substitution θ and each (variable-renamed) rule $\ell' \rightarrow r' \in \mathcal{L}$.

We write $s \xrightarrow{n} t$ instead of $s \xrightarrow{n}_{\mathcal{L}/\mathcal{G} \cup \mathcal{A}} t$. Again, we define $q \xrightarrow{n}_{\ell \rightarrow r, n} p$ if $q \xrightarrow{n} p$ or $c = 0$ and $q \mapsto_{\ell \rightarrow r, n} p$. Then the following theorem show how to prove validity of innermost conjectures.

Theorem 9.44 (Proving Conjectures). *Let $s \xrightarrow{i}^{n,?} t$ be a conjecture for \mathcal{L} with $n \in \mathcal{V}(s)$. If $s\{n/0\} \xrightarrow{i\beta}_{n}^* t\{n/0\}$ and $s\{n/n+1\} \xrightarrow{i\beta}_{n}^{+} t\{n/n+1\}$ for some $i\beta, i\beta \in \mathcal{T}(\Sigma_{\mathbb{N}}, \mathcal{V}(s))$, then the conjecture $s \xrightarrow{i}^{n,?} t$ is valid for \mathcal{L} .*

To prove Theorem 9.44, we need to adapt Lemma 9.27 for non-overlapping rewriting modulo.

9.8. PROVING INNERMOST CONJECTURES

Lemma 9.45 (Stability of Non-Overlapping Rewriting Modulo). *Let ℓ, r, s , and t be well-typed terms where s only contains variables of type \mathbb{N} , and let $\mu : \mathcal{V}(s) \rightarrow \mathbb{N}$. Then we have:*

- (a) $s \xrightarrow[n]{c} t$ implies $s\mu \xrightarrow[n]{c\mu} t\mu$
- (b) $s \xrightarrow[n]{c}_{\ell \rightarrow r, n} t$ implies that there is a substitution $\sigma : \mathcal{V}(\ell) \rightarrow \mathbb{N}$ with $n\sigma = n\mu$ and $m\sigma \geq m\mu$ for all $m \in \mathcal{V}(\ell)$ such that $s\mu \xrightarrow[n]{c\mu}_{\ell\sigma \rightarrow r\sigma, n} t\mu$.

Proof. We prove both claims individually.

- (a) By the definition of $\xrightarrow[n]{c}$, $s \xrightarrow[n]{c} t$ implies that there is a context C , a substitution σ , and a rule $\ell \xrightarrow{c'} r \in \mathcal{L}$ such that $s \equiv_{\mathcal{G} \cup \mathcal{A}} C[\ell\sigma]$, $C[r\sigma] \equiv_{\mathcal{G} \cup \mathcal{A}} t$, and $c'\sigma \equiv_{\mathcal{A}} c$. Moreover, $\ell\sigma \triangleright t \notin \mathcal{V}$ implies $t\theta \not\equiv_{\mathcal{E}} \ell'\theta$ for each substitution θ and each (variable-renamed) rule $\ell' \rightarrow r' \in \mathcal{L}$.

We clearly have $s\mu \equiv_{\mathcal{G} \cup \mathcal{A}} C[\ell\sigma]\mu$ and $C[r\sigma]\mu \equiv t\mu$.

Assume that there exists a proper non-variable subterm q' of $\ell\sigma\mu$ that unifies modulo $\mathcal{G} \cup \mathcal{A}$ with a variable-renamed left-hand side of a rule from \mathcal{L} . Since the root symbol of q' must be from $\Sigma_d(\mathcal{L})$ and the range of μ does not include any defined symbols, we must have $q' = q\mu$ for some term q that is a proper non-variable subterm of $\ell\sigma$. But then q would already unify modulo $\mathcal{G} \cup \mathcal{A}$ with a variable-renamed left-hand side of a rule from \mathcal{L} , which is a contradiction to $s \xrightarrow[n]{c} t$ above. Thus, we can also conclude $s\mu \xrightarrow[n]{c\mu} t\mu$.

- (b) If we also have $s \xrightarrow[n]{c}_{\ell \rightarrow r, n} t$, then the claim follows from (a). Otherwise, the proof is analogous to the proof of Lemma 9.27 (b). \square

Using Lemma 9.45, Theorem 9.44 can be proven analogously to Theorem 9.22. Furthermore, Theorem 9.26 and Theorem 9.31 trivially carry over to innermost rewriting, too. The reason is that they just determine the costs of the family of rewrite sequences whose existence is ensured by Theorem 9.22 resp. Theorem 9.44. Thereby, the question whether this family of rewrite sequences is innermost or not is irrelevant.

The following example illustrates why the restriction to non-overlapping rewriting in Theorem 9.44 is crucial.

CHAPTER 9. INDUCTION TECHNIQUE

Example 9.46. Let

$$\mathcal{R} = \{f(\text{succ}(x)) \rightarrow f(g(x)), f(g(x)) \rightarrow f(x), f(\text{zero}) \rightarrow \text{zero}, g(\text{zero}) \rightarrow a\}$$

and assume that we replace \xrightarrow{n} by \xrightarrow{i} Theorem 9.44. Then we have

$$f(\text{succ}^n(\text{zero}))\{n/0\} \xrightarrow{i} \text{zero}$$

and

$$\begin{array}{ccc} f(\text{succ}^n(\text{zero}))\{n/n+1\} & \xrightarrow[\frac{1}{i}]{\frac{1}{i}} & f(g(\text{succ}^n(\text{zero}))) \\ & & f(\text{succ}^n(\text{zero})) \\ & \mapsto f(\text{succ}^n(\text{zero})) \rightarrow \text{zero}, n & \text{zero}, \end{array}$$

i.e., we could prove that $f(\text{succ}^n(\text{zero})) \xrightarrow{i} \text{zero}$ is a valid innermost conjecture. However, for all $n > 0$, we have $f(\text{succ}^n(\text{zero})) \not\xrightarrow{i}^* \text{zero}$. Note that the step $f(g(\text{succ}^n(\text{zero}))) \xrightarrow{i} f(\text{succ}^n(\text{zero}))$ is overlapping, as $g(\text{succ}^n(\text{zero})) = g(\text{zero})$ for $n = 0$, i.e., by taking the generator equations into account, $g(\text{succ}^n(\text{zero}))$ unifies with a left-hand side. Thus, the rewrite sequences above violate the prerequisites of Theorem 9.44.

9.9 Induction Technique vs. Loop Detection

In this section, we investigate the relation between the two presented techniques for the inference of lower bounds for term rewriting, namely loop detection (Chapter 8) and the induction technique presented in the current chapter, and we highlight their respective strengths and weaknesses.

Conceptually, the induction technique has two main drawbacks: its efficiency is limited, since it heavily relies on equational unification, which is expensive in practice. Moreover, it builds upon several heuristics, which restrict its power. For instance, narrowing is used to speculate conjectures in Section 9.2, which is non-deterministic. Hence, heuristics are applied to reduce the search space and to decide when to stop narrowing. Consequently, the induction technique may fail due to unfavorable heuristic decisions during the construction of narrowing trees. Moreover, the definition of the generator functions in Definition 9.4 is a heuristic as well, i.e., $\text{gen}_\tau(n)$ should be a “suitable” term of type τ . However, there are examples where other generator functions than those in Definition 9.4 are needed.

Example 9.47 (Failure due to Inappropriate Generator Functions). Reconsider Example 7.2. If one uses the heuristic of Definition 9.4 for the choice of generator functions, then gen_{List} only yields lists of zeros. However, for such inputs the complexity of `contains` is constant, i.e., then one cannot prove the desired linear lower bound.

In contrast, the loop detection technique from Chapter 8 does not require equational unification (i.e., it is more efficient than the induction technique). Moreover, it is not based on type inference and generator equations. Thus, it avoids the problems that are due to the heuristics in the induction technique. While the loop detection technique also applies narrowing, it only needs to find a *single* narrowing sequence satisfying a specific condition. In contrast, the induction technique requires *multiple* narrowing sequences which are suitable for generalization, resulting in a narrowing *tree*.

However, as illustrated with the Quicksort-TRS of Example 9.1, the induction technique can also infer *super-linear polynomial bounds*, which is not possible with loop detection. Thus, the induction technique is indispensable in practice. For *linear* lower bounds, loop detection is an extremely powerful technique as mentioned in Section 8.3. In fact, as shown in [55, Theorem 38], regarding the inference of linear lower bounds for ordinary left linear TRSs, it even subsumes the induction technique. Moreover, the induction technique fails for Example 7.10, while loop detection proves an exponential lower bound. However, for exponential bounds, the induction technique and loop detection are orthogonal, as the following example shows.

Example 9.48. Consider the TRS \mathcal{R} with the following rules:

$$\begin{aligned} f(\text{zero}) &\rightarrow \text{zero} \\ f(\text{succ}(x)) &\rightarrow \text{succ}(f(f(x))) \end{aligned}$$

Here, f rewrites its only argument to itself in exponentially many steps. The induction technique can prove the rewrite lemma

$$f(\text{gen}_{\text{Nat}}(n)) \xrightarrow{c} \text{gen}_{\text{Nat}}(n)$$

where c is exponential, i.e., the induction technique can prove an exponential lower bound. The reason is that the induction hypothesis is applied twice in the proof of the induction step. However, Theorem 8.12 cannot infer an exponential lower bound by loop detection. The reason is that in this TRS, there is no function symbol with an arity greater than 1. With such symbols one cannot construct terms that have two independent positions ι_1, ι_2 . Hence, there are no two compatible decreasing loops.

Hence, for exponential bounds, there exist examples where the induction technique is successful whereas loop detection fails and vice versa. As the orthogonality of both techniques also becomes evident in our experimental evaluation (cf. Chapter 12), both techniques should be used in practice, as it is done by recent versions of our tool AProVE. See [55] for a more detailed comparison of loop detection and the induction technique.

9.10 Related Work

Except for the loop detection technique from Chapter 8 (which has been compared with the induction technique in detail in Section 9.9), there are no other techniques for the inference of lower bounds on the complexity of term rewriting. Apart from loop detection, techniques to prove non-termination of TRSs are most closely related to our work. Among these, techniques to prove looping non-termination are more similar to the loop detection technique and have been discussed in detail in Section 8.5. Thus, we now focus on techniques to prove *non-looping* non-termination.

The approach to prove non-looping non-termination which is most closely related to our technique is [41]. As mentioned in Chapter 6, it also generates “meta rules” which represent families of rewrite sequences. However, it is unclear whether the calculus from [41] can be extended in order to infer lower bounds on the length of the rewrite sequences represented by these meta rules. Moreover, the meta rules from [41] are only parameterized with a single variable, whereas our rewrite lemmas may have several arguments of type \mathbb{N} . Consequently, common rewrite lemmas like

$$\text{times}(\text{gen}_{\text{Nat}}(n), \text{gen}_{\text{Nat}}(m)) \rightarrow \text{gen}_{\text{Nat}}(n \cdot m)$$

cannot be represented with the meta rules from [41]. On the other hand, [41] does not rely on expensive techniques like equational unification. Thus, the ideas from [41] and the current chapter are orthogonal.

Besides the approach from [41], the only other technique which can detect non-looping non-termination of term rewrite systems is [43]. There, the underlying idea is to find a tree-automaton that accepts a non-empty language \mathcal{L} which is closed under rewriting and does not contain normal forms. To this end, a SAT encoding is used. The existence of such a language proves non-termination, as any rewrite sequence starting in $t \in \mathcal{L} \neq \emptyset$ is non-terminating. The reason is that the existence of a rewrite sequence $t \rightarrow_{\mathcal{R}}^* s$ where s is a normal form would be a violation of \mathcal{L} ’s closure under rewriting (if $s \notin \mathcal{L}$) or of the property that \mathcal{L} does not contain normal forms (if $s \in \mathcal{L}$). Clearly, this approach is fundamentally different from ours.

9.11 Conclusion and Future Work

We presented the *induction technique*, which uses inductive theorem proving to infer lower bounds on the runtime complexity of term rewrite systems. The basic idea is to speculate and prove *conjectures* representing families of rewrite sequences. Thereby, *narrowing* is used to find representative rewrite sequences, so-called *sample conjectures*, which are suitable for generalization (Section 9.2). The generalized conjectures are then proven via induction, resulting in *rewrite lemmas* (Section 9.3).

To achieve a symbolic representation of families of rewrite sequences and to facilitate the inductive proofs, data structures are abstracted using *generator symbols* in conjectures and rewrite lemmas. The semantics of these generator symbols is provided by a set of *generator equations*. Consequently, these generator equations have to be taken into account when rewrite lemmas are used to speculate and prove further conjectures. To this end, the induction technique relies on *rewriting* and *narrowing modulo equations*.

While the complexity of a rewrite lemma can be obtained from its inductive proof by solving recurrence equations, we also showed how to obtain the (asymptotic) complexity of a rewrite lemma directly, i.e., without solving recurrence equations (Section 9.4). By taking the size of instances of left-hand sides of rewrite lemmas into account, one finally obtains a lower bound on the complexity of the analyzed TRS (Section 9.5).

One drawback of the induction technique is that speculating valid conjectures is challenging in practice. Moreover, our technique to prove conjectures may fail, even if the conjecture is valid. Thus, we presented two techniques which allow us to infer a non-trivial lower bound even if we fail to speculate or prove a conjecture for the original TRS: *Indefinite lemmas* (Section 9.6) allow us to reason about families of rewrite sequences without knowing their result and *argument filtering* (Section 9.7) allows us to simplify the TRS by discarding arguments which do not influence its complexity, but potentially complicate the inference of rewrite lemmas.

Finally, we also showed how to adapt our approach to *innermost rewriting* (Section 9.8).

In future work, one should generalize the concept of indefinite lemmas, as it significantly improves the power of the induction technique in practice (cf. Chapter 12). For example, one may consider *partially* indefinite rewrite lemmas, where only proper subterms of the right-hand side are unknown. Furthermore, one should investigate different heuristics to speculate conjectures. The reason is that the heuristic from Section 9.2 is based on narrowing, which is highly non-deterministic. Thus, implementing it is delicate. Another problem which requires further attention is the handling of non-homogeneous data structures like, e.g., lists with several different elements. While the argument filtering technique from Section 9.7 is an adequate way to deal with such data structures in some cases, its gain is limited in practice (cf. Chapter 12). Hence, one should

9.11. CONCLUSION AND FUTURE WORK

develop more powerful argument filters and one should also try to deal with non-homogeneous data structures directly, e.g., by supporting different generator functions than those introduced in Section 9.1. Finally, a more direct adaption of the approach from [41] (cf. Section 9.10) might help to bypass expensive techniques like equational unification.

See Chapter 12 for an extensive experimental evaluation of the induction technique.

Deciding Constant Upper Bounds

So far, we have seen two techniques that allow us to infer *lower* bounds for term rewriting. The inference of *upper* bounds for term rewriting has already been widely studied. Nevertheless, this chapter introduces a novel technique to deduce upper bounds for term rewrite systems. More precisely, we present a semi-decision procedure to prove $\text{rc}_{cp}(\mathcal{R})(n) \in \mathcal{O}(1)$ resp. $\text{rc}_{cp_i}(\mathcal{R})(n) \in \mathcal{O}(1)$ for ordinary TRSs, i.e., for TRSs where all rules have cost 1. Hence, in this chapter we prove that the question whether the runtime complexity of an ordinary TRS is constant is semi-decidable. This complements our results from Section 8.3, where we proved that the question whether the runtime complexity of a TRS is at least linear is undecidable.

While one usually considers the ability to provide guarantees w.r.t. the resource usage as the main motivation for the inference of upper complexity bounds, constant bounds are also important for the detection of bugs. The reason is that the runtime of non-trivial algorithms is usually not constant. Hence, if a complexity analysis tool can infer a constant upper bound, then this is often due to a programming error like, e.g., an unsatisfiable loop condition. Thus, the technique presented in the current chapter can be used to detect such bugs in term rewrite systems and hence, as discussed in Section 1.2, also in programs operating on tree-shaped data.

In the following, we present semi-decision procedures for $\text{rc}_{cp}(\mathcal{R})(n) \in \mathcal{O}(1)$ and $\text{rc}_{cp_i}(\mathcal{R})(n) \in \mathcal{O}(1)$ in Sections 10.1 and 10.2. After discussing related work in Section 10.3, we conclude in Section 10.4. See Chapter 12 for a detailed experimental evaluation of the presented techniques.

10.1 Constant Upper Bounds via Narrowing

To implement a semi-decision procedure to prove $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$, we exploit the observation from Section 8.3 that the runtime complexity of a TRS \mathcal{R} is constant if and only if $\leadsto_{\mathcal{R}}$ terminates on basic terms, i.e., if all narrowing sequences $t_0 \xrightarrow{\sigma_1}_{\mathcal{R}} t_1 \xrightarrow{\sigma_2}_{\mathcal{R}} \dots$ where $t_0 \sigma_1 \dots \sigma_n$ is a basic term for each $n \in \mathbb{N}$ are finite.¹ From now on, we call such narrowing sequences *constructor-based*.² The following example illustrates how the correspondence between termination of constructor-based narrowing and constant runtime can be used to infer constant upper bounds.

Example 10.1 (Constant Upper Bounds vs. Termination of Narrowing). The following TRS is a variation of SK90/4.51 from the *Termination Problems Data Base* where two rules which are not reachable from basic terms were removed for the sake of clarity.

$$f(a) \rightarrow g(h(a)) \qquad h(g(x)) \rightarrow g(h(f(x)))$$

Its defined symbols are f and h . Narrowing $f(x)$ terminates after one step:

$$f(x) \xrightarrow{\{x/a\}} g(h(a))$$

Similarly, narrowing $f(a)$ terminates after one step. Let $t \notin \mathcal{V}$ be a constructor term. If $t \neq a$, then $f(t)$ is a normal form w.r.t. \leadsto . For $h(x)$, we get

$$h(x) \xrightarrow{\{x/g(x')\}} g(h(f(x'))) \xrightarrow{\{x'/a\}} g(h(g(h(a)))) \xrightarrow{\text{id}} g(g(h(f(h(a)))))$$

Similarly, narrowing $h(g(x))$ or $h(g(a))$ terminates after three steps. If $t \neq a$, then narrowing $h(g(t))$ terminates after a single step. Finally, if $\text{root}(t) \neq g$, then $h(t)$ is a normal form w.r.t. \leadsto . Since the cases consider above cover all constructor-based narrowing sequences (up to variable renaming), this proves that the runtime complexity of the TRS is constant.

In contrast, if we change the second rule to $h(g(x)) \rightarrow g(h(x))$, then the runtime complexity becomes linear and we obtain the non-terminating narrowing sequence

$$h(x) \xrightarrow{\{x/g(x')\}} g(h(x')) \xrightarrow{\{x'/g(x'')\}} g(g(h(x''))) \xrightarrow{\{x''/g(x''')\}} \dots$$

Unfortunately, the reasoning from Example 10.1 is hard to automate, since we explicitly reasoned about all narrowing sequences starting with one out of infinitely many basic terms. Thus, to enable automation, we exploit the fact

¹More precisely, we proved that we have $\text{rc}_{cp(\mathcal{R})}(n) \in \Omega(n)$ if and only if there is such an infinite narrowing sequence. However, this is equivalent to the statement above due to Corollary 8.17.

²Note that our notion of constructor-based narrowing differs from *basic narrowing* [11, 89], where one is not allowed to narrow subterms which have been introduced by preceding narrowing substitutions.

10.1. CONSTANT UPPER BOUNDS VIA NARROWING

that narrowing sequences can be “generalized” in such a way that termination of constructor-based narrowing can be proven by just considering finitely many start terms. Then a semi-decision procedure for termination of constructor-based narrowing can be obtained by enumerating narrowing sequences with increasing length.

However, in Section 8.3 we only stated the equivalence between constant runtime and termination of constructor-based narrowing for a quite restricted class of TRSs. Thus, to use this semi-decision procedure to prove constant upper bounds for arbitrary ordinary TRSs, we need to generalize the ideas from Section 8.3.

We first introduce the generalization technique for narrowing sequences mentioned above and prove the equivalence between constant runtime and termination of constructor-based narrowing for ordinary TRSs afterwards. Our generalization technique is based on the following partial ordering on narrowing sequences, which clarifies which narrowing sequences are considered to be more general than others in our setting.

Definition 10.2 (Ordering Narrowing Sequences). Let \mathcal{R} be a TRS and let

$$\begin{array}{ccccccc} t_0 & \xrightarrow[\mathcal{R}]{\sigma_1} & t_1 & \xrightarrow[\mathcal{R}]{\sigma_2} & \dots & \xrightarrow[\mathcal{R}]{\sigma_m} & t_m \quad \text{and} \\ p_0 & \xrightarrow[\mathcal{R}]{\theta_1} & p_1 & \xrightarrow[\mathcal{R}]{\theta_2} & \dots & \xrightarrow[\mathcal{R}]{\theta_m} & p_m \end{array}$$

be narrowing sequences. We have $t_0 \rightsquigarrow_{\mathcal{R}}^m t_m \succeq p_0 \rightsquigarrow_{\mathcal{R}}^m p_m$ if there is a substitution η such that $t_i \sigma_{i+1} \dots \sigma_m \eta = p_i \theta_{i+1} \dots \theta_m$ for all $i \in \{0, \dots, m\}$.

In the following, we call substitutions σ such that $\text{rng}(\sigma) \subseteq \mathcal{T}(\Sigma_c(\mathcal{R}), \mathcal{V})$ *constructor substitutions*. The next lemma is the foundation of our generalization technique, as it shows that every constructor-based narrowing sequence is a specialization (w.r.t. \succeq) of a sequence starting with a basic term $f(x_1, \dots, x_n)$ where x_1, \dots, x_n are pairwise different variables. Thus, it allows us to reason about termination of constructor-based narrowing by just considering sequences stating with such basic terms. This is the foundation of our semi-decision procedure, as there are just finitely many such terms (up to variable renaming).

Lemma 10.3 (Generalizing Narrowing Sequences). *Let \mathcal{R} be a TRS, let $m \in \mathbb{N}$ and let $p_0 \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ with $\text{root}(p_0) = f$ such that $p_0 \rightsquigarrow_{\mathcal{R}}^m p_m$ is constructor-based. Then we have $f(x_1, \dots, x_k) = t_0 \rightsquigarrow_{\mathcal{R}}^m t_m$ for pairwise different variables x_1, \dots, x_k where $t_0 \rightsquigarrow_{\mathcal{R}}^m t_m \succeq p_0 \rightsquigarrow_{\mathcal{R}}^m p_m$.*

Proof. Note that $\theta|_{\mathcal{V}(p_0)}$ is a constructor substitution, as the narrowing sequence $p_0 \rightsquigarrow_{\mathcal{R}}^m p_m$ is constructor-based. We use induction on m to prove that

$$\begin{aligned} p_0 \xrightarrow[\mathcal{R}]{\theta_1} \dots \xrightarrow[\mathcal{R}]{\theta_m} p_m \text{ where } (\theta_1 \diamond \dots \diamond \theta_m)|_{\mathcal{V}(p_0)} \text{ is a constructor substitution} \\ \text{implies } f(x_1, \dots, x_k) = t_0 \xrightarrow[\mathcal{R}]{\sigma_1} \dots \xrightarrow[\mathcal{R}]{\sigma_m} t_m \succeq p_0 \xrightarrow[\mathcal{R}]{\theta_1} \dots \xrightarrow[\mathcal{R}]{\theta_m} p_m \end{aligned}$$

CHAPTER 10. DECIDING CONSTANT UPPER BOUNDS

where $\text{root}(p_i|_\tau) \in \Sigma_d(\mathcal{R})$ implies $\text{root}(t_i|_\tau) \in \Sigma_d(\mathcal{R})$ for each $i \in \{0, \dots, m\}$ and each $\tau \in \text{pos}(p_i)$. The case $m = 0$ is trivial.

In the induction step, we have

$$p_0 \xrightarrow{\theta_1}_{\mathcal{R}} \dots \xrightarrow{\theta_{m+1}}_{\mathcal{R}} p_{m+1} \text{ where } (\theta_1 \diamond \dots \diamond \theta_{m+1})|_{\mathcal{V}(p_0)} \\ \text{is a constructor substitution.}$$

Note that we have $p_0\theta_1 \dots \theta_m \xrightarrow{m}_{\mathcal{R}} p_m$ and hence $\mathcal{V}(p_m) \subseteq \mathcal{V}(p_0\theta_1 \dots \theta_m)$. Thus, $\theta_{m+1}|_{\mathcal{V}(p_m)}$ is a constructor substitution, too.

Let π and $\ell \rightarrow r \in \mathcal{R}$ be the position and rule used for the last narrowing step $p_m \xrightarrow{\theta_{m+1}}_{\mathcal{R}} p_{m+1}$, i.e., we have $\theta_{m+1} = \text{mgu}(\ell, p_m|_\pi)$. The induction hypothesis implies

$$t_0 \xrightarrow{\sigma_1}_{\mathcal{R}} \dots \xrightarrow{\sigma_m}_{\mathcal{R}} t_m \succeq p_0 \xrightarrow{\theta_1}_{\mathcal{R}} \dots \xrightarrow{\theta_m}_{\mathcal{R}} p_m$$

where $\text{root}(p_i|_\tau) \in \Sigma_d(\mathcal{R})$ implies $\text{root}(t_i|_\tau) \in \Sigma_d(\mathcal{R})$ for all $i \in \{0, \dots, m\}$ and all $\tau \in \text{pos}(p_i)$. Moreover, it implies that there is a substitution η such that $t_i\sigma_{i+1} \dots \sigma_m\eta = p_i\theta_{i+1} \dots \theta_m$ for all $i \in \{0, \dots, m\}$. Hence, there is a substitution η such that

$$t_m\eta\theta_{m+1} = p_m\theta_{m+1} = p_m[\ell]_\pi\theta_{m+1}. \quad (10.1)$$

W.l.o.g.,

$$\bigcup_{i=0}^m \mathcal{V}(t_i\sigma_{i+1} \dots \sigma_m), \mathcal{V}(p_m), \text{ and } \mathcal{V}(\ell) \text{ are disjoint.} \quad (10.2)$$

Since we have $\text{root}(p_m|_\pi) = \text{root}(\ell) \in \Sigma_d(\mathcal{R})$ and thus also $\text{root}(t_m|_\pi) \in \Sigma_d(\mathcal{R})$, (10.1) and (10.2) imply that $t_m|_\pi$ is a non-variable subterm of t_m that unifies with ℓ . Let $\sigma_{m+1} = \text{mgu}(t_m|_\pi, \ell)$. Then we have

$$t_m \xrightarrow{\sigma_{m+1}}_{\mathcal{R}} t_m[r]_\pi\sigma_{m+1} = t_{m+1}. \quad (10.3)$$

We now prove that

$$\text{root}(p_{m+1}|_\tau) \in \Sigma_d(\mathcal{R}) \text{ implies } \text{root}(t_{m+1}|_\tau) \in \Sigma_d(\mathcal{R}).$$

If $\tau \in \text{pos}(p_m[r]_\pi)$, then we have

$$\begin{aligned} \text{root}(p_{m+1}|_\tau) &\in \Sigma_d(\mathcal{R}) \\ \iff \text{root}(p_m[r]_\pi\theta_{m+1}|_\tau) &\in \Sigma_d(\mathcal{R}) \text{ as } p_{m+1} = p_m[r]_\pi\theta_{m+1} \\ \iff \text{root}(p_m[r]_\pi|_\tau) &\in \Sigma_d(\mathcal{R}) \text{ as } \tau \in \text{pos}(p_m[r]_\pi) \\ \implies \text{root}(t_m[r]_\pi|_\tau) &\in \Sigma_d(\mathcal{R}) \text{ by the induction hypothesis} \\ \implies \text{root}(t_m[r]_\pi\sigma_{m+1}|_\tau) &\in \Sigma_d(\mathcal{R}) \\ \iff \text{root}(t_{m+1}|_\tau) &\in \Sigma_d(\mathcal{R}) \text{ as } t_{m+1} = t_m[r]_\pi\sigma_{m+1}. \end{aligned}$$

If $\tau \notin \text{pos}(p_m[r]_\pi)$, then we have $\tau \geq \pi$, as $\theta_{m+1}|_{\mathcal{V}(p_m)}$ is a constructor substitution. Thus, there are positions τ_r and τ' such that $\tau = \pi.\tau_r.\tau'$ and $r|_{\tau_r} = x \in \mathcal{V}$.

10.1. CONSTANT UPPER BOUNDS VIA NARROWING

Since $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$, there is also a position τ_ℓ such that $\ell|_{\tau_\ell} = x$. Thus, we get

$$\begin{aligned}
& \text{root}(p_{m+1}|_\tau) && \in \Sigma_d(\mathcal{R}) \\
\iff & \text{root}(p_m[r]_\pi \theta_{m+1}|_\tau) && \in \Sigma_d(\mathcal{R}) \quad \text{as } p_{m+1} = p_m[r]_\pi \theta_{m+1} \\
\iff & \text{root}(p_m[r]_\pi \theta_{m+1}|_{\pi.\tau_r.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } \tau = \pi.\tau_r.\tau' \\
\iff & \text{root}(p_m[\ell]_\pi \theta_{m+1}|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } r|_{\tau_r} = \ell|_{\tau_\ell} \\
\iff & \text{root}(p_m \theta_{m+1}|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } \theta_{m+1} = \text{mgu}(\ell, p_m|_\pi) \\
\iff & \text{root}(p_m|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } \theta_{m+1}|_{\mathcal{V}(p_m)} \text{ is a} \\
& && \text{constructor substitution} \\
\implies & \text{root}(t_m|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{by the induction hypothesis} \\
\implies & \text{root}(t_m \sigma_{m+1}|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \\
\iff & \text{root}(t_m[\ell]_\pi \sigma_{m+1}|_{\pi.\tau_\ell.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } \sigma_{m+1} = \text{mgu}(\ell, t_m|_\pi) \\
\iff & \text{root}(t_m[r]_\pi \sigma_{m+1}|_{\pi.\tau_r.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } r|_{\tau_r} = \ell|_{\tau_\ell} \\
\iff & \text{root}(t_{m+1}|_{\pi.\tau_r.\tau'}) && \in \Sigma_d(\mathcal{R}) \quad \text{as } t_{m+1} = t_m[r]_\pi \sigma_{m+1} \\
\iff & \text{root}(t_{m+1}|_\tau) && \in \Sigma_d(\mathcal{R}) \quad \text{as } \tau = \pi.\tau_r.\tau'.
\end{aligned}$$

It remains to show that there is a substitution η' such that

$$t_i \sigma_{i+1} \dots \sigma_{m+1} \eta' = p_i \theta_{i+1} \dots \theta_{m+1} \text{ for all } i \in \{0, \dots, m+1\}.$$

Since t_m , p_m , and ℓ are variable disjoint, (10.1) implies that

$$\mu = (\eta \diamond \theta_{m+1})|_{\mathcal{V}(\mathcal{V}(p_m) \cup \mathcal{V}(\ell))} \cup \theta_{m+1}|_{\mathcal{V}(p_m) \cup \mathcal{V}(\ell)} \quad (10.4)$$

is a unifier of t_m , p_m , and $p_m[\ell]_\pi$. Then μ is also a unifier of $t_m|_\pi$ and ℓ . Since $\sigma_{m+1} = \text{mgu}(t_m|_\pi, \ell)$, there is a substitution η' such that $\sigma_{m+1} \diamond \eta' = \mu$. Thus, we get

$$\begin{aligned}
& t_{m+1} \eta' \\
= & t_m[r]_\pi \sigma_{m+1} \eta' \quad \text{as } t_{m+1} = t_m[r]_\pi \sigma_{m+1} \\
= & t_m[r]_\pi \mu \quad \text{as } \sigma_{m+1} \diamond \eta' = \mu \\
= & t_m \eta[r]_\pi \theta_{m+1} \quad \text{by (10.2) and (10.4) since } \mathcal{V}(r) \subseteq \mathcal{V}(\ell) \\
= & p_m[r]_\pi \theta_{m+1} \quad \text{as } t_m \eta = p_m \text{ by the induction hypothesis} \\
= & p_{m+1} \quad \text{as } p_{m+1} = p_m[r]_\pi \theta_{m+1}.
\end{aligned}$$

For all $i \in \{0, \dots, m\}$ we have

$$\begin{aligned}
& t_i \sigma_{i+1} \dots \sigma_{m+1} \eta' \\
= & t_i \sigma_{i+1} \dots \sigma_m \mu \quad \text{as } \sigma_{m+1} \diamond \eta' = \mu \\
= & t_i \sigma_{i+1} \dots \sigma_m \eta \theta_{m+1} \quad \text{by (10.2) and (10.4)} \\
= & p_i \theta_{i+1} \dots \theta_m \theta_{m+1} \quad \text{as } t_i \sigma_{i+1} \dots \sigma_m \eta = p_i \theta_{i+1} \dots \theta_m \\
& \quad \text{by the induction hypothesis.}
\end{aligned}$$

□

CHAPTER 10. DECIDING CONSTANT UPPER BOUNDS

Example 10.4. Reconsider the narrowing sequences from Example 10.1. Indeed, all mentioned narrowing sequence are specializations of (prefixes of) the presented narrowing sequences starting with $f(x)$ resp. $h(x)$. For example, we have:

$$\begin{array}{ccccccc}
 h(x) & \xrightarrow{\{x/g(x')\}} & g(h(f(x'))) & \xrightarrow{\{x'/a\}} & g(h(g(h(a)))) & \xrightarrow{id} & g(g(h(f(h(a)))) \\
 \downarrow \{x/g(x')\} \circ \{x'/a\} & & \downarrow \{x'/a\} & & \downarrow id & & \downarrow id \\
 h(g(a)) & \xrightarrow{id} & g(h(f(a))) & \xrightarrow{id} & g(h(g(h(a)))) & \xrightarrow{id} & g(g(h(f(h(a))))
 \end{array}$$

The following theorem shows that constant runtime complexity is indeed equivalent to termination of constructor-based narrowing for arbitrary ordinary TRSs. Hence, it generalizes the corresponding result from Section 8.3, where we only considered a more restricted class of TRSs.

Theorem 10.5 (Termination of Narrowing means Constant Upper Bound). *Let \mathcal{R} be an ordinary TRS. We have $rc_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$ if and only if there is no infinite constructor-based narrowing sequence.*

Proof. For the “if” direction, assume $rc_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$. Then for each $m \in \mathbb{N}$ there is a rewrite sequence of length m starting with a basic term $f(\dots)$. Since $\Sigma_d(\mathcal{R})$ is finite, there exists an $f \in \Sigma_d(\mathcal{R})$ such that there are rewrite sequences $s_1 \xrightarrow{\mathcal{R}}^{m_1} q_1, s_2 \xrightarrow{\mathcal{R}}^{m_2} q_2, \dots$ with $m_1 < m_2 < \dots$ and $\text{root}(s_1) = \text{root}(s_2) = \dots = f$ where s_1, s_2, \dots are basic. By Lemma 10.3 this means that $f(x_1, \dots, x_k)$ starts narrowing sequences $f(x_1, \dots, x_k) \xrightarrow{\mathcal{R}}^{\sigma_1, m_1} t_1, f(x_1, \dots, x_k) \xrightarrow{\mathcal{R}}^{\sigma_2, m_2} t_2, \dots$ where $f(x_1, \dots, x_k)\sigma_1, f(x_1, \dots, x_k)\sigma_2, \dots$ match s_1, s_2, \dots . Note that Lemma 10.3 is applicable as every rewrite sequence is also a valid narrowing sequence (where the narrowing substitutions just instantiate variables in the applied rules, but not in the narrowed terms). Hence, $f(x_1, \dots, x_k)\sigma_1, f(x_1, \dots, x_k)\sigma_2, \dots$ are basic. Thus, if we just consider constructor-based narrowing sequences $f(x_1, \dots, x_k) \xrightarrow{\mathcal{R}}^m t$, the narrowing tree with the root $f(x_1, \dots, x_k)$ still has infinitely many nodes. Since $\xrightarrow{\mathcal{R}}$ is finitely branching, by König’s Lemma the tree has an infinite path, i.e., there is an infinite constructor-based narrowing sequence starting with $f(x_1, \dots, x_k)$.

For the “only if” direction, assume that there is an infinite constructor-based narrowing sequence $t_0 \xrightarrow{\mathcal{R}}^{\sigma_1} t_1 \xrightarrow{\mathcal{R}}^{\sigma_2} \dots$. Then for each $c \in \mathbb{N}$, we have $t_0\sigma_1 \dots \sigma_{c+1} \xrightarrow{\mathcal{R}}^{c+1} t_{c+1}$. As $t_0\sigma_1 \dots \sigma_{c+1}$ is basic, this is a contradiction to $rc_{cp(\mathcal{R})}(\|t_0\sigma_1 \dots \sigma_{c+1}\|_t) \leq c$ and hence proves $rc_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$. \square

In combination with Lemma 10.3, Theorem 10.5 yields the main result of this chapter: The question whether a TRS has a constant upper bound is semi-decidable. The semi-decision procedure is presented in Algorithm 5.

Corollary 10.6 (Constant Upper Bounds are Semi-Decidable). *Algorithm 5 is a semi-decision procedure for $rc_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$ for all ordinary TRSs \mathcal{R} .*

10.1. CONSTANT UPPER BOUNDS VIA NARROWING

Algorithm 5 Semi-Decision Procedure for $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$

1. For each $f \in \Sigma_d(\mathcal{R})$
 - 1.1. Set $j := 0$
 - 1.2. Set $j := j + 1$
 - 1.3. For each narrowing sequence $f(x_1, \dots, x_k) \xrightarrow[\mathcal{R}]{\theta}^j t$
 - 1.3.1. If $f(x_1, \dots, x_k)\theta$ is basic
 - 1.3.1.1. Go to Step 1.2
 2. Return $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$
-

Proof. By Theorem 10.5, we have $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$ if and only if there is no infinite constructor-based narrowing sequence. To prove that all such narrowing sequences are finite, it suffices to prove that all constructor-based narrowing sequences that start with terms of the form $f(x_1, \dots, x_k)$ where $f \in \Sigma_d(\mathcal{R})$ and x_1, \dots, x_k are pairwise different variables are finite by Lemma 10.3.

Assume that there is an infinite constructor-based narrowing sequence

$$f(x_1, \dots, x_k) = t_0 \xrightarrow[\mathcal{R}]{\sigma_1} t_1 \xrightarrow[\mathcal{R}]{\sigma_2} \dots,$$

i.e., $\text{rc}_{cp(\mathcal{R})}(n) \notin \mathcal{O}(1)$. Then for each $n \in \mathbb{N}$, there is a narrowing sequence of length n such that the condition in Step 1.3.1 is satisfied and hence Step 1.3.1.1 is executed, i.e., Algorithm 5 does not terminate.

Now assume that there is no infinite constructor-based narrowing sequence, i.e., $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$. Let c be the length of the longest constructor-based narrowing sequence $f(x_1, \dots, x_k) = t_0 \xrightarrow[\mathcal{R}]{\sigma_1} t_1 \dots \xrightarrow[\mathcal{R}]{\sigma_c} t_c$. Then the outer Loop 1 is executed $|\Sigma_d(\mathcal{R})|$ times and in each iteration of the outer loop, the inner loop (from Step 1.2 to Step 1.3.1.1) is executed at most c times. Thus, Algorithm 5 terminates and returns $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$. \square

The following example illustrates how to prove a constant upper bound for Example 10.1 using our semi-decision procedure.

Example 10.7 (Constant Upper Bounds via Narrowing). Reconsider the variation of SK90/4.51 from the *Termination Problems Data Base* from Example 10.1. Until 2016, no tool proved that the runtime complexity of SK90/4.51 is constant at a *Termination and Complexity Competition* [121].³ Since the *Termination and Complexity Competition 2016*, AProVE can prove a constant upper bound via Algorithm 5. To do so, it suffices to construct the narrowing sequences starting with $f(x)$ and $h(x)$ from Example 10.1.

³Note that, without the technique presented in the current chapter, the two leading complexity analysis tools for term rewriting (AProVE and TCT) also fail to prove a constant upper bound for our streamlined version of SK90/4.51.

CHAPTER 10. DECIDING CONSTANT UPPER BOUNDS

Clearly, Algorithm 5 is also sound for non-ordinary TRSs, i.e., for TRSs where some rules have cost 0. However, for this class of TRSs, it is not a semi-decision procedure anymore. To see this, consider the TRS $\{f \xrightarrow{1} g, f \xrightarrow{0} f\}$. Its runtime complexity is constant, but it admits the non-terminating constructor-based narrowing sequence $f \rightsquigarrow f \rightsquigarrow \dots$

10.2 Constant Bounds for Innermost Rewriting

We now adapt the results from Section 10.1 to *innermost* rewriting. To this end, we adapt Theorem 10.5 such that only narrowing sequences that correspond to innermost rewrite sequences are taken into account.

Theorem 10.8 (Termination of Narrowing and Innermost Rewriting). *Let \mathcal{R} be an ordinary TRS. We have $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$ if and only if there is no infinite constructor-based narrowing sequence $t_0 \xrightarrow{\sigma_1}_{\mathcal{R}} t_1 \xrightarrow{\sigma_2}_{\mathcal{R}} \dots$ such that we have $t_0 \sigma_1 \dots \sigma_c \xrightarrow{i}_{\mathcal{R}} t_c$ for each $c \in \mathbb{N}$.*

Proof. For the “if” direction, assume $\text{rc}_{cp_i(\mathcal{R})}(n) \notin \mathcal{O}(1)$. Then for each $m \in \mathbb{N}$ there is an innermost rewrite sequence of length m starting with a basic term $f(\dots)$. Since $\Sigma_d(\mathcal{R})$ is finite, there exists an $f \in \Sigma_d(\mathcal{R})$ such that there are rewrite sequences $s_1 \xrightarrow{i}_{\mathcal{R}}^{m_1} q_1, s_2 \xrightarrow{i}_{\mathcal{R}}^{m_2} q_2, \dots$ with $m_1 < m_2 < \dots$ and $\text{root}(s_1) = \text{root}(s_2) = \dots = f$ where s_1, s_2, \dots are basic. By Lemma 10.3 this means that $f(x_1, \dots, x_k)$ starts narrowing sequences

$$f(x_1, \dots, x_k) \xrightarrow{\sigma_1}_{\mathcal{R}}^{m_1} t_1 \succeq s_1 \xrightarrow{i}_{\mathcal{R}}^{m_1} q_1, f(x_1, \dots, x_k) \xrightarrow{\sigma_2}_{\mathcal{R}}^{m_2} t_2 \succeq s_2 \xrightarrow{i}_{\mathcal{R}}^{m_2} q_2, \dots$$

Note that Lemma 10.3 is applicable as every rewrite sequence is also a valid narrowing sequence (where the narrowing substitutions just instantiate variables in the applied rules, but not in the narrowed terms). Then

$$f(x_1, \dots, x_k) \xrightarrow{\sigma_i}_{\mathcal{R}}^{m_i} t_i \succeq s_i \xrightarrow{i}_{\mathcal{R}}^{m_i} q_i$$

implies that $f(x_1, \dots, x_k)\sigma_i$ is basic and that $f(x_1, \dots, x_k)\sigma_i \xrightarrow{\mathcal{R}}^{m_i} t_i$ is an innermost rewrite sequence.⁴ Thus, if we just consider constructor-based narrowing sequences $f(x_1, \dots, x_k) \xrightarrow{\sigma}_{\mathcal{R}}^m t$ such that $f(x_1, \dots, x_k)\sigma \xrightarrow{i}_{\mathcal{R}}^m t$, the narrowing tree with the root $f(x_1, \dots, x_k)$ still has infinitely many nodes. Since $\xrightarrow{\mathcal{R}}$ is finitely branching, by König’s Lemma the tree has an infinite path, i.e., there is an infinite constructor-based narrowing sequence starting with $f(x_1, \dots, x_k)$.

For the “only if” direction, assume that there is an infinite narrowing sequence $t_0 \xrightarrow{\sigma_1}_{\mathcal{R}} t_1 \xrightarrow{\sigma_2}_{\mathcal{R}} \dots$ such that $t_0 \sigma_0 \dots \sigma_c$ is basic and $t_0 \sigma_1 \dots \sigma_c \xrightarrow{i}_{\mathcal{R}}^c t_c$ for each $c \in \mathbb{N}$. This contradicts $\text{rc}_{cp_i(\mathcal{R})}(|t_0 \sigma_1 \dots \sigma_{c+1}|) \leq c$ and hence proves $\text{rc}_{cp_i(\mathcal{R})}(n) \notin \mathcal{O}(1)$. \square

Example 10.9. Consider the TRS with the following rules:

$$a \rightarrow f(b) \quad f(b) \rightarrow f(b) \quad b \rightarrow c$$

Its *full* runtime complexity is not constant. To see this, it suffices to con-

⁴To see this, note that the narrowing sequences $t_0 \xrightarrow{\sigma}_{\mathcal{R}}^m t_m$ and $p_0 \xrightarrow{\theta}_{\mathcal{R}}^m p_m$ narrow the same positions in the same order according to the proof of Lemma 10.3.

CHAPTER 10. DECIDING CONSTANT UPPER BOUNDS

sider all constructor-based narrowing sequences starting with the terms \mathbf{a} , \mathbf{b} , and $f(x)$ as explained in Section 10.1. We have $f(x) \rightsquigarrow_{\{x/b\}} f(\mathbf{b})$, but this narrowing step is not constructor-based, as $f(x)\{x/b\} = f(\mathbf{b})$ is not a basic term. Moreover, we have $\mathbf{b} \rightsquigarrow^{\text{id}} \mathbf{c}$ where \mathbf{c} is a normal form w.r.t. \rightsquigarrow . For \mathbf{a} , we obtain $\mathbf{a} \rightsquigarrow^{\text{id}} f(\mathbf{b}) \rightsquigarrow^{\text{id}} f(\mathbf{c})$ and the non-terminating constructor-based narrowing sequence $\mathbf{a} \rightsquigarrow^{\text{id}} f(\mathbf{b}) \rightsquigarrow^{\text{id}} f(\mathbf{b}) \rightsquigarrow \dots$. However, since the corresponding rewrite sequence $\mathbf{a} \rightarrow f(\mathbf{b}) \rightarrow f(\mathbf{b}) \rightarrow \dots$ is not innermost (due to the inner redex \mathbf{b} in $f(\mathbf{b})$), we nevertheless get $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$ by Theorem 10.8.

Algorithm 6 Semi-Decision Procedure for $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$

1. For each $f \in \Sigma_d(\mathcal{R})$
 - 1.1. Set $j := 0$
 - 1.2. Set $j := j + 1$
 - 1.3. For each narrowing sequence $f(x_1, \dots, x_k) \xrightarrow{\mathcal{R}}^j t$
 - 1.3.1. If $f(x_1, \dots, x_k)\theta$ is basic and $f(x_1, \dots, x_k)\theta \xrightarrow{i}^j t$
 - 1.3.1.1. Go to Step 1.2
 2. Return $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$
-

Thus, as for full rewriting, we obtain a semi-decision procedure to prove $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$. The adaption of Algorithm 5 for innermost rewriting is presented in Algorithm 6.

Corollary 10.10 (Innermost Constant Upper Bounds are Semi-Decidable). *Algorithm 6 is a semi-decision procedure for $\text{rc}_{cp_i(\mathcal{R})}(n) \in \mathcal{O}(1)$ for all ordinary TRSs \mathcal{R} .*

Proof. The proof is analogous to the proof of Corollary 10.6. □

10.3 Related Work

There exists numerous techniques to prove upper bounds on the complexity of term rewrite systems [15, 79, 86, 87, 105, 125, 131]. The spectrum includes adaptations of the *Dependency Pair Framework* [66] from termination analysis of term rewriting [15, 79, 105], automata-based techniques [125], techniques based on relative rewriting [131], and adaptations of the *potential method* [120] for term rewriting [86, 87]. While some of these approaches use narrowing as an auxiliary technique to perform case analyses, we use narrowing as a standalone complexity analysis technique. Moreover, all of the techniques mentioned above are incomplete, whereas our technique is a semi-decision procedure for its specific use case. Finally, none of these techniques focuses on the inference of *constant* upper bounds.

In contrast, the tool **Oops** [42] is specifically designed for the inference of constant bounds, just like our technique. It checks whether the runtime complexity of a given TRS is constant for one of the following reasons:

- There are only finitely many basic terms, as all constructors or all defined symbols are constants. If all rewrite sequences starting with one of these terms are finite, then the runtime complexity of the TRS is constant. This can be tested by evaluating each basic term in all possible ways.
- There is no rule whose left-hand side is basic. Thus, no rule can be applied to any basic term.

Clearly, our technique succeeds in both scenarios and hence subsumes **Oops**.

A property which is closely related to constant runtime complexity is the *finite variant property*, which is, e.g., of interest in the context of equational unification. For example, [22, Section 9] states that finite runtime complexity implies *IR-boundedness*, which in turn implies the finite variant property. Thus, complexity analysis techniques such as the one presented in this chapter can be used to prove that a TRS has the finite variant property.

In [89], it is proven that termination of *basic narrowing* follows if the considered TRS is terminating and confluent and there is no infinite basic narrowing sequence starting with a right-hand side of a rule. The paper [11] generalizes this result by removing the superfluous preconditions that the TRS needs to be terminating and confluent. While these results are related to our semi-decision procedure for constructor-based narrowing, the respective restrictions of the narrowing relation differ. We disallow narrowing sequences $s \xrightarrow{\mathcal{R}}^* t$ where $s\sigma$ is not basic, whereas basic narrowing disallows narrowing steps that reduce subterms which have been introduced by preceding narrowing substitutions. Thus, basic narrowing does not terminate for the TRS \mathcal{R} with the rules

$$f(a) \rightarrow f(a) \qquad a \rightarrow b$$

CHAPTER 10. DECIDING CONSTANT UPPER BOUNDS

due to the infinite basic narrowing sequence $f(a) \rightsquigarrow_{\mathcal{R}} f(a) \rightsquigarrow_{\mathcal{R}} \dots$. In contrast, all constructor-based narrowing sequences w.r.t. \mathcal{R} have at most length 1.

However, termination of basic and constructor-based narrowing coincides for left-linear constructor systems. To see this, recall that it suffices to consider sequences starting with right-hand sides of rules to prove termination of basic narrowing [11, 89]. Instead, one can also consider all sequences starting with basic terms of the form $f(\mathbf{x})$ for pairwise different variables \mathbf{x} (which narrow to the right-hand sides of the TRS in one step). Note that for left-linear constructor systems, *all* narrowing sequences starting with basic terms are basic, as narrowing substitutions cannot introduce defined symbols for such TRSs. For the same reason, all narrowing sequences starting with basic terms are constructor based for such TRSs. Thus, the resulting semi-decision procedure for termination of basic narrowing w.r.t. constructor systems coincides with the semi-decision procedure for termination of constructor-based narrowing from Section 10.1.

10.4 Conclusion and Future Work

We presented a semi-decision procedure to prove $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$ for arbitrary ordinary TRSs (Section 10.1). The semi-decision procedure builds upon the observation that constant runtime complexity is equivalent to termination of a restricted form of narrowing (which we call *constructor-based narrowing*). Moreover, it exploits the fact that narrowing sequences can be “generalized” in such a way that termination of constructor-based narrowing can be proven by just considering finitely many start terms. Thus, a semi-decision procedure for termination of constructor-based narrowing (and hence for the inference of constant upper bounds) can be obtained by enumerating narrowing sequences with increasing length.

The resulting technique is able to prove constant upper bounds for TRSs where state-of-the-art complexity analysis tools failed to prove $\text{rc}_{cp(\mathcal{R})}(n) \in \mathcal{O}(1)$ so far, cf. Example 10.7. See Chapter 12 for a detailed experimental evaluation.

To increase the applicability of our technique to real-world programs (where one is often only interested in program runs with an eager evaluation strategy), we also adapted our technique to innermost rewriting, cf. Section 10.2.

Besides providing guarantees w.r.t. the resource consumption of programs, our technique can also be used to detect bugs. The reason is that constant runtime complexity often results from programming errors like unsatisfiable loop conditions.

In future work, one should consider applications of our technique in the context of equational unification, cf. Section 10.3.

Strategy Switching – From Full to Innermost Rewriting and Vice Versa

All complexity analysis techniques presented in this thesis so far apply to innermost as well as full rewriting. However, there are also many techniques which are specific to innermost rewriting [13, 14, 105]. In particular, the results of the annual *Termination and Complexity Competition* [121] show that automatic techniques to infer upper bounds for full rewriting are still substantially weaker than corresponding techniques for innermost rewriting. At the *Termination and Complexity Competitions* 2015 and 2016,¹ 899 examples were analyzed for both full and innermost runtime complexity.² For 235 of them, super-polynomial lower bounds were inferred for full rewriting. Hence, no upper bounds can be obtained for these examples since the participating tools only compute polynomial upper bounds. For the remaining 664 examples, a polynomial upper bound on the innermost runtime complexity was proven for 357 TRSs (53.8%) by at least one tool at one of the competitions. In contrast, a polynomial upper bound on the full runtime complexity was inferred for just 218 examples (32.8%).³

These numbers indicate that current techniques for complexity analysis of TRSs are much better in analyzing innermost than full runtime complexity, or that innermost runtime complexity is significantly easier to handle than full runtime complexity. In both cases, it is worthwhile to identify (decidable) classes of

¹Note that the results of the *Termination and Complexity Competition* 2017 were unavailable at the time of writing (even on request). Thus, the following numbers are based on the results of earlier competitions, whose results have been backed up by the author of this thesis, cf. https://aprove-developers.github.io/termcomp_results.

²We consider examples as equal if they have the same name. Note that the results of the *Termination and Complexity Competitions* 2015 and 2016 are orthogonal. On the one hand, the participating tools improved from 2015 to 2016, but on the other hand, the timeout per example was reduced from 300 s in 2015 to just 30 s in 2016. Hence, in the numbers above, we consider the best results of both competitions to represent the state of the art.

³Here, we ignore upper bounds on the full runtime complexity proven by our tool AProVE [62] in 2016. The reason is that at the *Termination and Complexity Competition* 2016, AProVE used a preliminary version of the new technique presented in the current chapter and we want to compare with the state of the art *before* the introduction of this technique. Before 2016, AProVE was not able to infer any upper bounds on the complexity of full rewriting.

CHAPTER 11. STRATEGY SWITCHING

TRSs where full and innermost runtime complexity coincide. In this chapter, we provide a criterion to prove that full and innermost runtime complexity coincide which is easy to automate. It builds upon an important result from [124] that a relaxation of innermost rewriting called *non-dup generalized innermost rewriting* (“*ndg* rewriting”) does not yield longer evaluation sequences than innermost rewriting itself. Our main contribution is a criterion to automatically identify classes of TRSs where *all* rewrite sequences starting with basic terms are *ndg*, which then implies that full and innermost runtime complexity coincide. Thus, our criterion allows us to switch between the rewrite strategies “full rewriting” and “innermost rewriting” during the analysis of a TRS.

Consequently, our *strategy switching* technique allow us to apply all existing and all *future* approaches specific to innermost rewriting to analyze full rewriting directly, which is particularly interesting for the inference of upper bounds. For the inference of lower bounds, our technique often allows us to use the unrestricted versions of loop detection (cf. Chapter 8) and the induction technique (cf. Chapter 8) for full rewriting even if we are interested in innermost runtime complexity. In other words, whenever the technique presented in the current chapter succeeds, then the additional restrictions for innermost rewriting from Section 8.4 and Section 9.8 are not required anymore.

However, in contrast to existing techniques for the inference of upper bounds, loop detection and the induction technique hardly benefit from strategy switching (cf. Chapter 12). Hence, in the context of program analysis strategy switching is especially interesting for the inference of upper bounds on the complexity of programs which are evaluated with a non-eager strategy, cf. Section 1.2.

In Section 11.1, we recall “*ndg* rewriting” and show that it is undecidable whether all rewrite sequences of a TRS are *ndg*. Hence, we develop a sufficient criterion for this property in Section 11.2 which is easy to check automatically. We implemented our contributions in the tool **AProVE** [62], resulting in a significant improvement of the state of the art in the automated analysis of full runtime complexity. See Chapter 12 for a detailed experimental evaluation. Finally, we discuss related work in Section 11.3 and conclude in Section 11.4.

11.1 Non-Dup Generalized Innermost Rewriting

In this section, we recall the definition of *ndg rewriting* from [124]. The idea of “ndg” is that variables occurring multiple times in right-hand sides of rules may only be instantiated by normal forms (we call such rewrite steps *spare*). So the main difference to full rewriting is that ndg rewriting does not allow rewrite steps that duplicate redexes. Moreover, proper subterms of left-hand sides with defined root may only be instantiated to normal forms. In Section 11.2, we show how to automatically prove that *every* rewrite sequence starting with a basic term is ndg.

Definition 11.1 (Spare and ndg Rewriting [124]). Let \mathcal{R} be a TRS with $\ell \rightarrow r \in \mathcal{R}$ such that $s \rightarrow_{\ell \rightarrow r, \pi} t$ where σ is the matcher with $\ell\sigma = s|_{\pi}$. The rewrite step $s \rightarrow_{\ell \rightarrow r, \pi} t$ is *spare* if $x\sigma$ is a normal form for all variables x with $\#_x(r) > 1$. It is *non-dup generalized innermost (ndg)*, denoted $s \xrightarrow{\text{ndg}}_{\ell \rightarrow r, \pi} t$, if it is spare and $\ell|_{\tau}\sigma$ is a normal form for all $\tau \in \text{pos}(\ell) \setminus \{\epsilon\}$ with $\text{root}(\ell|_{\tau}) \in \Sigma_d(\mathcal{R})$. \mathcal{R} is spare resp. ndg if every $\rightarrow_{\mathcal{R}}$ -sequence starting with a basic term only consists of spare resp. ndg rewrite steps.

Example 11.2. For \mathcal{R}_{fib} from Example 7.10, the rewrite step

$$\text{add}(\text{succ}(x), \text{add}(\text{zero}, z)) \rightarrow_{\mathcal{R}_{\text{fib}}} \text{add}(x, \text{succ}(\text{add}(\text{zero}, z)))$$

is ndg, but it is not innermost due to the redex $\text{add}(\text{zero}, z)$. In contrast,

$$\text{fib}(\text{succ}(\text{succ}(\text{add}(\text{zero}, z)))) \rightarrow_{\mathcal{R}_{\text{fib}}} \text{add}(\text{fib}(\text{succ}(\text{add}(\text{zero}, z))), \text{fib}(\text{add}(\text{zero}, z)))$$

is neither ndg nor spare, as the redex $\text{add}(\text{zero}, z)$ is duplicated.

Corollary 11.3 states two straightforward observations: innermost rewrite steps are ndg, since an innermost redex has no redexes as proper subterms. Moreover, sparseness and ndg are the same for so-called *overlay systems*, where no proper non-variable subterm of a left-hand side unifies with a redex.

Corollary 11.3 (Innermost Rewriting, Sparseness, and ndg).

- (a) Every innermost rewrite step is ndg, i.e., $\rightarrow_i_{\mathcal{R}} \subseteq \xrightarrow{\text{ndg}}_{\mathcal{R}}$.
- (b) Every spare overlay system is ndg.

The following examples show that, in general, full and innermost runtime complexity do not coincide if \mathcal{R} is not ndg.

Example 11.4. Consider the TRS \mathcal{R}_{nn} with the rules

$$c \rightarrow f(a) \quad f(a) \rightarrow f(a) \quad a \rightarrow b.$$

It is spare, but not ndg due to the rewrite sequence

$$c \rightarrow_{\mathcal{R}_{nn}} f(a) \rightarrow_{\mathcal{R}_{nn}} f(a) \rightarrow_{\mathcal{R}_{nn}} \dots$$

where the subterm a below the root of the left-hand side is not in normal form. Thus, we have $rc_{cp(\mathcal{R}_{nn})}(n) \in \Theta(\omega)$. However, the innermost runtime complexity of \mathcal{R}_{nn} is clearly constant.

Now consider the following TRS \mathcal{R}_{ns} :

$$f(\text{zero}, y) \rightarrow y \quad g(x) \rightarrow f(x, a) \quad f(\text{succ}(x), y) \rightarrow f(x, \text{node}(y, y)) \quad a \rightarrow b$$

It is not spare, because the sequence

$$g(\text{succ}^n(\text{zero})) \rightarrow_{\mathcal{R}_{ns}} f(\text{succ}^n(\text{zero}), a) \rightarrow_{\mathcal{R}_{ns}} f(\text{succ}^{n-1}(\text{zero}), \text{node}(a, a)) \rightarrow_{\mathcal{R}_{ns}} \dots$$

duplicates redexes. Here, we have $rc_{cp_i(\mathcal{R}_{ns})}(n) \in \Theta(n)$, but the full runtime complexity of \mathcal{R}_{ns} is exponential.

The next TRS \mathcal{R}_{nl} is a non-left-linear, but non-duplicating overlay system. It shows why for spareness it is not enough if $x\sigma$ is a normal form whenever $\#_x(\ell) < \#_x(r)$ (i.e., if x is duplicated):

$$g(\text{zero}, \text{succ}(\text{zero})) \rightarrow f(h, h) \quad f(x, x) \rightarrow g(x, x) \quad h \rightarrow \text{zero} \quad h \rightarrow \text{succ}(\text{zero})$$

We have $rc_{cp(\mathcal{R}_{nl})}(n) \in \Theta(\omega)$ due to the non-terminating rewrite sequence

$$g(\text{zero}, \text{succ}(\text{zero})) \rightarrow_{\mathcal{R}_{nl}} f(h, h) \rightarrow_{\mathcal{R}_{nl}} g(h, h) \rightarrow_{\mathcal{R}_{nl}}^2 g(\text{zero}, \text{succ}(\text{zero})) \rightarrow_{\mathcal{R}_{nl}} \dots$$

However, $rc_{cp_i(\mathcal{R}_{nl})}(n) \in \Theta(1)$ holds, as we have, e.g.,

$$g(\text{zero}, \text{succ}(\text{zero})) \rightarrow_i \mathcal{R}_{nl} f(h, h) \rightarrow_i^2 \mathcal{R}_{nl} f(\text{zero}, \text{zero}) \rightarrow_i \mathcal{R}_{nl} g(\text{zero}, \text{zero}).$$

All other $\rightarrow_i \mathcal{R}_{nl}$ -sequences that start with basic terms have at most length 4, too. Note that if spareness only required $x\sigma$ to be a normal form for variables x that are duplicated, then this TRS would trivially be spare although $rc_{cp(\mathcal{R}_{nl})} \neq rc_{cp_i(\mathcal{R}_{nl})}$. But with our definition of spareness the TRS is not spare, since the variable x which occurs twice in the right-hand side $g(x, x)$ is instantiated by the redex h in the above reduction.

Our technique relies on the following important result of [124].

Theorem 11.5 (Length of ndg Rewriting [124, Lemma 8]). *If $s \xrightarrow[\text{ndg}]{n} \mathcal{R} t$, then $s \xrightarrow_i^n \mathcal{R} u$ for some term u .*

11.1. NDG REWRITING

Corollary 11.6 follows from Theorem 11.5, because if \mathcal{R} is ndg, then $s \rightarrow_{\mathcal{R}}^n t$ implies $s \xrightarrow[\text{ndg}]{\mathcal{R}}^n t$ for basic terms s .

Corollary 11.6 ($\text{rc}_{cp(\mathcal{R})} = \text{rc}_{cp_i(\mathcal{R})}$). *Let \mathcal{R} be an ordinary TRS which is ndg. Then $\text{rc}_{cp(\mathcal{R})} = \text{rc}_{cp_i(\mathcal{R})}$.*

Thus, we obtain the following processors.

Corollary 11.7 (Strategy Switching). *Let \mathcal{R} be an ordinary TRS which is ndg. Then the processor mapping $cp(\mathcal{R})$ to $cp_i(\mathcal{R})$ is sound for lower and upper bounds. Similarly, the processor mapping $cp_i(\mathcal{R})$ to $cp(\mathcal{R})$ is sound for lower and upper bounds.*

According to Corollary 11.6, innermost and full runtime complexity coincide for ordinary TRSs that are ndg. The following example shows why this result does not carry over to arbitrary TRSs.

Example 11.8. Consider the TRS \mathcal{R} consisting of the rules

$$\alpha_1 = f(x) \xrightarrow{1} f(a) \quad \text{and} \quad \alpha_2 = a \xrightarrow{0} a.$$

It is clearly ndg and its full runtime complexity is unbounded due to the rewrite sequence $f(x) \xrightarrow{1}_{\alpha_1} f(a) \xrightarrow{1}_{\alpha_1} f(a) \xrightarrow{1}_{\alpha_1} \dots$. However, its innermost runtime complexity is constant, as we have $f(x) \xrightarrow{i}_{\alpha_1} f(a) \xrightarrow{i}_{\alpha_2} f(a) \xrightarrow{i}_{\alpha_2} \dots$.

Hence, from now on we restrict ourselves to ordinary TRSs. Unfortunately, the question whether a TRS is spare resp. ndg is undecidable.

Theorem 11.9 (Spareness is Undecidable). *It is undecidable whether a TRS is spare (resp. ndg).*

Proof. Recall that a TRS \mathcal{R} is basic if $\ell, r \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ for all $\ell \rightarrow r \in \mathcal{R}$. The proof relies on an encoding of Turing machines to left-linear basic TRSs where each configuration of the Turing machine is represented by a ground term (i.e., it relies on the Turing completeness of left-linear basic TRSs).

For any Turing machine (cf. Definition 7.14) $\mathcal{M} = (Q, \Gamma, \delta)$, we define the TRS $\mathcal{R}_{\mathcal{M}}$ by adapting the encoding from Section 8.3. In contrast to Section 8.3, we now only consider configurations with finitely many non-blank symbols on the tape. The reason is that we want to reduce the halting problem for Turing machines to spareness of TRSs and the halting problem only considers such configurations. Again, in $\mathcal{R}_{\mathcal{M}}$ there is a function symbol f of arity 4, all symbols from Γ become function symbols of arity 1, and $Q \cup \{\underline{a} \mid a \in \Gamma\}$ are constants.

CHAPTER 11. STRATEGY SWITCHING

Then $\mathcal{R}_{\mathcal{M}}$ is defined as follows:

$$\begin{aligned} \mathcal{R}_{\mathcal{M}} = & \{f(q_1, a_2(xs), \underline{a_1}, ys) \rightarrow f(q_2, xs, \underline{a_2}, b(ys)) \mid a_2 \in \Gamma, \delta(q_1, a_1) = (q_2, b, L)\} \\ & \cup \{f(q_1, xs, \underline{a_1}, a_2(ys)) \rightarrow f(q_2, b(xs), \underline{a_2}, ys) \mid a_2 \in \Gamma, \delta(q_1, a_1) = (q_2, b, R)\} \\ & \cup \{f(q_1, \square, \underline{a_1}, ys) \rightarrow f(q_2, \square, \square, b(ys)) \mid \delta(q_1, a_1) = (q_2, b, L)\} \\ & \cup \{f(q_1, xs, \underline{a_1}, \square) \rightarrow f(q_2, b(xs), \square, \square) \mid \delta(q_1, a_1) = (q_2, b, R)\} \end{aligned}$$

Obviously, $\mathcal{R}_{\mathcal{M}}$ is basic and left-linear. A configuration (q, w, a, w') of the Turing machine can now be encoded as the ground term

$$(q, w, a, w')_{\mathcal{T}} = f(q, w_{\mathcal{T}}, \underline{a}, w'_{\mathcal{T}})$$

where $v_{\mathcal{T}} = \square$ if $v = \square^{\omega}$ (i.e., if v is the infinite word consisting only of \square) and otherwise, $v_{\mathcal{T}} = a(v'_{\mathcal{T}})$ where $v = a.v'$. Now we can prove that $\mathcal{R}_{\mathcal{M}}$ indeed simulates the Turing machine \mathcal{M} . More precisely, we prove that we have

$$\begin{aligned} (q_1, w_1, a_1, w'_1) & \rightarrow_{\mathcal{M}} (q_2, w_2, a_2, w'_2) \\ \text{if and only if} & \\ (q_1, w_1, a_1, w'_1)_{\mathcal{T}} & \rightarrow_{\mathcal{R}_{\mathcal{M}}} (q_2, w_2, a_2, w'_2)_{\mathcal{T}}. \end{aligned} \tag{11.1}$$

In the following, we write $f_1 f_2 \dots f_n c$ for terms of the form $f_1(f_2(\dots f_n(c) \dots))$ to ease readability. For the “only if” direction of (11.1), we just regard the case $\delta(q_1, a_1) = (q_2, b, L)$. The case $\delta(q_1, a_1) = (q_2, b, R)$ is analogous. Hence, $w_1 = a_2.w_2$ and $w'_1 = b.w'_2$. Let $w_2 = b_1.b_2 \dots b_n.\square^{\omega}$ and $w'_1 = c_1.c_2 \dots c_m.\square^{\omega}$. Note that w_2 and w'_1 have to be of this form, as we just consider configuration with finitely many non-blank symbols.

First consider the case $w_1 \neq \square^{\omega}$. Then we have

$$(q_1, w_1, a_1, w'_1)_{\mathcal{T}} = f(q_1, a_2 b_1 b_2 \dots b_n \square, \underline{a_1}, (w'_1)_{\mathcal{T}}).$$

By definition, $f(q_1, a_2(xs), \underline{a_1}, ys) \rightarrow f(q_2, xs, \underline{a_2}, b(ys)) \in \mathcal{R}_{\mathcal{M}}$. Hence, we get

$$\begin{aligned} (q_1, w_1, a_1, w'_1)_{\mathcal{T}} & = f(q_1, a_2 b_1 b_2 \dots b_n \square, \underline{a_1}, (w'_1)_{\mathcal{T}}) \\ & \rightarrow_{\mathcal{R}_{\mathcal{M}}} f(q_2, b_1 b_2 \dots b_n \square, \underline{a_2}, b(w'_1)_{\mathcal{T}}) \\ & = (q_2, w_2, a_2, w'_2)_{\mathcal{T}}. \end{aligned}$$

Now consider the case $w_1 = \square^{\omega}$. Thus,

$$(q_1, w_1, a_1, w'_1)_{\mathcal{T}} = f(q_1, \square, \underline{a_1}, (w'_1)_{\mathcal{T}}).$$

By definition, $f(q_1, \square, \underline{a_1}, ys) \rightarrow f(q_2, \square, \square, b(ys)) \in \mathcal{R}_{\mathcal{M}}$. Note that $w_1 = a_2.w_2$ implies $a_2 = \square$ and $w_2 = \square^{\omega}$. Hence,

$$\begin{aligned} (q_1, w_1, a_1, w'_1)_{\mathcal{T}} & = f(q_1, \square, \underline{a_1}, (w'_1)_{\mathcal{T}}) \\ & \rightarrow_{\mathcal{R}_{\mathcal{M}}} f(q_2, \square, \square, b(w'_1)_{\mathcal{T}}) \\ & = (q_2, w_2, a_2, w'_2)_{\mathcal{T}}. \end{aligned}$$

11.1. NDG REWRITING

For the “if” direction of (11.1), first consider the case that a rule

$$f(q_1, a_2(xs), \underline{a_1}, ys) \rightarrow f(q_2, xs, \underline{a_2}, b(ys))$$

is applied to rewrite $(q_1, w_1, a_1, w'_1)\mathcal{T}$ to $(q_2, w_2, a_2, w'_2)\mathcal{T}$. The case that a rule of the form $f(q_1, xs, \underline{a_1}, a_2(ys)) \rightarrow f(q_2, b(xs), \underline{a_2}, ys)$ is applied is analogous. Then we get $w_1 = a_2.w_2$ and $w'_2 = b.w'_1$. Moreover, we have $\delta(q_1, a_1) = (q_2, b, L)$ by the definition of $\mathcal{R}_{\mathcal{M}}$. Hence, we get

$$\begin{aligned} (q_1, w_1, a_1, w'_1) &= (q_1, a_2.w_2, a_1, w'_1) \\ &\rightarrow_{\mathcal{M}} (q_2, w_2, a_2, b.w'_1) \\ &= (q_2, w_2, a_2, w'_2). \end{aligned}$$

Now consider the case that a rule

$$f(q_1, \square, \underline{a_1}, ys) \rightarrow f(q_2, \square, \square, b(ys))$$

is applied to rewrite $(q_1, w_1, a_1, w'_1)\mathcal{T}$ to $(q_2, w_2, a_2, w'_2)\mathcal{T}$. The case that a rule of the form $f(q_1, xs, \underline{a_1}, \square) \rightarrow f(q_2, b(xs), \square, \square)$ is applied is analogous. Then we get $w_1 = w_2 = \square^\omega$, $a_2 = \square$, and $w'_2 = b.w'_1$. Moreover, $\delta(q_1, a_1) = (q_2, b, L)$ by the definition of $\mathcal{R}_{\mathcal{M}}$. Hence,

$$\begin{aligned} (q_1, w_1, a_1, w'_1) &= (q_1, \square^\omega, a_1, w'_1) \\ &\rightarrow_{\mathcal{M}} (q_2, \square^\omega, \square, b.w'_1) \\ &= (q_2, w_2, a_2, w'_2), \end{aligned}$$

which finishes the proof of (11.1).

By (11.1), undecidability of the halting problem for Turing machines implies undecidability of normalization of basic ground terms w.r.t. left-linear basic TRSs like $\mathcal{R}_{\mathcal{M}}$, as $(q, w, a, w')\mathcal{T}$ is a basic ground term. The reason is that for any Turing machine \mathcal{M} , we have:

$$\begin{aligned} &\mathcal{M} \text{ halts on the start configuration } (q, w, a, w') \\ \iff &\mathcal{R}_{\mathcal{M}} \text{ is terminating on } (q, w, a, w')\mathcal{T} && \text{by (11.1)} \\ \iff &\mathcal{R}_{\mathcal{M}} \text{ is normalizing on } (q, w, a, w')\mathcal{T} \end{aligned}$$

For the last step, note that termination and normalization of $\mathcal{R}_{\mathcal{M}}$ on basic ground terms are equivalent as $\mathcal{R}_{\mathcal{M}}$ is basic and non-overlapping.

Now we can prove that sparseness of TRSs is undecidable. To this end, let \mathcal{R} be a left-linear basic TRS over the signature Σ . As \mathcal{R} is basic, every rewrite sequence that starts with a basic term only leads to basic terms. Hence, \mathcal{R} is sparse.

Given a basic ground term $f(t_1, \dots, t_k) \in \mathcal{T}_{\text{basic}}(\mathcal{R})$, we define a constructor system \mathcal{R}' over the signature $\Sigma' = \Sigma \uplus \{\mathbf{a}, \mathbf{c}, \mathbf{g}, \mathbf{h}, \text{inf}\}$ such that normalization of $f(t_1, \dots, t_k)$ w.r.t. \mathcal{R} can be checked by checking sparseness of \mathcal{R}' instead. Since we have shown that normalization of basic ground terms w.r.t. left-linear basic

CHAPTER 11. STRATEGY SWITCHING

TRSs is undecidable, in this way one can prove that sparseness is also undecidable. As a constructor system is spare if and only if it is ndg by Corollary 11.3 (b), this also shows that it is undecidable whether a TRS is ndg.

The construction of \mathcal{R}' works as follows: All rules of \mathcal{R} are also included in \mathcal{R}' . Moreover, for each defined function symbol $e \in \Sigma_d(\mathcal{R})$, we add rules $e(\dots) \rightarrow a$ to \mathcal{R}' such that for any $p_1, \dots, p_m \in \mathcal{T}(\Sigma_c(\mathcal{R}))$, $e(p_1, \dots, p_m)$ can be reduced to a if and only if $e(p_1, \dots, p_m)$ is in $\rightarrow_{\mathcal{R}}$ -normal form. Note that this is easily possible, as \mathcal{R} is a left-linear constructor system and we only consider basic ground terms $e(p_1, \dots, p_m)$. So the new rules $e(\dots) \rightarrow a$ need to cover all constructor ground terms that are not matched by the left-hand sides of the other e -rules of \mathcal{R} . Furthermore, we add the rules $g \rightarrow h(\text{inf}, f(t_1, \dots, t_k))$, $h(x, a) \rightarrow c(x, x)$, and $\text{inf} \rightarrow \text{inf}$. By construction, \mathcal{R}' is not spare if and only if $f(t_1, \dots, t_k)$ is normalizing w.r.t. \mathcal{R} . To see this, recall that sparseness of \mathcal{R}' means that all rewrite sequences starting with basic terms are spare. Clearly, only rules from \mathcal{R} are applicable to basic terms whose root is from $\Sigma_d(\mathcal{R})$ and thus, all these rewrite sequences are spare. Hence, we now consider all basic terms with root g , h , or inf .

- For the basic term g we have $g \xrightarrow{\text{ndg}}_{\mathcal{R}'} h(\text{inf}, f(t_1, \dots, t_k))$. If $f(t_1, \dots, t_k)$ is not normalizing, then by construction, we can never evaluate h and hence the resulting rewrite sequence is spare. If $f(t_1, \dots, t_k)$ is normalizing, then let t be a normal form of $f(t_1, \dots, t_k)$. Note that as \mathcal{R} is a basic TRS, t is also a basic term. Hence, we get

$$h(\text{inf}, f(t_1, \dots, t_k)) \xrightarrow{\text{ndg}}_{\mathcal{R}'}^* h(\text{inf}, t) \xrightarrow{\text{ndg}}_{\mathcal{R}'} h(\text{inf}, a) \rightarrow_{\mathcal{R}'} c(\text{inf}, \text{inf}).$$

Note that the last step is not spare.

- Each basic term of the form $h(s, s')$ is either a normal form (if $s' \neq a$) or just enables the spare rewrite step $h(s, a) \xrightarrow{\text{ndg}}_{\mathcal{R}'} c(s, s)$.
- The only reduction starting with inf is $\text{inf} \xrightarrow{\text{ndg}}_{\mathcal{R}'} \text{inf} \xrightarrow{\text{ndg}}_{\mathcal{R}'} \dots$

Hence, a semi-decision procedure for sparseness yields a semi-decision procedure for non-normalization of arbitrary basic ground terms for basic left-linear TRSs. \square

On the other hand, the question whether a TRS is *not* spare resp. *not* ndg is semi-decidable. A semi-decision procedure is obtained by enumerating all rewrite sequences starting with basic terms and checking whether these sequences contain non-spare resp. non-ndg steps. In fact, sparseness and ndg are even undecidable for left-linear constructor systems (which correspond to first-order functional programs), as the TRS \mathcal{R}' constructed in the proof of Theorem 11.9 is a left-linear constructor system. However, there are some obvious sufficient syntactic criteria for sparseness.

11.1. NDG REWRITING

Lemma 11.10 (Right-Linear TRSs are Spare). *Every right-linear TRS is spare. Hence, every right-linear overlay system is ndg.*

Proof. Immediate consequence of Definition 11.1 and Corollary 11.3 (b). \square

Lemma 11.11 (TRSs Without Nested Defined Symbols in Right-Hand Sides are ndg). *If there is no rule $\ell \rightarrow r \in \mathcal{R}$ with $\text{root}(r|_\pi), \text{root}(r|_{\pi.\tau}) \in \Sigma_d(\mathcal{R})$ where $\pi, \pi.\tau \in \text{pos}(r)$ and $\tau \neq \epsilon$, then \mathcal{R} is ndg.*

Proof. Let $t_0 \rightarrow_{\mathcal{R}} t_1 \rightarrow_{\mathcal{R}} \dots$ be a rewrite sequence where t_0 is basic. As there is no rule where defined symbols are nested on the right-hand side, defined symbols are not nested in any t_i , $i \in \mathbb{N}$. Hence, $t_0 \rightarrow t_1 \rightarrow_{\mathcal{R}} \dots$ is an innermost and thus ndg rewrite sequence by Corollary 11.3 (a). \square

Thus, the processors from Corollary 11.7 are applicable to right-linear overlay systems as well as TRSs without nested defined symbols in right-hand sides. We will present a much more powerful sufficient criterion for spareness in Section 11.2 which is still easy to automate. For spareness, this criterion subsumes Lemma 11.10 and Lemma 11.11. According to Corollary 11.3 (b), it can be used to prove that overlay systems are ndg. Hence, for checking ndg, the criterion of Section 11.2 subsumes Lemma 11.10.

11.2 Approximating Spareness

According to Corollary 11.6, innermost and full runtime complexity coincide for ndg TRSs. We presented two simple syntactic sufficient criteria which ensure that a TRS is ndg in Lemma 11.10 and Lemma 11.11, but these criteria are still far too restrictive. Hence, we now introduce a much more powerful technique which allows us to prove spareness in many cases. So for overlay systems, due to Corollary 11.3 (b) this technique can be used to prove ndg-ness.

The idea of our technique is to over-approximate all non-innermost redexes which are reachable from basic terms by a finite set of contexts Def where the inner redex is below \square . Similarly, for all rules $\ell \rightarrow r$ with $\#_x(r) > 1$ we over-approximate the redexes $\ell\sigma$ which are reachable from basic terms by a finite set of contexts Dup where \square stands for the “duplicated” subterm $x\sigma$. Then, we check if there are contexts in Def and Dup that “overlap”. If this is not the case, then the analyzed TRS is spare.

To formalize the notions of “overlap” and “over-approximation” we introduce an instance relation on contexts. Then, two contexts overlap if they have a common instance and a context D over-approximates all contexts that are instances of D . The intuition behind the instance relation is that D is “more general” than C if C results from D by instantiating variables and replacing \square by a term containing \square . Then, we can use a context D to represent all terms $C[t]$ where C is an instance of D and t has a certain property (like “may be duplicated” or “may contain redexes”).

Definition 11.12 (Instance \sqsubseteq). Given two contexts $C[\square]_\pi, D[\square]_\tau$ we call C an *instance* of D ($C \sqsubseteq D$) if $\pi \geq \tau$ and $D[x]$ matches C , where x is a fresh variable.

In other words, $C \sqsubseteq D$ holds if and only if there is a context C' and a substitution σ with $C = D\sigma[C']$.

Example 11.13. The context $\text{add}(\text{succ}(\square), \text{succ}(y))$ is an instance of the context $\text{add}(\square, y)$, as we have $1.1 \geq 1$ (where 1.1 and 1 are the positions of \square in $\text{add}(\text{succ}(\square), \text{succ}(y))$ and $\text{add}(\square, y)$, respectively) and $\text{add}(\square, y)[x] = \text{add}(x, y)$ matches $\text{add}(\text{succ}(\square), \text{succ}(y))$.

The following corollary states some useful observations on the instance relation.

Corollary 11.14 (Properties of \sqsubseteq).

- (a) \sqsubseteq is transitive, i.e., $C \sqsubseteq D$ and $D \sqsubseteq E$ implies $C \sqsubseteq E$
- (b) for any context C and any substitution σ we have $C\sigma \sqsubseteq C$
- (c) $\pi, \tau \in \text{pos}(t)$ and $\pi \geq \tau$ implies $t[\square]_\pi \sqsubseteq t[\square]_\tau$ for any term t

11.2. APPROXIMATING SPARENES

Two contexts C and D *overlap* if they have a common instance. In other words, C and D overlap if there exist terms that are represented both by C and by D .

Definition 11.15 (Overlapping Contexts). Given two contexts C, D we say that C and D *overlap* if there is a context E such that $E \sqsubseteq C$ and $E \sqsubseteq D$.

Example 11.16. The contexts $\text{add}(\square, \text{succ}(y))$ and $\text{add}(\text{succ}(\square), y)$ overlap, as $\text{add}(\text{succ}(\square), \text{succ}(y))$ is an instance of both of them.

Note that for any two contexts C and D , it is decidable whether they overlap: One has to check whether the positions of \square in C and D are not independent and whether $C[x]$ and a variable-renamed version of $D[y]$ unify.

A context is *duplicating* if it results from a rule $\ell \rightarrow r$ where a variable x occurs more than once in r and if $\ell\sigma$ appears in a rewrite sequence that starts with a basic term. To turn $\ell\sigma$ into a context, one replaces a subterm of $x\sigma$ by \square .

Definition 11.17 (Duplicating Context). Given a TRS \mathcal{R} , we call a context C *duplicating* if there is a term $s \in \mathcal{T}_{\text{basic}}(\mathcal{R})$, a substitution σ , and a rewrite sequence $s \rightarrow_{\mathcal{R}}^* t \supseteq \ell\sigma$ where ℓ is the left-hand side of a rule $\ell \rightarrow r \in \mathcal{R}$ such that $\ell|_{\pi} = x \in \mathcal{V}$, $\#_x(r) > 1$, and $C = \ell\sigma[\square]_{\pi.\tau}$ for some $\pi.\tau \in \text{pos}(\ell\sigma)$.

Example 11.18. Reconsider the TRS \mathcal{R}_{fib} from Example 7.10. Rule β_4 is the only rule where a variable occurs more than once on the right-hand side. Its left-hand side is $\text{fib}(\text{succ}(\text{succ}(x)))$. If one starts rewriting with a basic term, one can only reach instantiations of the form $\text{fib}(\text{succ}(\text{succ}(t)))$ with $t \in \mathcal{T}(\Sigma_c(\mathcal{R}_{\text{fib}}), \mathcal{V})$. As the variable x of the left-hand side is duplicated, the duplicating contexts of \mathcal{R}_{fib} are $\text{fib}(\text{succ}^n(\square))$ where $t \in \mathcal{T}(\Sigma_c(\mathcal{R}_{\text{fib}}), \mathcal{V})$ and $n \geq 2$. So in other words, sparseness of \mathcal{R}_{fib} can only be violated if a basic term can be rewritten to a term containing an instance of $\text{fib}(\text{succ}^n(\square))$, where \square is replaced by a term that is not a normal form.

As the following theorem shows, it is undecidable if a context is duplicating.

Theorem 11.19. *It is undecidable if a context is duplicating.*

Proof. Let \mathcal{R} be a left-linear basic TRS. We adapt the construction from the proof of Theorem 11.9 by introducing an additional argument for \mathbf{h} and by modifying the \mathbf{g} - and \mathbf{h} -rules to

$$\mathbf{g} \rightarrow \mathbf{h}(\text{inf}, \text{inf}, \mathbf{f}(t_1, \dots, t_k)) \quad \mathbf{h}(x, y, \mathbf{a}) \rightarrow \mathbf{c}(x, x)$$

Then the context $\mathbf{h}(\square, \text{inf}, \mathbf{a})$ is duplicating w.r.t. \mathcal{R}' if and only if $\mathbf{f}(t_1, \dots, t_k)$ is normalizing w.r.t. \mathcal{R} . To see this, we consider all rewrite sequences starting

CHAPTER 11. STRATEGY SWITCHING

with basic terms. Clearly, there is no term q such that $h(\square, \text{inf}, a)[q]$ is reachable from a basic term whose root is from Σ . Hence, we now regard all basic terms with root g , h , or inf .

- For the basic term g we have $g \rightarrow_{\mathcal{R}'} h(\text{inf}, \text{inf}, f(t_1, \dots, t_k))$. If $f(t_1, \dots, t_k)$ is not normalizing, then by construction, the rewrite sequence can never reach a term containing $h(\square, \text{inf}, a)[q]$ for any term q . If $f(t_1, \dots, t_k)$ is normalizing, then let t be a normal form of $f(t_1, \dots, t_k)$. Since \mathcal{R} is a basic TRS, t is again a basic term. Hence, we get

$$h(\text{inf}, \text{inf}, f(t_1, \dots, t_k)) \rightarrow_{\mathcal{R}'}^* h(\text{inf}, \text{inf}, t) \rightarrow_{\mathcal{R}'}^* h(\text{inf}, \text{inf}, a) = h(\square, \text{inf}, a)[\text{inf}],$$

i.e., $h(\square, \text{inf}, a)$ is duplicating.

- Each basic term of the form $h(s, s', s'')$, is either a normal form (if $s'' \neq a$) or just enables the rewrite step $h(s, s', a) \rightarrow_{\mathcal{R}'} c(s, s)$. Hence, $h(\square, s', a)$ is duplicating. However, as $h(s, s', a)$ is basic, we have $s' \neq \text{inf}$ (i.e., this does not mean that $h(\square, \text{inf}, a)$ is duplicating). Since s does not contain defined symbols, $c(s, s)$ is a normal form, i.e., no further terms with root h are reachable from $h(s, s', a)$.
- The only reduction starting with inf is $\text{inf} \rightarrow_{\mathcal{R}'} \text{inf} \rightarrow_{\mathcal{R}'} \dots$

Hence, $h(\square, \text{inf}, a)$ is duplicating if and only if $f(t_1, \dots, t_k)$ is normalizing. Since normalization of basic ground terms w.r.t. left-linear basic TRSs is undecidable (cf. the proof of Theorem 11.9), this implies that it is also undecidable whether a context is duplicating. \square

Hence, whether a context is duplicating is even undecidable for left-linear constructor systems (as \mathcal{R}' in the proof above is such a system). However, the duplicating contexts of a TRS can easily be over-approximated by a finite set of contexts Dup such that every duplicating context is an instance of an element of Dup . In this approximation, we do not take the requirement into account that a duplicating context must be reachable by a rewrite sequence that starts with a basic term. Moreover, we disregard possible instantiations of left-hand sides and only consider contexts where \square is at the position of a duplicated variable.

Definition 11.20 ($\text{Dup}_{\mathcal{R}}$). Given a TRS \mathcal{R} , we define

$$\text{Dup}_{\mathcal{R}} = \{C \mid C[x] \rightarrow r \in \mathcal{R}, \#_x(r) > 1\}.$$

We omit the index \mathcal{R} if it is clear from the context.

11.2. APPROXIMATING SPARENESS

Example 11.21. We have $Dup_{\mathcal{R}_{\text{fib}}} = \{\text{fib}(\text{succ}(\text{succ}(\square)))\}$, as

$$\text{fib}(\text{succ}(\text{succ}(x))) \rightarrow \text{add}(\text{fib}(\text{succ}(x)), \text{fib}(x))$$

is \mathcal{R}_{fib} 's only rule with a non-linear right-hand side and 1.1.1 is the only position of x on the left-hand side. Note that all duplicating contexts $\text{fib}(\text{succ}^n(\square))$, $n \geq 2$, are instances of $\text{fib}(\text{succ}(\text{succ}(\square)))$.

Dup indeed over-approximates all duplicating contexts.

Lemma 11.22 (*Dup Over-Approximates Duplicating Contexts*). *If C is duplicating, then there is a $D \in Dup$ such that $C \sqsubseteq D$.*

Proof. If C is duplicating, then there is a rule $\ell \rightarrow r \in \mathcal{R}$ and a rewrite sequence $s \rightarrow_{\mathcal{R}}^* t \sqsupseteq \ell\sigma = C[p]_{\pi.\tau}$ where $\ell|_{\pi} = x$ is a variable that occurs more than once on the right-hand side. Then we have $\ell[\square]_{\pi} \in Dup$ and $C \sqsubseteq \ell[\square]_{\pi}$. To see this, note that $\ell[x]_{\pi} = \ell$ matches $C[p]_{\pi.\tau} = \ell\sigma$. So if x' is a fresh variable, then $\ell[x']_{\pi}$ also matches $C[\square]_{\pi.\tau} = C$. Moreover, we have $\pi.\tau \geq \pi$. \square

Defined contexts characterize contexts with defined root that can be reached by rewriting basic terms, where a redex may occur at the position of \square .

Definition 11.23 (*Defined Context*). Given a TRS \mathcal{R} , we call a context C *defined* if there is a term $s \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ and a rewrite sequence $s \rightarrow_{\mathcal{R}}^* t \sqsupseteq C[p]$ where $\text{root}(C) \in \Sigma_{\text{d}}(\mathcal{R})$ and p is a redex.

Example 11.24. Reconsider the TRS \mathcal{R}_{fib} . For any $t \in \mathcal{T}(\Sigma_{\text{c}}(\mathcal{R}_{\text{fib}}), \mathcal{V})$, the contexts $\text{add}(\square, \text{fib}(t))$ and $\text{add}(\text{fib}(\text{succ}(t)), \square)$ are defined due to the rewrite sequence $\text{fib}(\text{succ}(\text{succ}(t))) \rightarrow_{\mathcal{R}_{\text{fib}}} \text{add}(\text{fib}(\text{succ}(t)), \text{fib}(t))$.

Theorem 11.25 states that our notions of Definition 11.17 and Definition 11.23 are indeed suitable to determine spareness.

Theorem 11.25 (*No Defined and Duplicating Context \implies Spare*). *If no defined context is duplicating, then \mathcal{R} is spare. If \mathcal{R} is left-linear and spare, then no defined context is duplicating.*

Proof. If \mathcal{R} is not spare, then there is a sequence $s \rightarrow_{\mathcal{R}}^* t \rightarrow_{\ell \rightarrow r, \mu} u$ with $s \in \mathcal{T}_{\text{basic}}(\mathcal{R})$ where all but the last step are spare, i.e., there are positions π, τ such that $t|_{\mu.\pi.\tau}$ is a redex, $\ell|_{\pi} = x \in \mathcal{V}$, and $\#_x(r) > 1$. Thus, $t|_{\mu}[\square]_{\pi.\tau}$ is defined and duplicating.

Now assume that \mathcal{R} is left-linear and there is a context C which is defined and duplicating. Then there is a rewrite sequence $s \rightarrow_{\mathcal{R}}^* t \sqsupseteq C[p]$ where s is basic

CHAPTER 11. STRATEGY SWITCHING

and p is a redex by the definition of defined contexts. Moreover, there is a rule $\ell \rightarrow r$, a variable x with $\#_x(r) > 1$, a substitution σ , and positions π and τ such that $C = \ell\sigma[\Box]_{\pi.\tau}$ and $\ell|_{\pi} = x \in \mathcal{V}$ by the definition of duplicating contexts. As ℓ is linear, this implies $C[p] = \ell\theta$ where θ is the substitution with $x\theta = x\sigma[p]_{\tau}$ and $y\theta = y\sigma$ for all variables $y \neq x$. Hence we get $s \rightarrow_{\mathcal{R}}^* t \supseteq C[p] = \ell\theta \rightarrow_{\ell \rightarrow r} r\theta$ where $x\theta$ is not a normal form, i.e., where the last rewrite step is not spare. Thus, \mathcal{R} is not spare. \square

So while the absence of contexts that are both defined and duplicating always implies sparseness, the following example shows that the converse only holds for left-linear TRSs.

Example 11.26. Consider the TRS \mathcal{R} with the rules

$$f(x, x) \rightarrow g(x, x) \quad b \rightarrow f(c, a) \quad c \rightarrow f(a, a).$$

Since the basic terms b , c , or $f(t, t)$ for $t \in \mathcal{T}(\Sigma_c(\mathcal{R}), \mathcal{V})$ only start rewrite sequences with spare steps, the TRS is spare. However, the context $f(\Box, a)$ is both defined (due to the rewrite sequence $b \rightarrow_{\mathcal{R}} f(c, a)$) and duplicating (due to $c \rightarrow_{\mathcal{R}} f(a, a)$).

Definedness of contexts is undecidable, too.

Theorem 11.27. *It is undecidable if a context is defined.*

Proof. Using the construction from the proof of Theorem 11.9, $h(\Box, a)$ is defined w.r.t. \mathcal{R}' if and only if $f(t_1, \dots, t_k)$ is normalizing w.r.t. \mathcal{R} . \square

Our aim is to use Theorem 11.25 to prove sparseness of TRSs. Thus, we have to show that there is no context that is defined and duplicating. While these properties are both undecidable by Theorem 11.19 and Theorem 11.27, we can approximate duplicating contexts by *Dup* due to Lemma 11.22. Hence, we now have to find a similar over-approximation for defined contexts. Here, a problem is that a defined context may have *several* inner redexes (i.e., redexes can also occur on positions independent of the position of \Box).

Example 11.28. Consider a TRS \mathcal{R} containing the rule $f(x) \rightarrow h(e, g(x))$, where h , e , and g are defined symbols (and thus e is a redex). To compute all defined contexts, we have to consider all terms t with $g(s) \rightarrow_{\mathcal{R}}^* t$ for some $s \in \mathcal{T}(\Sigma_c(\mathcal{R}), \mathcal{V})$, as each of these terms gives rise to a rewrite sequence $f(s) \rightarrow h(e, g(s)) \rightarrow_{\mathcal{R}}^* h(e, t)$ and thus $h(\Box, t)$ is a defined context.

To avoid reasoning about all terms t reachable from instances of some term $g(x)$ as in Example 11.28, we abstract inner defined symbols to variables in order to approximate the set of all defined contexts (e.g., the context $h(\Box, g(x))$ with

11.2. APPROXIMATING SPARENES

the defined symbols h and g is abstracted to $h(\square, x_1)$). However, inner defined symbols above \square are abstracted to \square (e.g., the context $g(g(\square))$ is abstracted to $g(\square)$ and $h(e, g(\square))$ is abstracted to $h(x_1, \square)$). Moreover, we also abstract from variables that occur multiple times in a term. To this end, we replace all occurrences of variables by pairwise different variables. The reason is that equal subterms may be reduced differently, i.e., equality of subterms is not preserved by rewriting. Thus, Definition 11.29 introduces the *abstraction* of a context C , which results from replacing all its topmost proper subterms that are variables or have a defined root (but do not contain \square) by pairwise different variables.

Definition 11.29 (Abstraction of Contexts). Let \mathcal{R} be a TRS. For a context C , let $\tilde{C} = C[\square]_\tau$ where τ is the topmost position of C with $\tau \neq \epsilon$, $C|_\tau \sqsupseteq \square$, and $\text{root}(C|_\tau) \in \Sigma_d(\mathcal{R}) \cup \{\square\}$. Let

$$\Pi_d = \left\{ \pi \mid \text{root}(\tilde{C}|_\pi) \in \Sigma_d(\mathcal{R}), \pi \neq \epsilon \right\} \quad \text{resp.} \quad \Pi_v = \left\{ \pi \mid \tilde{C}|_\pi \in \mathcal{V} \right\}$$

contain all positions of \tilde{C} 's proper subterms with defined root resp. all positions of variables in \tilde{C} . Moreover, let Π consist of the topmost positions of $\Pi_d \cup \Pi_v$, i.e., Π is the smallest subset of $\Pi_d \cup \Pi_v$ such that for each $\tau \in \Pi_d \cup \Pi_v$ there is a $\pi \in \Pi$ with $\pi \leq \tau$. Finally, let π_1, \dots, π_n be Π 's elements in lexicographic order. Then we call $\lfloor C \rfloor = \tilde{C}[x_1]_{\pi_1} \dots [x_n]_{\pi_n}$ the *abstraction* of C , where $x_1, \dots, x_n \in \mathcal{V}$ are pairwise different.

Example 11.30. Recall the rule $f(x) \rightarrow h(e, g(x))$ from Example 11.28. To approximate the defined contexts induced by this rule we first replace one topmost proper subterm of the right-hand side with defined root by \square and then we take the abstraction of the resulting context. In this way, we obtain the contexts $\lfloor h(\square, g(x)) \rfloor = h(\square, x_1)$ and $\lfloor h(e, \square) \rfloor = h(x_1, \square)$.

Lemma 11.31 states that every context C is an instance of its abstraction $\lfloor C \rfloor$. Moreover, if C is an instance of D , then $\lfloor C \rfloor$ is also an instance of D , provided that D is a linear basic context.

Lemma 11.31 (Properties of $\lfloor \cdot \rfloor$). Let \mathcal{R} be a TRS.

- (a) For any context C , we have $C \sqsubseteq \lfloor C \rfloor$.
- (b) For any context C and any linear basic context D , $C \sqsubseteq D$ implies $\lfloor C \rfloor \sqsubseteq D$.
- (c) For any context C , any $\pi \in \text{pos}(C)$ with $C|_\pi \not\sqsupseteq \square$, and any term t with $\text{root}(t) \in \Sigma_d(\mathcal{R})$, we have $\lfloor C \rfloor \sqsubseteq \lfloor C[t]_\pi \rfloor$.

Proof. For (a), we first show $C \sqsubseteq \tilde{C}$. We have $\tilde{C}|_\tau = \square$. The position of \square

CHAPTER 11. STRATEGY SWITCHING

in C is indeed below τ since $C|_\tau \supseteq \square$. Moreover $\tilde{C}[x]_\tau$ matches C for a fresh variable x , since $\tilde{C}[x]_\tau = C[x]_\tau$.

Now we show that $\tilde{C} \sqsubseteq [C]$. For all $\pi \in \Pi_d$, $\tilde{C}|_\pi$ does not contain \square . Thus, \square is at the same position in \tilde{C} and $[C]$. Moreover, by instantiating every x_i by $\tilde{C}|_{\pi_i}$, $[C]$ matches \tilde{C} . Hence, the claim follows from transitivity of \sqsubseteq .

For (b), let $C \sqsubseteq D$. We first show that this implies $\tilde{C} \sqsubseteq D$. Let $C|_\pi = \square$ and $D|_\mu = \square$. Then $C \sqsubseteq D$ implies $\pi \geq \mu$. Moreover, $\tilde{C}|_\tau = \square$ with $\pi \geq \tau$. We also obtain $\tau \geq \mu$, because otherwise we have $\mu > \tau$, but then $D[x]_\mu$ would not match C , since $\text{root}(C|_\tau) \in \Sigma_d(\mathcal{R})$ and $\text{root}(D[x]_\mu|_\tau) = \text{root}(D|_\tau) \notin \Sigma_d(\mathcal{R})$ as D is basic. Let σ be the matcher with $D[x]_\mu \sigma = C$. By defining $x\sigma' = \tilde{C}|_\mu$ and $y\sigma' = y\sigma$ for all variables $y \neq x$, we get $D[x]_\mu \sigma' = D\sigma[\tilde{C}|_\mu]_\mu = C[\tilde{C}|_\mu]_\mu = \tilde{C}$.

To show $[C] \sqsubseteq D$, note again that \square is at the same position τ in \tilde{C} and $[C]$, i.e., for $D|_\mu = \square$ we have $\tau \geq \mu$. Moreover, as $D[x]_\mu$ matches \tilde{C} and D is basic, we must have $D|_{\pi_i} \in \mathcal{V}$ for all $i \in \{1, \dots, n\}$. The variables $D|_{\pi_i}$ are pairwise different, since D is linear. Hence, $D[x]_\mu$ matches $[C]$.

For (c), since $C|_\pi \not\supseteq \square$, the position of \square is the same in $[C]$ and $[C[t]_\pi]$. Moreover up to variable renaming, their only difference is that there could be a position above or equal to π where $[C[t]_\pi]$ has a fresh variable (since $\text{root}(t) \in \Sigma_d(\mathcal{R})$). Hence $[C[t]_\pi]$ matches $[C]$. \square

To approximate the set of all defined contexts, we iteratively compute a set Def such that each defined context is an instance of an element of Def . For every rule $\ell \rightarrow r$ where ℓ is basic, every subterm $C[p]$ of r leads to a defined context $C\sigma$ if $\text{root}(C) \in \Sigma_d(\mathcal{R})$ and $p\sigma$ reduces to a redex. Moreover, given a rule $\ell \rightarrow r$ with $\ell|_\pi = x \in \mathcal{V}$ and a defined context D , consider the case that D overlaps with the context $\ell[\square]_\pi$. So D represents terms which have a redex below the position of \square and $\ell[\square]_\pi$ also represents some of these terms. Then by the application of the rule $\ell \rightarrow r$, the inner defined symbol represented by \square is copied to all occurrences of x in r . If one of these occurrences is below a defined symbol, then we again obtain a defined context.

Example 11.32. Consider the following TRS:

$$\alpha_1 : f(w, x, y, z) \rightarrow g(h) \quad \alpha_2 : g(\text{succ}(x)) \rightarrow f(x, x, x, h) \quad \alpha_3 : h \rightarrow \text{succ}(h)$$

The context $g(\square)$ is defined due to α_1 . By replacing the variable x in the left-hand side of α_2 with \square , we obtain the context $g(\text{succ}(\square))$. As $g(\square)$ and $g(\text{succ}(\square))$ overlap, the defined symbol below \square in $g(\square)$ can be copied to all occurrences of x in the right-hand side of α_2 . Hence, instances of $f(\square, x, x, h)$, $f(x, \square, x, h)$, and $f(x, x, \square, h)$ might be defined. To avoid reasoning about the terms reachable from h , we replace it with a variable. Finally, we abstract from the multiple occurrences of x by replacing them with different variables. Hence, we add $[f(\square, x, x, h)] = f(\square, x_1, x_2, x_3)$, $[f(x, \square, x, h)] = f(x_1, \square, x_2, x_3)$, and $[f(x, x, \square, h)] = f(x_1, x_2, \square, x_3)$ to Def .

11.2. APPROXIMATING SPARENES

In rule α_1 of Example 11.32, we obtained a defined context by replacing the subterm h of the right-hand side with \square . In general, we have to consider all instances of subterms which reduce to a redex. For the sake of simplicity, we over-approximate the set of such subterms by considering all subterms p of right-hand sides with $\text{root}(p) \in \Sigma_d(\mathcal{R})$. Then, we obtain an over-approximation of all defined contexts by iterating the construction illustrated in Example 11.32.

Definition 11.33 ($Def_{\mathcal{R}}$). Given a TRS \mathcal{R} , we define $Def_{\mathcal{R}}$ to be the smallest set such that:

- (a) If $\ell \rightarrow r \in \mathcal{R}$, $r \geq C[p]$, and $\text{root}(C), \text{root}(p) \in \Sigma_d(\mathcal{R})$, then $\lfloor C \rfloor \in Def_{\mathcal{R}}$.
- (b) If $D \in Def_{\mathcal{R}}$, $\ell[x]_{\pi} \rightarrow r \in \mathcal{R}$ with $r \geq C[x]$ and $\text{root}(C) \in \Sigma_d(\mathcal{R})$, and D and $\ell[\square]_{\pi}$ overlap, then $\lfloor C \rfloor \in Def_{\mathcal{R}}$.

We omit the index \mathcal{R} if it is clear from the context.

Lemma 11.34 shows that Def is finite (and hence computable), as it only contains abstractions of contexts that result from replacing subterms of right-hand sides of rules with \square .

Lemma 11.34 (Finiteness of Def). *For each TRS \mathcal{R} , $Def_{\mathcal{R}}$ is finite.*

Proof. We have $Def_{\mathcal{R}} \subseteq \{\lfloor C \rfloor \mid \ell \rightarrow r \in \mathcal{R}, r \geq C[p]\}$, which is finite. \square

Example 11.35. For the TRS \mathcal{R}_{fib} of Example 7.10, we have

$$\lfloor \text{add}(\square, \text{fib}(x)) \rfloor = \text{add}(\square, x_1) \in Def_{\mathcal{R}_{\text{fib}}}$$

by Definition 11.33 (a) due to the right-hand side of Rule β_4 . This context overlaps with the context $\text{add}(\text{succ}(\square), y)$ obtained from the left-hand side of Rule β_1 by replacing the variable x by \square . Since the corresponding right-hand side is $\text{add}(x, \text{succ}(y))$, this implies

$$\lfloor \text{add}(\square, \text{succ}(y)) \rfloor = \text{add}(\square, \text{succ}(x_1)) \in Def_{\mathcal{R}_{\text{fib}}}.$$

Similarly, we get

$$\begin{aligned} \lfloor \text{add}(\text{fib}(\text{succ}(x)), \square) \rfloor &= \text{add}(x_1, \square) \in Def_{\mathcal{R}_{\text{fib}}} \quad \text{and} \\ \lfloor \text{add}(x, \text{succ}(\square)) \rfloor &= \text{add}(x_1, \text{succ}(\square)) \in Def_{\mathcal{R}_{\text{fib}}} \end{aligned}$$

due to β_4 and β_1 . Thus, we have

$$Def_{\mathcal{R}_{\text{fib}}} = \{\text{add}(\square, x_1), \text{add}(\square, \text{succ}(x_1)), \text{add}(x_1, \square), \text{add}(x_1, \text{succ}(\square))\}.$$

CHAPTER 11. STRATEGY SWITCHING

Lemma 11.36 shows that the approximation of Definition 11.33 is indeed correct.

Lemma 11.36 (*Def Over-Approximates Defined Contexts*). *If C is defined, then there is a $D \in Def$ such that $C \sqsubseteq D$.*

Proof. We use induction on n to prove that if there is a rewrite sequence $s \rightarrow_{\mathcal{R}}^n t \sqsupseteq_{\pi} C[p]_{\tau}$ where s is basic, $C|_{\tau} = \square$, and $\text{root}(C), \text{root}(p) \in \Sigma_d(\mathcal{R})$, then there is a $D \in Def$ with $C \sqsubseteq D$.

Case 1. Induction Base ($n = 0$).

As s is basic, s cannot have a subterm $C[p]$ such that $\text{root}(C), \text{root}(p) \in \Sigma_d(\mathcal{R})$. Hence, our claim trivially holds.

Case 2. Induction Step ($n > 0$).

Here, we have $s \rightarrow_{\mathcal{R}}^{n-1} s' \rightarrow_{\ell \rightarrow r, \mu} t \sqsupseteq_{\pi} C[p]_{\tau}$ for some rule $\ell \rightarrow r$.

Case 2.1. μ and π are independent positions, i.e., $\mu \parallel \pi$.

Then we also have $s \rightarrow_{\mathcal{R}}^{n-1} s' \sqsupseteq_{\pi} C[p]_{\tau}$ and hence our claim follows from the induction hypothesis.

Case 2.2. μ is below π , but independent to $\pi.\tau$, i.e., $\mu = \pi.\iota$ and $\iota \parallel \tau$.

We get $s \rightarrow_{\mathcal{R}}^{n-1} s' \sqsupseteq_{\pi} C[\ell\sigma]_{\iota}[p]_{\tau}$. By the induction hypothesis, there is a $D \in Def$ such that $C[\ell\sigma]_{\iota}[\square]_{\tau} \sqsubseteq D$. By construction, each $D \in Def$ is basic and linear. Hence by Lemma 11.31 (b), we get $\lfloor C[\ell\sigma]_{\iota}[\square]_{\tau} \rfloor \sqsubseteq D$. Moreover, we have $C \sqsubseteq \lfloor C \rfloor$ by Lemma 11.31 (a) and

$$\lfloor C \rfloor = \lfloor C[r\sigma]_{\iota}[\square]_{\tau} \rfloor \sqsubseteq \lfloor C[\ell\sigma]_{\iota}[\square]_{\tau} \rfloor$$

by Lemma 11.31 (c), since $\text{root}(\ell\sigma) \in \Sigma_d(\mathcal{R})$. Hence, $C \sqsubseteq D$ follows by transitivity of \sqsubseteq (Corollary 11.14 (a)).

Case 2.3. μ is below $\pi.\tau$ ($\mu \geq \pi.\tau$).

Here, $s \rightarrow_{\mathcal{R}}^{n-1} s' \sqsupseteq_{\pi} C[q]_{\tau}$ with $\text{root}(q) \in \Sigma_d(\mathcal{R})$, as $q = \ell\sigma$ if $\mu = \pi.\tau$ and $\text{root}(q) = \text{root}(p)$ if $\mu > \pi.\tau$. So our claim follows from the induction hypothesis.

Case 2.4. μ is strictly below π , but strictly above $\pi.\tau$ ($\pi.\tau > \mu > \pi$).

Then there is a position ν with $\pi.\nu = \mu$. We get $s \rightarrow_{\mathcal{R}}^{n-1} s' \sqsupseteq_{\pi} C[\ell\sigma]_{\nu}$ where $\text{root}(\ell\sigma) \in \Sigma_d(\mathcal{R})$. The induction hypothesis implies that there is a $D \in Def$ such that $C[\square]_{\nu} \sqsubseteq D$. Clearly, we have $\nu < \tau$ and hence, $C = C[\square]_{\tau} \sqsubseteq C[\square]_{\nu}$ by Corollary 11.14 (c). Hence, $C \sqsubseteq D$ follows from transitivity of \sqsubseteq (Corollary 11.14 (a)).

Case 2.5. π is below μ , i.e., $\mu \leq \pi$.

Then there is a position ν such that $\mu.\nu = \pi$. Thus, we have

$$s \rightarrow_{\mathcal{R}}^{n-1} s' \rightarrow_{\ell \rightarrow r, \mu} s'[r\sigma]_{\mu} \sqsupseteq r\sigma[C[p]_{\tau}]_{\nu}.$$

11.2. APPROXIMATING SPARENES

Case 2.5.1. $\nu.\tau \in \text{pos}(r)$ and $r|_{\nu.\tau} \notin \mathcal{V}$.

Then $\text{root}(C) = \text{root}(r|_{\nu})$ and $\text{root}(p) = \text{root}(r|_{\nu.\tau})$. Hence, we obtain $[r|_{\nu}[\square]_{\tau}] \in \text{Def}$ by Definition 11.33 (a). Moreover, $C \sqsubseteq r|_{\nu}[\square]_{\tau}$ holds as C also has \square at the position τ , and as $r|_{\nu}\sigma = C[p]_{\tau}$ and thus, $r|_{\nu}[x]_{\tau}\sigma' = C$ if $x\sigma' = \square$ and $y\sigma' = y\sigma$ for all variables $y \neq x$. By Lemma 11.31 (a) and transitivity of \sqsubseteq (Corollary 11.14 (a)), we get $C \sqsubseteq [r|_{\nu}[\square]_{\tau}]$.

Case 2.5.2. $\nu \in \text{pos}(r)$, $r|_{\nu} \notin \mathcal{V}$, and $(\nu.\tau \notin \text{pos}(r) \text{ or } r|_{\nu.\tau} \in \mathcal{V})$.

In this case, the root of C is “above” and p is “below” some variable x of r in $r\sigma$, cf. Figure 11.1. So $\tau = \xi.\iota$ such that $\xi \neq \epsilon$, $r|_{\nu.\xi} = x \in \mathcal{V}$, and $x\sigma|_{\iota} = r\sigma|_{\nu.\xi.\iota} = r\sigma|_{\nu.\tau} = p$. As $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$, there is also some $\pi \in \text{pos}(\ell)$ with $\ell|_{\pi} = x$. Note that $\ell\sigma|_{\pi.\iota} = x\sigma|_{\iota} = p$. As $\ell \notin \mathcal{V}$, we have

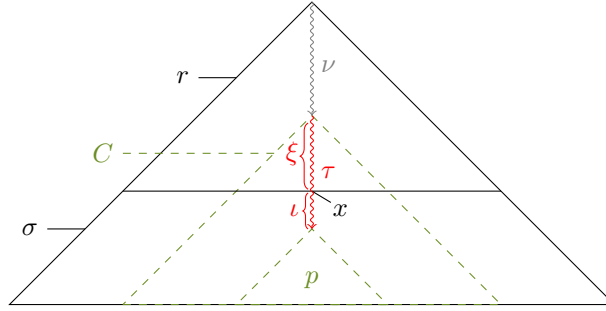


Figure 11.1: Case 5.2

$\pi \neq \epsilon$. Since $\text{root}(\ell), \text{root}(p) \in \Sigma_d(\mathcal{R})$ and $s \rightarrow^{n-1} s' \supseteq \ell\sigma = \ell\sigma[p]_{\pi.\iota}$, there is a $D \in \text{Def}$ such that $\ell\sigma[\square]_{\pi.\iota} \sqsubseteq D$ by the induction hypothesis. Moreover, we have $\text{root}(r|_{\nu}) \in \Sigma_d(\mathcal{R})$ as ν is the position of C in $r\sigma$ and as $r|_{\nu} \notin \mathcal{V}$. We obtain $[r|_{\nu}[\square]_{\xi}] \in \text{Def}$ by Definition 11.33 (b), since the following holds:

- $D \in \text{Def}$
- $\ell \rightarrow r = \ell[x]_{\pi} \rightarrow r \in \mathcal{R}$ with $r = r[x]_{\nu.\xi} \supseteq r|_{\nu}[x]_{\xi}$
- $\text{root}(r|_{\nu}) \in \Sigma_d(\mathcal{R})$
- D and $\ell[\square]_{\pi}$ overlap as $\ell\sigma[\square]_{\pi.\iota} \sqsubseteq D$ and $\ell\sigma[\square]_{\pi.\iota} \sqsubseteq \ell[\square]_{\pi}$; the latter holds due to Corollary 11.14 (b), (c), and (a)

We now prove $C \sqsubseteq r|_{\nu}[\square]_{\xi}$. Then, $C \sqsubseteq [r|_{\nu}[\square]_{\xi}]$ follows by Lemma 11.31 (a) and transitivity of \sqsubseteq (Corollary 11.14 (a)). Note that \square is at position τ in C and at position ξ in $r|_{\nu}[\square]_{\xi}$ with $\xi \leq \tau = \xi.\iota$. Moreover, let x' be

CHAPTER 11. STRATEGY SWITCHING

a fresh variable where $x'\sigma = x\sigma[\square]_\iota$. Then

$$\begin{aligned}
 r|_\nu[x']_\xi\sigma &= r\sigma|_\nu[x'\sigma]_\xi \\
 &= r\sigma|_\nu[x\sigma[\square]_\iota]_\xi \\
 &= (r\sigma[x\sigma]_\nu\xi[\square]_\nu\xi_\iota)|_\nu \\
 &= r\sigma[\square]_\nu\tau|_\nu \\
 &= r\sigma|_\nu[\square]_\tau \\
 &= C[p]_\tau[\square]_\tau \\
 &= C.
 \end{aligned}$$

Case 2.5.3. $\nu \notin \text{pos}(r)$ or $r|_\nu \in \mathcal{V}$.

Then there is an $x \in \mathcal{V}(r)$ with $x\sigma \supseteq C[p]$. As we also have $x \in \mathcal{V}(\ell)$, we obtain $s' \supseteq \ell\sigma \supseteq x\sigma \supseteq C[p]$, i.e., the claim follows by the induction hypothesis. \square

This leads to our main theorem: If the contexts in Def do not overlap with the contexts in Dup , then \mathcal{R} is spare. So if \mathcal{R} is an overlay system then $\text{rc}_{cp(\mathcal{R})}$ and $\text{rc}_{cp_i(\mathcal{R})}$ coincide by Corollary 11.3 (b) and Corollary 11.6.

Theorem 11.37 (Approximating Spareness by Def and Dup). *If there is no $D \in \text{Def}_{\mathcal{R}}$ which overlaps with some $C \in \text{Dup}_{\mathcal{R}}$, then \mathcal{R} is spare.*

Proof. Assume that \mathcal{R} is not spare. By Theorem 11.25, then there is a defined context E that is duplicating. By Lemma 11.22 and Lemma 11.36 there are $C \in \text{Dup}_{\mathcal{R}}$ and $D \in \text{Def}_{\mathcal{R}}$ with $E \sqsubseteq C$ and $E \sqsubseteq D$. Hence, C and D overlap which contradicts the prerequisite of the theorem. \square

The criterion of Theorem 11.37 can easily be automated since $\text{Def}_{\mathcal{R}}$ and $\text{Dup}_{\mathcal{R}}$ are computable finite sets of contexts and for any two contexts, it is decidable whether they overlap.

Example 11.38. We have

$$\begin{aligned}
 \text{Dup}_{\mathcal{R}_{\text{fib}}} &= \{\text{fib}(\text{succ}(\text{succ}(\square)))\} && \text{and} \\
 \text{Def}_{\mathcal{R}_{\text{fib}}} &= \{\text{add}(\square, x_1), \text{add}(\square, \text{succ}(x_1)), \text{add}(x_1, \square), \text{add}(x_1, \text{succ}(\square))\},
 \end{aligned}$$

cf. Example 11.21 and Example 11.35. Clearly, $\text{Dup}_{\mathcal{R}_{\text{fib}}}$ and $\text{Def}_{\mathcal{R}_{\text{fib}}}$ do not overlap. As \mathcal{R}_{fib} is an overlay system, this implies $\text{rc}_{cp(\mathcal{R}_{\text{fib}})} = \text{rc}_{cp_i(\mathcal{R}_{\text{fib}})}$.

Note that for spareness, Theorem 11.37 clearly subsumes Lemma 11.10 and Lemma 11.11. Lemma 11.10 is subsumed as $\text{Dup}_{\mathcal{R}} = \emptyset$ if \mathcal{R} is right-linear. Lemma 11.11 is subsumed since $\text{Def}_{\mathcal{R}} = \emptyset$ if \mathcal{R} has no rules where defined symbols are nested on right-hand sides. Thus, in both cases Theorem 11.37 implies spareness.

11.3 Related Work

Sufficient criteria such that full and innermost termination coincide are presented in [70]. The least restrictive criterion in [70] requires the TRS to be a locally confluent overlay system. Our technique is also particularly well suited for overlay systems, but in [53] we show how one can also handle non-overlay systems in combination with a suitable preprocessing. Moreover, instead of local confluence we require that one may only use instantiations which keep certain subterms of the rules in normal form. So compared to [70], both the properties of interest (termination vs. complexity) as well as the identified sufficient criteria are very different. Example 11.39 shows that the conditions required by [70] are not sufficient to ensure $\text{rc}_{cp}(\mathcal{R}) = \text{rc}_{cp_i}(\mathcal{R})$.

Example 11.39. Consider the following TRS \mathcal{R} :

$$\begin{array}{ll} \text{f}(\text{zero}, y) & \rightarrow y & \text{g}(x) & \rightarrow \text{f}(x, \text{a}) \\ \text{f}(\text{succ}(x), y) & \rightarrow \text{f}(x, \text{node}(y, y)) & \text{a} & \rightarrow \text{b} \end{array}$$

\mathcal{R} is non-overlapping and thus a locally confluent overlay system. Hence, termination and innermost termination of \mathcal{R} coincide by [70]. Any basic term of size n only leads to innermost rewrite sequences of length $\mathcal{O}(n)$. In contrast, arbitrary rewrite sequences can have exponential length. For example, the basic term $\text{g}(\text{succ}^n(\text{zero}))$ of size $n + 2$ is first reduced to $\text{f}(\text{succ}^n(\text{zero}), \text{a})$. Instead of evaluating the subterm a , one could now apply the second f -rule repeatedly and obtain a complete binary tree of height n whose (exponentially many) leaves are a 's. Finally, these leaves can all be reduced to b in 2^n rewrite steps. Thus, the innermost runtime complexity of \mathcal{R} is linear, whereas its full runtime complexity is exponential.

In [111], the authors identify criteria which ensure that all normal forms of a term w.r.t. full rewriting are also reachable via innermost rewriting. This turns out to be the case for right-linear terminating overlay systems. As mentioned before, our technique is also particularly well suited for overlay systems, but we neither require termination nor right-linearity. In fact, non-right-linear rules are common in many TRSs like \mathcal{R}_{fib} from Example 7.10 which implement natural algorithms. The following example illustrates that the property that every normal form is reachable via innermost rewriting is not sufficient for $\text{rc}_{cp}(\mathcal{R}) = \text{rc}_{cp_i}(\mathcal{R})$.

Example 11.40. Consider the TRS \mathcal{R} with the rules $\text{c} \rightarrow \text{f}(\text{a})$, $\text{f}(\text{a}) \rightarrow \text{f}(\text{a})$, and $\text{a} \rightarrow \text{b}$. Clearly, all normal forms are reachable via innermost rewriting as the only possible non-innermost rewrite steps have the form $\text{f}^n(\text{a}) \rightarrow \text{f}^n(\text{a})$. However, we have $\text{rc}_{cp_i}(\mathcal{R})(n) \in \Theta(1)$ but $\text{rc}_{cp}(\mathcal{R})(n) \in \Theta(\omega)$ due to the non-terminating rewrite sequence $\text{c} \rightarrow \text{f}(\text{a}) \rightarrow \text{f}(\text{a}) \rightarrow \dots$ that starts with the basic term c .

CHAPTER 11. STRATEGY SWITCHING

However, $\text{rc}_{cp}(\mathcal{R}) = \text{rc}_{cp_i}(\mathcal{R})$ indeed holds for right-linear overlay systems, which is a special case of the criterion introduced in Section 11.2.

In [77], it is shown that for non-duplicating overlay systems, whenever a term t has a reduction to a normal form then t also starts an innermost reduction of the same length. Thus, this implies $\text{rc}_{cp}(\mathcal{R}) = \text{rc}_{cp_i}(\mathcal{R})$ for non-duplicating terminating overlay systems, which can be used to improve automated complexity analysis [15]. In contrast, our approach does not require termination and it allows us to infer $\text{rc}_{cp}(\mathcal{R}) = \text{rc}_{cp_i}(\mathcal{R})$ for many TRSs that are duplicating.

The paper [106] introduces general methodologies to compare the efficiency of rewrite strategies and hence generalizes the results from [124] that are the foundation of our technique. Thus, these methodologies may serve as a starting point for a generalization of the presented technique. To this end, one has to investigate to which extent the ideas from [106] can be automated.

Finally, there is plenty of research regarding reachability analysis for term rewrite systems via tree automata techniques, e.g., [47, 60, 61]. In principle, such techniques could also be used to approximate sparseness (and hence ndg-ness) of a TRS \mathcal{R} . To this end, one could define a language \mathcal{L} containing all instances of left-hand sides where a variable which occurs more than once on the right-hand side is instantiated with a non-normal form. If reachability tools like Timbuk [61] can then prove that \mathcal{L} is not reachable from $\mathcal{T}_{\text{basic}}(\mathcal{R})$, then this implies sparseness. However, as reachability is undecidable, the corresponding techniques are necessarily incomplete and potentially computationally expensive. In contrast to such general purpose techniques, our approximation from Section 11.2 is lightweight, easy to implement, and specifically designed for our use case. Thus, a combination of the ideas presented in this chapter and general purpose reachability analysis techniques may result in an improvement in terms of precision, but it might also result in regressions w.r.t. performance.

11.4 Conclusion and Future Work

We presented a sufficient criterion to prove that a TRS is *sparse*, i.e., that rewrite sequences starting with basic terms can never duplicate redexes. For overlay systems, sparseness is equivalent to *ndg-ness*. A term rewrite system is ndg if every rewrite sequence starting with basic terms complies with a rewrite strategy called *non-dup generalized innermost rewriting* [124].

The crucial property of this rewrite strategy is that it is at most as expensive as innermost rewriting. Thus, by proving that all rewrite sequences starting with basic terms are ndg, one also proves that innermost rewriting is the “worst” (i.e., the most expensive) of all possible evaluation strategies. Hence, in such cases full and innermost runtime complexity coincide, which permits *strategy switching*, i.e., it allows us to analyze innermost instead of full runtime complexity or vice versa. In this way, we can use all existing and future techniques for innermost (resp. full) runtime complexity to also analyze full (resp. innermost) runtime complexity.

Our sufficient criterion for sparseness computes sets of contexts *Dup* and *Def*, which represent families of terms via an *instance relation*. Thereby, the occurrence of “ \square ” represents a subterm that may be duplicated (in the case of *Dup*) or that may contain nested redexes (in the case of *Def*) in rewrite sequences starting with basic terms. Thus, if *Def* and *Dup* do not overlap (i.e., do not have common instances), then redexes may never be duplicated in rewrite sequences starting with basic terms, i.e., then the TRS is sparse.

The presented criterion is easy to automate and implemented in the tool AProVE. Furthermore, in combination with the technique to remove rules with non-basic left-hand sides from [53, Section 5], it can also be used to prove ndg-ness of non-overlay systems. However, non-overlay systems are irrelevant in the context of program verification and thus beyond the scope of this thesis. See Chapter 12 for a detailed evaluation of our strategy switching technique.

In future work, one should investigate alternative approximations of sparseness, for example based on tree automata techniques [47, 60, 61]. Moreover, ideas from [106] may lead to even more general criteria for $rc_{cp}(\mathcal{R}) = rc_{cp_i}(\mathcal{R})$. Hence, complexity analysis techniques for full runtime complexity might benefit from the automation of ideas from [106].

Experiments

In this chapter, we evaluate the power of the techniques presented in Part III regarding full (Section 12.1) and innermost (Section 12.2) rewriting. Therefore, we use 1022 examples from the category “Runtime Complexity – Innermost Rewriting” and 899 examples from the category “Runtime Complexity – Full Rewriting” of the *Termination Problems Data Base (TPDB) 10.4* [122], the collection of examples which was used at the *Termination and Complexity Competition 2016* [121]. Note that, for these categories, the TPDB 10.5 which was used at the *Termination and Complexity Competition 2017* is a subset of the TPDB 10.4, as all non-left-linear and all non-constructor systems were removed from the category “Runtime Complexity – Innermost Rewriting” prior to the *Termination and Complexity Competition 2017*. In the category “Runtime Complexity – Full Rewriting”, we disregard 60 non-standard TRSs with extra variables on right-hand sides of rules.

In all benchmarks, we preprocessed all TRSs with AProVE’s implementation of the technique from [53, Section 5] to remove rules which are not reachable from basic terms. For all benchmarks with AProVE, we used a timeout of 60 seconds (including the aforementioned preprocessing). Besides AProVE, we also tested with TCT [17]. Here, we also preprocessed the TRSs with AProVE as mentioned above. For TCT, we also used a timeout of 60 seconds, excluding the time that AProVE needed for its preprocessings.

All tools that were used for the evaluation, all strategies that were used to configure AProVE, and detailed information on the invocation of the tools are available at [52].

12.1 Full Rewriting

We first consider full rewriting, where we analyzed the examples from the TPDB with AProVE's implementation of the following techniques for the inference of lower bounds:

- loop detection (Chapter 8)
 - standalone
 - with argument filtering (Section 9.7)
- induction technique (Chapter 9)
 - standalone
 - with indefinite lemmas (Section 9.6)
 - with argument filtering (Section 9.7)
 - with indefinite lemmas and argument filtering

Recall that the techniques for the inference of lower bounds presented in this thesis are the first of their kind, i.e., we cannot compare our results with any other techniques for lower bounds.¹ Moreover, we used the following configurations to infer upper bounds with AProVE:

- semi-decision procedure for constant bounds (Chapter 10)
- Matchbounds, an automata-based technique for the inference of linear upper bounds [126, 127]
- strategy switching in combination with
 - the semi-decision procedure for constant bounds
 - the transformation from TRSs to RNTSs described in [103] together with the technique from Chapter 5
 - the transformation from TRSs to RNTSs described in [103] together with CoFloCo
 - Dependency Tuples [105], an adaption of the Dependency Pair Framework [66] for complexity analysis

This covers all techniques for the inference of upper bounds that are implemented in AProVE. Note that AProVE's implementation of Matchbounds does not take the rewrite strategy into account, i.e., combining Matchbounds with strategy switching does not have any effect. Finally, we also used TCT [17] with and without strategy switching for the inference of upper bounds.

¹Note that TCT can also infer lower bounds for TRSs. To this end, it uses the loop detection technique from Chapter 8, which is also implemented in AProVE. Thus, taking TCT's lower bounds into account would not add any value to our evaluation.

12.1. FULL REWRITING

As witnessed by the annual *Termination and Complexity Competition* [121], AProVE and TCT have been the leading tools for complexity analysis of term rewriting since many years. Thus, the configurations mentioned above represent the current state of the art.

Loop Detection To evaluate the power of loop detection (standalone), Table 12.1 compares its results with the smallest upper bounds proven by all the configurations named above. Loop detection infers a non-trivial lower bound in all but 10 cases where no constant upper bound can be proven. Out of these 10 failures, 5 are caused by timeouts. From the remaining 5 examples, 2 are not left-linear and one example is not a constructor system. Note that while we are not aware of an ordinary left-linear constructor system where loop detection fails, it is easy to construct a non-left-linear TRS or a non-constructor system with at least linear complexity, but without a decreasing loop. From the remaining two examples, one has a decreasing loop of length 13 and our implementation stops narrowing too early to find it. The last example has constant runtime complexity, but the semi-decision procedure from Chapter 10 times out.

Loop detection infers an exponential bound in 143 cases and it proves that the runtime complexity is unbounded in 90 cases. The average runtime was 4.2 seconds per successfully analyzed example.

To see if the argument filtering technique from Section 9.7 has an impact on the power of loop detection, we compare the results of loop detection with and without argument filtering in Table 12.2. The results with argument filtering are strictly worse than without argument filtering, i.e., loop detection should not be combined with the technique from Section 9.7.

Induction Technique Table 12.3 compares the results of the induction technique with argument filtering and indefinite lemmas with the smallest upper bounds proven by any configuration. The induction technique fails in 170 cases where no constant upper bound is proven, i.e., significantly more often than loop detection. However, the main advantage of the induction technique in comparison with loop detection is its ability to infer super-linear polynomial bounds, which happens in 87 cases. The average runtime was 6.7 seconds per successfully analyzed example.

Table 12.4 compares the induction technique (standalone) with the combination of argument filtering and the induction technique. Thus, it illustrates how much the induction technique benefits from the argument filtering technique presented in Section 9.7. With argument filtering, the induction technique infers 8 additional bounds, two of which are exponential. Remarkably, loop detection fails to infer these two exponential bounds, as they are caused by nested recursion. In other words, in both cases the recursive calls that cause exponential costs are not at independent positions and hence they do not give rise to compatible decreasing loops. In such cases, the induction technique is

CHAPTER 12. EXPERIMENTS

| | | Loop Detection | | | | | | | | |
|------------------|-----------------------|----------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 56 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 4 | 232 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | — | 38 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | — | 15 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | 2 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | — | 2 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | — | — | — | — | — | — | 3 | — |
| | $\mathcal{O}(\omega)$ | 6 | 310 | — | — | — | — | — | 140 | 90 |

Table 12.1: Best Upper Bound vs. Loop Detection

| | | Loop Detection | | | | |
|---------------|------------------|----------------|-------------|-------|------------------|----|
| L.D. Filtered | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | EXP | $\Omega(\omega)$ | |
| | $\Omega(1)$ | 66 | — | — | — | — |
| | $\Omega(n)$ | — | 600 | 16 | — | — |
| | EXP | — | — | 127 | — | — |
| | $\Omega(\omega)$ | — | — | — | — | 90 |
| | | | | | | |

Table 12.2: Loop Detection Filtered vs. Loop Detection

| | | Induction Technique | | | | | | | | |
|------------------|-----------------------|---------------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 56 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 32 | 204 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | 3 | 28 | 7 | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | — | 5 | 3 | 7 | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | — | — | 2 | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | 1 | 1 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | 3 | — | — | — | — | — | — | — |
| | $\mathcal{O}(\omega)$ | 134 | 331 | 60 | 7 | 1 | — | — | 13 | — |

Table 12.3: Best Upper Bound vs. Induction Technique

| | | Induction Technique Standalone | | | | | |
|---------------|---------------|--------------------------------|-------------|---------------|---------------|---------------|-------|
| I.D. Filtered | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^5)$ | EXP |
| | $\Omega(1)$ | 459 | 1 | — | — | — | — |
| | $\Omega(n)$ | 5 | 351 | — | — | — | — |
| | $\Omega(n^2)$ | 1 | — | 62 | — | — | — |
| | $\Omega(n^3)$ | — | — | — | 14 | — | — |
| | $\Omega(n^5)$ | — | — | — | — | 1 | — |
| | EXP | 2 | — | — | — | — | 3 |

Table 12.4: Induction Technique Filtered vs. Induction Technique Standalone

12.1. FULL REWRITING

superior to loop detection.

Finally, Table 12.5 compares the induction technique (standalone) with the variant of the induction technique which also infers indefinite lemmas (cf. Section 9.6). Clearly, indefinite lemmas substantially increase the power of the induction technique. With indefinite lemmas, AProVE infers better bounds in 252 cases.

Loop Detection vs. Induction Technique The comparison of loop detection with the induction technique (with argument filtering and indefinite lemmas) is shown in Table 12.6. It clearly shows the orthogonality of both techniques. Hence, recent versions of AProVE use both techniques by default.

To conclude the evaluation of the presented techniques for the inference of lower bounds for full rewriting, Table 12.7 compares the best lower bounds inferred by all considered configurations with the respective upper bounds. Thus, this comparison represents the current state of the art, including the techniques presented in this thesis. The fact that there are no entries above the diagonal means that there are no conflicting bounds, i.e., there is no case where the presented techniques yield a lower bound which is greater than the respective upper bound. In 252 of the 296 cases (85%) where a non-trivial lower as well as a non-trivial upper bound is proven, the bounds are tight, i.e., the lower and the upper bound coincide. Note that the number of tight bounds would increase significantly if state-of-the-art complexity analysis tools had better support for exponential upper bounds.

Constant Bounds We now evaluate the presented semi-decision procedure for the inference of constant bounds. To this end, Table 12.8 compares its results with the smallest upper bounds inferred by all other considered configurations. The technique from Chapter 10 can infer 5 constant bounds which cannot be inferred by any other configuration. Moreover, whenever any other configuration infers a constant bound, then the technique from Chapter 10 succeeds, too. The average runtime per successfully analyzed example was 1.4 seconds. Since our semi-decision procedure subsumes all other techniques w.r.t. constant bounds, we can see from Table 12.7 that there are only 6 cases where the technique from Chapter 10 fails and we also fail to infer a non-trivial lower bound (i.e., to disprove constant complexity).

Strategy Switching To evaluate the effect of the strategy switching technique from Chapter 11, we compare the state of the art with and without strategy switching. To this end, Table 12.9 compares the best upper bounds proven by any considered configuration with the results of those configurations that can also infer upper bounds for full rewriting without strategy switching: TCT, Matchbounds, and the semi-decision procedure for upper bounds from Chapter 10. With strategy switching, we can infer upper bounds for 60 TRSs where all other techniques fail. This means that we can prove 353 upper bounds with strategy switching, which is a significant improvement in comparison to

CHAPTER 12. EXPERIMENTS

| Induction Technique Standalone | | | | | | | | |
|--------------------------------|---------------|-------------|-------------|---------------|---------------|---------------|---------------|-------|
| I.T.+Indefinite Lem. | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | EXP |
| | $\Omega(1)$ | 231 | — | — | — | — | — | — |
| | $\Omega(n)$ | 230 | 340 | — | — | — | 1 | — |
| | $\Omega(n^2)$ | — | 10 | 59 | — | — | — | — |
| | $\Omega(n^3)$ | — | — | 3 | 13 | — | — | — |
| | $\Omega(n^4)$ | — | — | — | 1 | — | — | — |
| | $\Omega(n^5)$ | — | — | — | — | — | — | — |
| | EXP | 6 | 2 | — | — | — | — | 3 |

Table 12.5: Induction Technique with Indefinite Lemmas vs. Induction Technique Standalone

| Induction Technique | | | | | | | | |
|---------------------|------------------|-------------|-------------|---------------|---------------|---------------|-------|------------------|
| Loop Detection | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | EXP | $\Omega(\omega)$ |
| | $\Omega(1)$ | 62 | 4 | — | — | — | — | — |
| | $\Omega(n)$ | 69 | 462 | 53 | 13 | 1 | 2 | — |
| | $\Omega(n^2)$ | — | — | — | — | — | — | — |
| | $\Omega(n^3)$ | — | — | — | — | — | — | — |
| | $\Omega(n^4)$ | — | — | — | — | — | — | — |
| | EXP | 42 | 76 | 13 | 1 | — | 11 | — |
| | $\Omega(\omega)$ | 53 | 31 | 4 | 2 | — | — | — |

Table 12.6: Loop Detection vs. Induction Technique

| Best Lower Bound | | | | | | | | | | |
|------------------|-----------------------|-------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 56 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 1 | 235 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | — | 31 | 7 | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | — | 5 | 3 | 7 | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | — | — | 2 | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | — | 2 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | — | — | — | — | — | — | 3 | — |
| | $\mathcal{O}(\omega)$ | 5 | 261 | 43 | 4 | 1 | — | — | 142 | 90 |

Table 12.7: Best Upper Bound vs. Best Lower Bound

| Constant Bounds | | | | | | | | | | |
|--------------------|-----------------------|------------------|------------------|--------------------|--------------------|--------------------|--------------------|-----------------------|-------|-----------------------|
| Other Upper Bounds | $rc(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n^5)$ | $\mathcal{O}(n^{10})$ | EXP | $\mathcal{O}(\omega)$ |
| | $\mathcal{O}(1)$ | 51 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 4 | — | — | — | — | — | — | — | 236 |
| | $\mathcal{O}(n^2)$ | — | — | — | — | — | — | — | — | 38 |
| | $\mathcal{O}(n^3)$ | — | — | — | — | — | — | — | — | 15 |
| | $\mathcal{O}(n^4)$ | — | — | — | — | — | — | — | — | 2 |
| | $\mathcal{O}(n^5)$ | — | — | — | — | — | — | — | — | 2 |
| | $\mathcal{O}(n^{10})$ | — | — | — | — | — | — | — | — | 1 |
| | EXP | — | — | — | — | — | — | — | — | 3 |
| | $\mathcal{O}(\omega)$ | 1 | — | — | — | — | — | — | — | 546 |

Table 12.8: Other Upper Bounds vs. Constant Bounds

12.1. FULL REWRITING

the 293 bounds that can be proven without strategy switching. Moreover, the precision improves in 8 cases.

| No Strategy Switching | Best Upper Bound | | | | | | | | | |
|-----------------------|-----------------------|------------------|------------------|--------------------|--------------------|--------------------|--------------------|-----------------------|-------|-----------------------|
| | $rc(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n^5)$ | $\mathcal{O}(n^{10})$ | EXP | $\mathcal{O}(\omega)$ |
| | $\mathcal{O}(1)$ | 56 | – | – | – | – | – | – | – | – |
| | $\mathcal{O}(n)$ | – | 210 | – | – | – | – | – | – | – |
| | $\mathcal{O}(n^2)$ | – | 7 | 14 | – | – | – | – | – | – |
| | $\mathcal{O}(n^3)$ | – | 1 | – | 3 | – | – | – | – | – |
| | $\mathcal{O}(n^4)$ | – | – | – | – | – | – | – | – | – |
| | $\mathcal{O}(n^5)$ | – | – | – | – | – | 1 | – | – | – |
| | $\mathcal{O}(n^{10})$ | – | – | – | – | – | – | 1 | – | – |
| | EXP | – | – | – | – | – | – | – | – | – |
| | $\mathcal{O}(\omega)$ | – | 18 | 24 | 12 | 2 | 1 | – | 3 | 546 |

Table 12.9: No Strategy Switching vs. Best Upper Bound

12.2 Innermost Rewriting

Now we consider innermost rewriting, where we analyzed the examples from the TPDB with AProVE's implementation of the following techniques for the inference of lower bounds:

- loop detection (Chapter 8)
 - standalone
 - with argument filtering (Section 9.7)
 - with strategy switching (Chapter 11)
- induction technique (Chapter 9)
 - standalone
 - with indefinite lemmas (Section 9.6)
 - with argument filtering (Section 9.7)
 - with indefinite lemmas and argument filtering
 - with indefinite lemmas, argument filtering, and strategy switching

Moreover, we used the following configurations to infer upper bounds with AProVE:

- semi-decision procedure for constant bounds (Chapter 10)
- Matchbounds [126, 127]
- the transformation from TRSs to RNTSs described in [103] together with the technique from Chapter 5
- the transformation from TRSs to RNTSs described in [103] together with CoFloCo
- Dependency Tuples [105]

Again, this covers all techniques for the inference of upper bounds on the innermost runtime complexity of term rewriting implemented in AProVE. Finally, we also used TCT [17] to infer upper bounds.

As in the case of full rewriting, these configurations represent the current state of the art, as AProVE and TCT have been the leading complexity analysis tools for term rewriting since many years [121].

12.2. INNERMOST REWRITING

Innermost Loop Detection To evaluate the power of innermost loop detection (standalone), Table 12.10 compares its results with the smallest upper bounds proven by all the configurations named above. Innermost loop detection infers a non-trivial lower bound in all but 9 cases where no constant upper bound can be proven. As for full rewriting, the reasons for failure include time-outs (3 cases), non-left-linearity (2 cases), and very long decreasing loops (1 case). From the remaining 3 examples where innermost loop detection fails, 2 are non-ordinary (i.e., they contain rules with cost 0). Again, we are not aware of an ordinary left-linear constructor system where loop detection fails, but it is easy to construct a non-ordinary TRS with at least linear complexity, but without a decreasing loop. The last example where innermost loop detection fails has a decreasing loop which cannot be obtained by narrowing the rules of the TRS. Instead, one would have to instantiate one of its rules. Thus, our narrowing-based heuristic fails to detect this loop.

Innermost loop detection infers an exponential bound in 41 cases and it proves that the runtime complexity is unbounded in 100 cases. The average runtime was 4.1 seconds per successfully analyzed example.

To see if the argument filtering technique from Section 9.7 has an impact on the power of innermost loop detection, we compare the results of innermost loop detection with and without argument filtering in Table 12.11 (see [54] for an adaption of our argument filtering technique for innermost rewriting). However, the results are equal in both settings, i.e., innermost loop detection does not benefit from the technique presented in Section 9.7.

Finally, Table 12.12 compares the power of innermost loop detection with the combination of strategy switching and innermost loop detection. The results are orthogonal, but the differences are marginal. In particular, the combination of strategy switching and innermost loop detection cannot prove a non-trivial lower bound for any of the examples where innermost loop detection fails.

Innermost Induction Technique Table 12.13 compares the results of the innermost induction technique with argument filtering and indefinite lemmas with the smallest upper bounds proven by any configuration (see [54] for an adaption of indefinite lemmas for innermost rewriting). As in the case of full rewriting, the innermost induction technique fails significantly more often than innermost loop detection, but demonstrates its strength by inferring 96 super-linear polynomial bounds. The average runtime was 7.5 seconds per successfully analyzed example.

Table 12.14 compares the innermost induction technique (standalone) with the combination of argument filtering and the innermost induction technique. With argument filtering, the innermost induction technique infers 16 additional bounds. As in the case of full rewriting, two of them are exponential and cannot be inferred by innermost loop detection, as they are caused by nested recursion. To illustrate the effect of indefinite lemmas, Table 12.15 compares the innermost induction technique (standalone) with the variant of the innermost induc-

CHAPTER 12. EXPERIMENTS

| Innermost Loop Detection | | | | | | | | | | |
|--------------------------|-----------------------|-------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best I. Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 58 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 6 | 340 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | — | 125 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | — | 36 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | 5 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | — | 3 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | 1 | — | — | — | — | — | 4 | — |
| | $\mathcal{O}(\omega)$ | 3 | 303 | — | — | — | — | — | 37 | 100 |

Table 12.10: Best Innermost Upper Bound vs. Innermost Loop Detection

| Innermost Loop Detection | | | | | |
|--------------------------|------------------|-------------|-------------|-------|------------------|
| I. L.D. Filt. | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | EXP | $\Omega(\omega)$ |
| | $\Omega(1)$ | 67 | — | — | — |
| | $\Omega(n)$ | — | 814 | — | — |
| | EXP | — | — | 41 | — |
| | $\Omega(\omega)$ | — | — | — | 100 |

Table 12.11: Innermost Loop Detection Filtered vs. Innermost Loop Detection

| Innermost Loop Detection | | | | | |
|--------------------------|------------------|-------------|-------------|-------|------------------|
| S.S. & I. L.D. | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | EXP | $\Omega(\omega)$ |
| | $\Omega(1)$ | 67 | 2 | — | — |
| | $\Omega(n)$ | — | 809 | — | 2 |
| | EXP | — | — | 41 | — |
| | $\Omega(\omega)$ | — | 3 | — | 98 |

Table 12.12: Innermost Loop Detection with Strategy Switching vs. Innermost Loop Detection

| Innermost Induction Technique | | | | | | | | | | |
|-------------------------------|-----------------------|-------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best I. Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 58 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 53 | 293 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | 6 | 100 | 19 | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | 1 | 18 | 6 | 11 | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | 2 | — | 3 | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | 1 | 2 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | 4 | — | — | — | — | — | 1 | — |
| | $\mathcal{O}(\omega)$ | 132 | 242 | 50 | 6 | 1 | — | — | 12 | — |

Table 12.13: Best Innermost Upper Bound vs. Innermost Induction Technique

12.2. INNERMOST REWRITING

tion technique which also infers indefinite lemmas. Clearly, indefinite lemmas substantially increase the power of the innermost induction technique. With indefinite lemmas, AProVE infers better bounds in 262 cases.

Finally, Table 12.16 compares the innermost induction technique with and without strategy switching. The results with strategy switching are strictly worse than without strategy switching due to one additional timeout. Thus, neither innermost loop detection nor the innermost induction technique significantly benefit from strategy switching, i.e., strategy switching is mainly of interest for the inference of upper bounds.

Innermost Loop Detection vs. Innermost Induction Technique The comparison of innermost loop detection with the innermost induction technique (with argument filtering and indefinite lemmas) is shown in Table 12.17. As in the case of full rewriting, it clearly shows the orthogonality of both techniques, such that they should be combined in practice.

To conclude the evaluation of the presented techniques for the inference of lower bounds for innermost rewriting, Table 12.18 compares the best lower bounds inferred by all considered configurations with the respective upper bounds. Thus, this comparison represents the current state of the art, including the techniques presented in this thesis. Again, the fact that there are no entries above the diagonal means that there are no conflicting bounds, i.e., there is no case where the presented techniques yield a lower bound which is greater than the respective upper bound. In 376 of the 517 cases (73%) where a non-trivial lower as well as a non-trivial upper bound is proven, the bounds are tight, i.e., the lower and the upper bound coincide.

Constant Innermost Bounds We finish our evaluation with Table 12.19, which compares the results of the semi-decision procedure for constant innermost bounds from Chapter 10 with the smallest upper bounds inferred by all other considered configurations. The technique from Chapter 10 can infer 3 constant bounds which cannot be inferred by any other configuration. Moreover, whenever any other configuration infers a constant bound, then the technique from Chapter 10 succeeds, too. The average runtime per successfully analyzed example was 1.4 seconds. As in the case of full rewriting, Table 12.18 shows that there are only 6 cases where the technique from Chapter 10 fails and we also fail to disprove constant complexity.

CHAPTER 12. EXPERIMENTS

| Innermost Induction Technique Standalone | | | | | | | |
|--|---------------|-------------|-------------|---------------|---------------|---------------|-------|
| I. I.T. Filtered | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^5)$ | EXP |
| | $\Omega(1)$ | 494 | 1 | — | — | — | — |
| | $\Omega(n)$ | 12 | 423 | — | — | — | — |
| | $\Omega(n^2)$ | 1 | — | 67 | — | — | — |
| | $\Omega(n^3)$ | 1 | — | — | 17 | — | — |
| | $\Omega(n^5)$ | — | — | — | — | 1 | — |
| | EXP | 2 | — | — | — | — | 3 |

Table 12.14: Innermost Induction Technique Filtered vs. Innermost Induction Technique Standalone

| Innermost Induction Technique Standalone | | | | | | | | |
|--|---------------|-------------|-------------|---------------|---------------|---------------|---------------|-------|
| I. I.T. + Indef. Lem. | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | EXP |
| | $\Omega(1)$ | 264 | — | — | — | — | — | — |
| | $\Omega(n)$ | 240 | 412 | — | — | — | 1 | — |
| | $\Omega(n^2)$ | — | 10 | 64 | — | — | — | — |
| | $\Omega(n^3)$ | — | — | 3 | 16 | — | — | — |
| | $\Omega(n^4)$ | — | — | — | 1 | — | — | — |
| | $\Omega(n^5)$ | — | — | — | — | — | — | — |
| | EXP | 6 | 2 | — | — | — | — | 3 |

Table 12.15: Innermost Induction Technique with Indefinite Lemmas vs. Innermost Induction Technique Standalone

| Innermost Induction Technique | | | | | | | |
|-------------------------------|---------------|-------------|-------------|---------------|---------------|---------------|-------|
| S.S. & I. I.T. | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | EXP |
| | $\Omega(1)$ | 251 | — | 1 | — | — | — |
| | $\Omega(n)$ | — | 662 | — | — | — | — |
| | $\Omega(n^2)$ | — | — | 74 | — | — | — |
| | $\Omega(n^3)$ | — | — | — | 20 | — | — |
| | $\Omega(n^4)$ | — | — | — | — | 1 | — |
| | EXP | — | — | — | — | — | 13 |

Table 12.16: Innermost Induction Technique with Strategy Switching vs. Innermost Induction Technique

| Innermost Induction Technique | | | | | | | | |
|-------------------------------|------------------|-------------|-------------|---------------|---------------|---------------|-------|------------------|
| I. Loop Detection | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | EXP | $\Omega(\omega)$ |
| | $\Omega(1)$ | 64 | 2 | 1 | — | — | — | — |
| | $\Omega(n)$ | 123 | 599 | 69 | 18 | 1 | 4 | — |
| | $\Omega(n^2)$ | — | — | — | — | — | — | — |
| | $\Omega(n^3)$ | — | — | — | — | — | — | — |
| | $\Omega(n^4)$ | — | — | — | — | — | — | — |
| | EXP | 7 | 24 | 1 | — | — | 9 | — |
| | $\Omega(\omega)$ | 57 | 37 | 4 | 2 | — | — | — |

Table 12.17: Innermost Loop Detection vs. Innermost Induction Technique

12.2. INNERMOST REWRITING

| Best Innermost Lower Bound | | | | | | | | | | |
|----------------------------|-----------------------|-------------|-------------|---------------|---------------|---------------|---------------|------------------|-------|------------------|
| Best I. Upper Bound | $rc(n)$ | $\Omega(1)$ | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^3)$ | $\Omega(n^4)$ | $\Omega(n^5)$ | $\Omega(n^{10})$ | EXP | $\Omega(\omega)$ |
| | $\mathcal{O}(1)$ | 58 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 4 | 342 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^2)$ | — | 106 | 19 | — | — | — | — | — | — |
| | $\mathcal{O}(n^3)$ | — | 19 | 6 | 11 | — | — | — | — | — |
| | $\mathcal{O}(n^4)$ | — | 2 | — | 3 | — | — | — | — | — |
| | $\mathcal{O}(n^5)$ | — | 3 | — | — | — | — | — | — | — |
| | $\mathcal{O}(n^{10})$ | — | 1 | — | — | — | — | — | — | — |
| | EXP | — | 1 | — | — | — | — | — | 4 | — |
| | $\mathcal{O}(\omega)$ | 2 | 246 | 45 | 4 | 1 | 1 | — | 41 | 103 |

Table 12.18: Best Innermost Upper Bound vs. Best Innermost Lower Bound

| Innermost Constant Bounds | | | | | | | | | | |
|---------------------------|-----------------------|------------------|------------------|--------------------|--------------------|--------------------|--------------------|-----------------------|-------|-----------------------|
| Other I. Upper Bounds | $rc(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n^5)$ | $\mathcal{O}(n^{10})$ | EXP | $\mathcal{O}(\omega)$ |
| | $\mathcal{O}(1)$ | 55 | — | — | — | — | — | — | — | — |
| | $\mathcal{O}(n)$ | 3 | — | — | — | — | — | — | — | 346 |
| | $\mathcal{O}(n^2)$ | — | — | — | — | — | — | — | — | 125 |
| | $\mathcal{O}(n^3)$ | — | — | — | — | — | — | — | — | 36 |
| | $\mathcal{O}(n^4)$ | — | — | — | — | — | — | — | — | 5 |
| | $\mathcal{O}(n^5)$ | — | — | — | — | — | — | — | — | 3 |
| | $\mathcal{O}(n^{10})$ | — | — | — | — | — | — | — | — | 1 |
| | EXP | — | — | — | — | — | — | — | — | 5 |
| | $\mathcal{O}(\omega)$ | — | — | — | — | — | — | — | — | 443 |

Table 12.19: Other Innermost Upper Bounds vs. Innermost Constant Bounds

Part IV

Postface

Conclusion and Outlook

We presented six techniques to analyze the complexity of rewrite systems, each with its respective strengths and weaknesses. Regarding lower bounds, the technique from Chapter 4, which is based on loop acceleration and chaining, supports full integer arithmetic, but its support for recursion is limited. Furthermore, it cannot handle data structures. In contrast, the loop detection technique (Chapter 8) and the induction technique (Chapter 9) lack support for arithmetic, but they can deal with data structures and full recursion. While loop detection is very efficient and widely applicable, it cannot infer super-linear polynomial bounds. In contrast, the induction technique supports such bounds, but at the price of being less efficient.

While we already discussed possible future improvements for the individual techniques in the respective chapters, in the big picture one should combine ideas from Chapter 4, Chapter 8, and Chapter 9 to enable the inference of lower bounds for programs operating on both, integers and data structures. The easiest way to support such programs is to represent numbers as terms and encode the semantics of (fragments of) integer arithmetic into rewrite lemmas to avoid losing domain specific knowledge. In this way, the induction technique could be used to analyze programs with both data structures and integers. However, our technique to reason about families of rewrite sequences via induction from Chapter 9 is certainly not on the same level as highly specialized tools like the recurrence solvers used by the technique from Chapter 4. Thus, one should also consider under-approximating transformations from data structures to numbers. Such transformations would allow us to also use the technique from Chapter 4 to reason about programs with arithmetic and data structures. Note that the generator functions used by the induction technique map natural numbers to data structures, i.e., the idea of representing data structures as numbers has already been exploited in Chapter 9, which shows the feasibility of this idea for the inference of lower bounds. However, these generator functions are currently limited to homogeneous data structure like, e.g., lists of zeros. For now, proving lower bounds by finding families of rewrite sequences that operate on inhomogeneous data structures remains an open

CHAPTER 13. CONCLUSION AND OUTLOOK

research problem.

Regarding upper bounds, Chapter 5 showed how support for recursion can be added to arbitrary existing techniques for the analysis of (non-recursive) integer programs. In Chapter 10 we saw how constant upper bounds for term rewrite systems can be inferred, which turns out to be semi-decidable. Thus, all presented techniques support full recursion. However, the technique from Chapter 5 supports arithmetic, but no data structures, whereas it is the other way around in Chapter 10.

Thus, as for lower bounds, future work may be concerned with the development of techniques which can handle both, data structures and arithmetic, in a sophisticated way. However, the development of such techniques is less urgent than in the case of lower bounds. The reason is that over-approximating transformations from data structures to numbers are very natural and widely used in practice (see, e.g., [2, 7, 51, 103]). Thus, upper bounds for programs with both arithmetic and data structures can already be deduced automatically using such transformations.

Note that the idea to prove constant upper bounds via narrowing from Chapter 10 naturally carries over to richer classes of rewrite systems like rewrite systems with integers and terms (ITRSs). However, for ITRSs one does not obtain a semi-decision procedure for constant upper bounds as for TRSs. The reason is that ITRS rules can encode arbitrary conjunctions of (possibly non-linear) constraints over the integers. Thus, the narrowing relation becomes undecidable due to Hilbert's tenth problem if one extends term rewriting with integer arithmetic.

Finally, the strategy switching technique from Chapter 11 takes a special position, as it has applications for both lower and upper bounds. More precisely, it allows us to use techniques for the inference of upper bounds on the innermost runtime complexity also for full rewriting. Similarly, it makes techniques for the inference of lower bounds on the full runtime complexity applicable to innermost rewriting. While our experiments indicate that the latter is of little practical value, future techniques for lower bounds may exploit the flexibility of full rewriting more aggressively. Then strategy switching might also be of interest in the context of lower bounds.

Regarding richer classes of rewrite systems, there seem to be no significant obstacles regarding an extension to ITRSs or even arbitrary LCTRSs (i.e., term rewrite systems with arbitrary built-in theories), which may be an interesting direction for future work as soon as complexity analysis tools for ITRSs or LCTRSs are available.

Missing Proofs

In this chapter, we present those theorems, lemmas, and proofs which were omitted previously to improve the reading experience. First, we prove the following generalization of Theorem 5.11 to an arbitrary number of defined symbols on the right-hand side.

Theorem A.1 (Size Bounds for ITSs (Generalized)). *Let \mathcal{P} be an ITS whose rules are of the form*

$$\ell \xrightarrow{w} u \ [\varphi]$$

or

$$\ell \xrightarrow{w} u + \sum_{i=1}^m v_i \cdot g_i(\mathbf{t}_i) \ [\varphi]$$

with $u, v_i \in \mathcal{T}(\Sigma_{\mathbb{Z}}, \mathcal{V})$ and $g_i \in \Sigma$. Moreover, let

$$\begin{aligned} \mathcal{P}_{\uparrow} = & \left\{ f'(\mathbf{x}, tv) \xrightarrow{u \cdot tv} \sum_{i=1}^m g'_i(\mathbf{t}_i, v_i \cdot tv) \ [\varphi] \mid f(\mathbf{x}) \xrightarrow{w} u + \sum_{i=1}^m v_i \cdot g_i(\mathbf{t}_i) \ [\varphi] \in \mathcal{P} \right\} \\ & \cup \left\{ f'(\mathbf{x}, tv) \xrightarrow{u \cdot tv} 0 \ [\varphi] \mid f(\mathbf{x}) \xrightarrow{w} u \ [\varphi] \in \mathcal{P} \right\} \end{aligned}$$

for a fresh variable $tv \in \mathcal{V}$ and let rt be a runtime bound for \mathcal{P}_{\uparrow} . Then sz with $\text{sz}(f) = \text{rt}(f')(\mathbf{x}, 1)$ for all $f \in \Sigma$ is a size bound for \mathcal{P} .

Proof. To be able to use the runtime bound rt for \mathcal{P}_{\uparrow} as a size bound for \mathcal{P} , it suffices to prove that $f(\mathbf{n}) \rightarrow_{\mathcal{P}}^* n$ implies $f'(\mathbf{n}, 1) \xrightarrow{n}_{\mathcal{P}_{\uparrow}}^* 0$ for all $k \in \mathbb{N}$, $f \in \Sigma^k$, $\mathbf{n} \in \mathbb{Z}^k$, and $n \in \mathcal{T}(\Sigma_{\mathbb{Z}})$. Instead, we prove the following generalized statement for all $d \in \mathbb{Z}$:

$$f(\mathbf{n}) \rightarrow_{\mathcal{P}}^* n \text{ implies } f'(\mathbf{n}, d) \xrightarrow{n \cdot d}_{\mathcal{P}_{\uparrow}}^* 0 \quad (1)$$

From this, the claim of Theorem A.1 follows.

We prove (1) by induction on the length of the derivation. In the base case, we have $f(\mathbf{n}) \rightarrow_{\mathcal{P}} n$ with some rule $f(\mathbf{x}) \rightarrow u \ [\varphi] \in \mathcal{P}$ and some substitution σ with

APPENDIX A. MISSING PROOFS

$u\sigma = n$, and thus by construction $f'(\mathbf{n}, d) \xrightarrow{n \cdot d}_{\mathcal{P}_\uparrow} 0$ as desired.

In the induction step, we have

$$f(\mathbf{n}) \rightarrow_{\mathcal{P}} \tilde{u} + \sum_{i=1}^m \tilde{v}_i \cdot g_i(\tilde{\mathbf{t}}_i) \rightarrow_{\mathcal{P}}^* n$$

for ground terms $\tilde{u}, \tilde{v}_i \in \mathcal{T}(\Sigma_{\mathbb{Z}})$. Thus, by construction

$$f'(\mathbf{n}, d) \xrightarrow{\tilde{u} \cdot d}_{\mathcal{P}_\uparrow} \sum_{i=1}^m g'_i(\tilde{\mathbf{t}}_i, \tilde{v}_i \cdot d).$$

Let $\tilde{n}_i \in \mathcal{T}(\Sigma_{\mathbb{Z}})$ be the \mathcal{P} -normal forms of $g_i(\tilde{\mathbf{t}}_i)$ such that $n = \tilde{u} + \sum_{i=1}^m \tilde{v}_i \cdot \tilde{n}_i$. From the induction hypothesis (1), we obtain

$$g'_i(\tilde{\mathbf{t}}_i, \tilde{v}_i \cdot d) \xrightarrow{\tilde{n}_i \cdot \tilde{v}_i \cdot d}_{\mathcal{P}_\uparrow}^* 0.$$

Hence, the total cost of the \mathcal{P}_\uparrow -derivation $f'(\mathbf{n}, d) \rightarrow_{\mathcal{P}_\uparrow}^* 0$ is

$$\tilde{u} \cdot d + \sum_{i=1}^m \tilde{n}_i \cdot \tilde{v}_i \cdot d = n \cdot d$$

as desired. □

Next, we finish the proof of Lemma 4.41.

Lemma 4.41 (Bounds for Function Composition). *Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ where $g(m) \in \mathcal{O}(m^d)$ for some $d \in \mathbb{N}$ with $d > 0$. Moreover, let $f(m)$ be weakly and let $g(m)$ be strictly monotonically increasing for large enough m .*

- *If $f(g(m)) \in \Omega(m^k)$ with $k \in \mathbb{N}$, then $f(m) \in \Omega(m^{\frac{k}{d}})$.*
- *If $f(g(m)) \in \Omega(k^m)$ with $k > 1$, then $f(m) \in \Omega(b^{\sqrt[d]{m}})$ for some $b > 1$.*

Proof. We prove the missing case $f(g(m)) \in \Omega(k^m)$. The proof is analogous to the proof of the case $f(g(m)) \in \Omega(m^k)$. Here, $g(m) \in \mathcal{O}(m^d)$ and $f(g(m)) \in \Omega(k^m)$ imply

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq m_0}. g(m) \leq c \cdot m^d \wedge c' \cdot k^m \leq f(g(m)).$$

We can choose m_0 large enough such that $f|_{\mathbb{N}_{\geq m_0}}$ is weakly and $g|_{\mathbb{N}_{\geq m_0}}$ is strictly monotonically increasing. Let $M = \{g(m) \mid m \geq m_0\}$ and let $g^{-1} : M \rightarrow \mathbb{N}_{\geq m_0}$ be the function such that $g(g^{-1}(m)) = m$. By instantiating m with $g^{-1}(m)$,

we obtain

$$\exists m_0, c, c' > 0. \forall m \in M.$$

$$g(g^{-1}(m)) \leq c \cdot (g^{-1}(m))^d \wedge c' \cdot k^{g^{-1}(m)} \leq f(g(g^{-1}(m)))$$

which simplifies to

$$\exists m_0, c, c' > 0. \forall m \in M. m \leq c \cdot (g^{-1}(m))^d \wedge c' \cdot k^{g^{-1}(m)} \leq f(m).$$

When dividing by c and building the d^{th} root on both sides of the first inequality, we get

$$\exists m_0, c, c' > 0. \forall m \in M. \sqrt[d]{\frac{m}{c}} \leq g^{-1}(m) \wedge c' \cdot k^{g^{-1}(m)} \leq f(m).$$

By monotonicity of $\sqrt[d]{\frac{m}{c}}$ and $f(m)$ in m , this implies

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. \sqrt[d]{\frac{m}{c}} \leq \lceil g^{-1} \rceil(m) \wedge c' \cdot k^{\lceil g^{-1} \rceil(m)} \leq f(m).$$

Note that $g|_{\mathbb{N}_{\geq m_0}}$ is total and hence, $g^{-1} : M \rightarrow \mathbb{N}_{\geq m_0}$ is surjective. Moreover, by strict monotonicity of $g|_{\mathbb{N}_{\geq m_0}}$, M is infinite and g^{-1} is also strictly monotonically increasing. Hence, by (4.17) we get $\lceil g^{-1} \rceil(m) \leq \lfloor g^{-1} \rfloor(m) + 1$ for all $m \in \mathbb{N}_{\geq g(m_0)}$. Thus,

$$\exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. \sqrt[d]{\frac{m}{c}} - 1 \leq \lfloor g^{-1} \rfloor(m) \wedge c' \cdot k^{\lfloor g^{-1} \rfloor(m)} \leq f(m)$$

which implies:

$$\begin{aligned} & \exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c' \cdot k^{\sqrt[d]{\frac{m}{c}} - 1} \leq f(m) \\ \iff & \exists m_0, c, c' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. \frac{c'}{k} \cdot k^{\sqrt[d]{\frac{m}{c}}} \leq f(m) \end{aligned}$$

Since $c' > 0$ and $k > 1$, we have $c'' = \frac{c'}{k} > 0$ and thus get:

$$\begin{aligned} & \exists m_0, c, c'' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c'' \cdot k^{\sqrt[d]{\frac{m}{c}}} \leq f(m) \\ \iff & \exists m_0, c, c'' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c'' \cdot k^{\frac{\sqrt[d]{m}}{\sqrt[d]{c}}} \leq f(m) \end{aligned}$$

APPENDIX A. MISSING PROOFS

Since $c > 0$, we have $r = \sqrt[d]{c} > 0$ and hence:

$$\begin{aligned} & \exists m_0, r, c'' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c'' \cdot k^{\frac{d\sqrt{m}}{r}} \leq f(m) \\ \iff & \exists m_0, r, c'' > 0. \forall m \in \mathbb{N}_{\geq g(m_0)}. c'' \cdot \sqrt[r]{k}^{\sqrt{m}} \leq f(m) \end{aligned}$$

Finally, $k > 1$ implies $b = \sqrt[r]{k} > 1$ and we obtain:

$$\begin{aligned} & \exists m_0, c'' > 0, b > 1. \forall m \in \mathbb{N}_{\geq g(m_0)}. c'' \cdot b^{\sqrt{m}} \leq f(m) \\ \implies & \exists b > 1. f(m) \in \Omega(b^{\sqrt{m}}) \end{aligned}$$

□

Finally, we prove Lemma A.2, which was used in the proof of Theorem 8.12.

Lemma A.2 (Ranges and Domains of Pumping Substitutions). *Let there be two compatible loops with pumping substitutions θ and θ' . For any $x \in \text{dom}(\theta)$, we have*

- (a) $\mathcal{V}(x\theta) \cap \text{dom}(\theta) = \{x\}$ and x only occurs once in $x\theta$
- (b) $\mathcal{V}(x\theta) \cap \text{dom}(\theta') \subseteq \{x\}$

Proof. The claim (a) follows from Definition 8.2, as ℓ is linear. For (b), assume there is a $y \in \mathcal{V}(x\theta) \cap \text{dom}(\theta')$ with $y \neq x$. Let ρ_1, \dots, ρ_d be all positions of $x\theta$ where y occurs, i.e., $(x\theta)|_{\rho_1} = \dots = (x\theta)|_{\rho_d} = y$. Thus, ρ_1, \dots, ρ_d are independent positions. Note that

$$x \in \text{dom}(\theta'). \tag{2}$$

To prove (2), note that otherwise, we would have $(x\theta'\theta)|_{\rho_1} = (x\theta)|_{\rho_1} = y$. On the other hand, we obtain $(x\theta\theta')|_{\rho_1} = (x\theta)|_{\rho_1}\theta' = y\theta'$. Since θ and θ' commute, this implies $y = y\theta'$ which is a contradiction to $y \in \text{dom}(\theta')$.

Since $x \in \text{dom}(\theta) \cap \text{dom}(\theta')$, by (a) there exist unique positions $\pi \neq \epsilon$ and $\zeta \neq \epsilon$ such that $(x\theta)|_{\pi} = x$ and $(x\theta')|_{\zeta} = x$. Moreover, (a) implies that applying θ (resp. θ') to any variable $y \neq x$ does not introduce occurrences of x . Hence, x only occurs once in $x\theta'\theta$. Since θ and θ' commute, the same holds for $x\theta\theta'$.

Hence, $(x\theta'\theta)|_{\zeta.\pi} = x$ and $(x\theta\theta')|_{\pi.\zeta} = x$ implies $\zeta.\pi = \pi.\zeta$. This means that

$$\text{there is an } \alpha \in \mathbb{N}^+ \text{ such that } \pi = \alpha^n \text{ and } \zeta = \alpha^m \text{ for } n, m \in \mathbb{N}. \tag{3}$$

Here, α^n stands for the position $\alpha.\alpha \dots \alpha$ where the sequence α is repeated n times. To see why (3) holds, we prove that (3) follows from $\zeta.\pi = \pi.\zeta$ for arbitrary positions π and ζ (in this proof, we also allow $\pi = \epsilon$ or $\zeta = \epsilon$). The

proof is done by induction on π and ζ . In the induction base, $\pi = \epsilon$ or $\zeta = \epsilon$ immediately implies (3). In the induction step, we have $\pi \neq \epsilon$ and $\zeta \neq \epsilon$. W.l.o.g., let $|\pi| \leq |\zeta|$. Then $\zeta.\pi = \pi.\zeta$ implies $\zeta = \pi.\pi'$ for some position π' . Hence, $\zeta.\pi = \pi.\zeta$ now becomes $\pi.\pi'.\pi = \pi.\pi.\pi'$ and thus, $\pi'.\pi = \pi.\pi'$. Since $\pi \neq \epsilon$, the induction hypothesis implies $\pi = \alpha^n$ and $\pi' = \alpha^m$ for some $\alpha \in \mathbb{N}^+$ and $n, m \in \mathbb{N}$. Thus, $\zeta = \pi.\pi' = \alpha^{n+m}$, which proves (3).

We now perform a case analysis on the relationship between π and ζ .

Case 1. $\pi \leq \zeta$

In this case, we have $\zeta = \pi.\pi'$ for some position π' . We obtain $(x\theta'\theta)|_{\zeta.\rho_1} = ((x\theta')|_{\zeta}\theta)|_{\rho_1} = (x\theta)|_{\rho_1} = y$. The commutation of θ' and θ implies that we also have $(x\theta\theta')|_{\zeta.\rho_1} = y$. However,

$$(x\theta\theta')|_{\zeta.\rho_1} = (x\theta\theta')|_{\pi.\pi'.\rho_1} = ((x\theta)|_{\pi}\theta')|_{\pi'.\rho_1} = (x\theta')|_{\pi'.\rho_1}.$$

Note that $x\theta'$ cannot contain the variable y , since $y \in \text{dom}(\theta')$ by the assumption at the beginning of the proof and $\mathcal{V}(x\theta') \cap \text{dom}(\theta') = \{x\}$ by (2) and (a).¹ Thus, this contradicts $(x\theta\theta')|_{\zeta.\rho_1} = y$.

Case 2. $\pi \not\leq \zeta$

By (3), we have $\zeta = \alpha^m$ and $\pi = \alpha^{m+k}$ for $m > 0$ and $k > 0$ (since $\zeta \neq \epsilon$ and $\pi \neq \epsilon$). As $y \in \text{dom}(\theta')$, by (a) there is a unique position κ such that $y\theta'|_{\kappa} = y$. Recall that ρ_1, \dots, ρ_d are the only positions where y occurs in $x\theta$. Due to (a), $\rho_1.\kappa, \dots, \rho_d.\kappa$ are the only positions where y occurs in $x\theta\theta'$ (since $(x\theta\theta')|_{\rho_i.\kappa} = ((x\theta)_{\rho_i}\theta')|_{\kappa} = (y\theta')|_{\kappa} = y$). Similarly, $\zeta.\rho_1, \dots, \zeta.\rho_d$ are the only positions where y occurs in $x\theta'\theta$ (since $(x\theta'\theta)|_{\zeta.\rho_i} = ((x\theta')|_{\zeta}\theta)|_{\rho_i} = (x\theta)|_{\rho_i} = y$). As $x\theta\theta' = x\theta'\theta$, the positions $\rho_1.\kappa, \dots, \rho_d.\kappa$ are the same as the positions $\zeta.\rho_1, \dots, \zeta.\rho_d$. Let ρ_1, \dots, ρ_d be ordered according to the (total) lexicographic ordering \sqsubset on tuples of numbers (i.e., $\rho_1 \sqsubset \rho_2 \sqsubset \dots \sqsubset \rho_d$).² Then we also have $\rho_1.\kappa \sqsubset \dots \sqsubset \rho_d.\kappa$ (as the ρ_i are independent positions) and $\zeta.\rho_1 \sqsubset \dots \sqsubset \zeta.\rho_d$. This implies $\rho_i.\kappa = \zeta.\rho_i$ for all $i \in \{1, \dots, d\}$, i.e., in particular $\rho_1.\kappa = \zeta.\rho_1$. As $\zeta = \alpha^m$, this means $\rho_1.\kappa = \alpha^m.\rho_1$.

Let e be the largest number such that $\rho_1 = \alpha^e.\rho'$ for some position ρ' . Thus, α is no prefix of ρ' . We perform a case analysis on the relation between e and k .

Case 2.1. $e \geq k$

Then

$$\begin{aligned} y &= (x\theta\theta')|_{\alpha^m.\rho_1} = (x\theta\theta')|_{\alpha^{m+e}.\rho'} \sqsubseteq (x\theta\theta')|_{\alpha^{m+k}} \\ &= (x\theta\theta')|_{\pi} = (x\theta)|_{\pi}\theta' = x\theta'. \end{aligned}$$

¹ To see why $\mathcal{V}(x\theta') \cap \text{dom}(\theta') = \{x\}$ holds, note that we have $x \in \text{dom}(\theta')$ by (2). Since (a) holds for the pumping substitution of any decreasing loop, it also holds for θ' . Hence, $x \in \text{dom}(\theta')$ implies $\mathcal{V}(x\theta') \cap \text{dom}(\theta') = \{x\}$.

²I.e., we have $(a_1 \dots a_n) \sqsubset (b_1 \dots b_m)$ if and only if $n = 0$ and $m > 0$ or $a_1 < b_1$ or $a_1 = b_1$ and $(a_2 \dots a_n) \sqsubset (b_2 \dots b_m)$.

APPENDIX A. MISSING PROOFS

But this contradicts (a), as $x \in \text{dom}(\theta')$ by (2). Thus, $x\theta'$ cannot contain y .

Case 2.2. $e < k$

Note that $\rho_1.\kappa = \alpha^m.\rho_1$ implies $\alpha^e.\rho'.\kappa = \alpha^m.\alpha^e.\rho'$, i.e., $\rho'.\kappa = \alpha^m.\rho'$. Since α is no prefix of ρ' , ρ' must be a (proper) prefix of α , since $m > 0$. Thus, we have $\rho' < \alpha$, which implies $\alpha^m.\rho_1 = \alpha^{m+e}.\rho' < \alpha^{m+e+1} \leq \alpha^{m+k}$, as $e < k$. Hence, we have

$$y = (x\theta\theta')|_{\alpha^m.\rho_1} \triangleright (x\theta\theta')|_{\alpha^{m+k}} = (x\theta\theta')|_{\pi} = (x\theta)|_{\pi}\theta' = x\theta'.$$

This is an immediate contradiction, because the variable y cannot have a proper subterm. \square

Examples Mentioned in Experimental Evaluations

In this chapter, we provide the names of those examples which are mentioned in the experimental evaluations of the presented techniques as their results are particularly interesting in some sense. Note that these names are of little interest on their own, such that they were not included in the main part of this thesis. However, they might be useful for the reproduction and verification of our experimental results.

B.1 Evaluation of Chapter 5

For the following 16 examples, the technique from Chapter 5 successfully deduced an upper bound and thereby inferred at least one super-linear polynomial size bound, whereas CoFloCo failed.

- AG01/#3.16
- AProVE_04/IJCAR_18
- AProVE_04/IJCAR_26a
- AProVE_04/IJCAR_26
- AProVE_07/kabasci02
- AProVE_07/thiemann03
- AProVE_07/thiemann18
- Frederiksen_Glenstrup/mul.better
- Frederiksen_Glenstrup/mul
- Rubio_04/selsort
- SK90/2.12
- Strategy_removed_AG01/#4.36
- Transformed_CSR_04/Ex2_Luc02a_L
- Transformed_CSR_04/Ex2_Luc02a_Z
- Transformed_CSR_04/ExSec11_1_Luc02a_L
- Transformed_CSR_04/ExSec11_1_Luc02a_Z

In the following 8 cases, the technique from Chapter 5 deduced an exponential upper bound and at least one exponential size bound, whereas CoFloCo failed.

- SK90/2.15
- SK90/2.21
- SK90/2.24
- SK90/4.05
- SK90/4.10
- Transformed_CSR_04/Ex1_Luc02b_Z
- Transformed_CSR_04/Ex4_Zan97_Z
- Transformed_CSR_04/Ex7_BLR02_Z

B.2 Evaluation of Part III

B.2.1 Evaluation of Full Rewriting

In the following 5 cases, the semi-decision procedure from Chapter 10 failed to prove a constant upper bound and loop detection timed out:

- Transformed_CSR_04/LISTUTILITIES_complete_noand_C
- Transformed_CSR_04/LISTUTILITIES_complete_C
- Transformed_CSR_04/LISTUTILITIES_nokinds_noand_C
- Transformed_CSR_04/LISTUTILITIES_complete_noand_GM
- Transformed_CSR_04/OvConsOS_complete_noand_C

The following two examples where both the technique from Chapter 10 and loop detection failed are non-left-linear.

- SK90/2.61
- Strategy_removed_mixed_05/ex6

Moreover, both techniques failed for the non-constructor system

Transformed_CSR_04/ExIntro.GM99.Z.

Also, both techniques failed for

SK90/4.57,

which has constant runtime complexity, but the semi-decision procedure from Chapter 10 timed out. Finally,

Transformed_CSR_04/LISTUTILITIES_nosorts-noand.Z

has the following decreasing loop of length 13, which was not found by our heuristics.

```
U61(tt, s(x), y, cons(z, zs))
→ U62(tt, activate(s(x)), activate(y), activate(cons(z, zs)))
→3 U62(tt, s(x), y, cons(z, zs))
→ U63(tt, activate(s(x)), activate(y), activate(cons(z, zs)))
→3 U63(tt, s(x), y, cons(z, zs))
→ U64(splitAt(activate(s(x)), activate(cons(z, zs))), activate(y))
→2 U64(splitAt(s(x), cons(z, zs)), activate(y))
→ U64(U61(tt, x, z, activate(zs)), activate(y))
→ U64(U61(tt, x, z, zs), activate(y))
```

APPENDIX B. EXAMPLES MENTIONED IN EXPERIMENTAL EVALUATIONS

The examples with nested recursion where the combination of argument filtering and the induction technique could prove exponential bounds, whereas the induction technique (standalone) and loop detection failed are:

- AProVE_07/thiemann08
- Rubio_04/nestrec

The semi-decision procedure from Chapter 10 inferred constant upper bounds for the following 5 examples where all other configurations failed to do so:

- Secret_06_TRS/10
- Secret_06_TRS/4
- SK90/2.59
- SK90/4.51
- Transformed_CSR_04/Ex24_GM04_Z

The 6 examples where neither a linear lower nor a constant upper bounds was proven in our experiments are:

- SK90/2.61
- SK90/4.57
- Strategy_removed_mixed_05/ex6
- Transformed_CSR_04/ExIntrod_GM99_Z
- LISTUTILITIES_complete_noand_GM
- Transformed_CSR_04/LISTUTILITIES_nosorts--noand_Z

B.2.2 Evaluation of Innermost Rewriting

In the following 3 cases, the semi-decision procedure from Chapter 10 failed to prove a constant upper bound and loop detection timed out:

- Frederiksen_Others/rematch
- Transformed_CSR_04/LISTUTILITIES_complete_noand_C
- Transformed_CSR_04/LISTUTILITIES_complete_noand_GM

The following two examples where both the technique from Chapter 10 and loop detection failed are non-left-linear.

- SK90/2.61

B.2. EVALUATION OF PART III

- Strategy_removed_mixed_05/ex6

Moreover, both techniques failed for the following two non-ordinary TRSs:

- Frederiksen_Glenstrup/ordered_better
- Frederiksen_Glenstrup/nolexicord

The example,

Frederiksen_Glenstrup/nestdec

contains the rule

$$\text{dec}(\text{cons}(\text{nil}, \text{cons}(x, xs))) \rightarrow \text{dec}(\text{cons}(x, xs))$$

which gives rise to a decreasing loop by instantiating x with nil , but this loop was not found by our heuristics. Finally,

Transformed_CSR_04/LISTUTILITIES_nosorts-noand_Z

has a decreasing loop of length 13 (cf. Appendix B.2.1), which was not found by our heuristics.

As in the case of full rewriting, the examples with nested recursion where the combination of argument filtering and the induction technique could prove exponential bounds, whereas the induction technique (standalone) and loop detection failed are:

- AProVE_07/thiemann08
- Rubio_04/nestrec

The semi-decision procedure from Chapter 10 inferred constant upper bounds for the following 3 examples where all other configurations failed to do so:

- Secret_06_TRS/4
- SK90/2.59
- SK90/4.51

The 6 examples where neither a linear lower nor a constant upper bounds was proven in our experiments are:

- Frederiksen_Glenstrup/nolexicord
- Frederiksen_Glenstrup/ordered_better
- SK90/2.61

APPENDIX B. EXAMPLES MENTIONED IN EXPERIMENTAL EVALUATIONS

- Strategy_removed.mixed.05/ex6
- Transformed_CSR.04/LISTUTILITIES_complete_noand.GM
- Transformed_CSR.04/LISTUTILITIES_nosorts--noand.Z

Bibliography

- [1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. “Closed-Form Upper Bounds in Static Cost Analysis”. In: *JAR* 46.2 (2011), pp. 161–203.
- [2] E. Albert, S. Genaim, and R. Gutiérrez. “A Transformational Approach to Resource Analysis with Typed-Norms”. In: *LOPSTR ’13*. LNCS 8901. 2013, pp. 38–53.
- [3] E. Albert, S. Genaim, and A. N. Masud. “On the Inference of Resource Usage Upper and Lower Bounds”. In: *ACM TOCL* 14.3 (2013), 22:1–22:35.
- [4] E. Albert, D. E. Alonso-Blas, P. Arenas, S. Genaim, and G. Puebla. “Asymptotic Resource Usage Bounds”. In: *APLAS ’09*. LNCS 5904. 2014, pp. 294–310.
- [5] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. “Termination Analysis of Java Bytecode”. In: *FMOODS ’08*. LNCS 5051. 2008, pp. 2–18.
- [6] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. “SACO: Static Analyzer for Concurrent Objects”. In: *TACAS ’14*. LNCS 8413. 2014, pp. 562–567.
- [7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. “Cost Analysis of Object-Oriented Bytecode Programs”. In: *TCS* 413.1 (2012), pp. 142–159.
- [8] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. “Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs”. In: *SAS ’10*. LNCS 6337. 2010, pp. 117–133.
- [9] D. E. Alonso-Blas, P. Arenas, and S. Genaim. “Precise Cost Analysis via Local Reasoning”. In: *ATVA ’13*. LNCS 8172. 2013, pp. 319–333.
- [10] D. E. Alonso-Blas and S. Genaim. “On the Limits of the Classical Approach to Cost Analysis”. In: *SAS ’12*. LNCS 7460. 2012, pp. 405–421.

BIBLIOGRAPHY

- [11] M. Alpuente, S. Escobar, and J. Iborra. “Termination of Narrowing Revisited”. In: *TCS* 410.46 (2009), pp. 4608–4625.
- [12] T. Arts and J. Giesl. “Termination of Term Rewriting Using Dependency Pairs”. In: *TCS* 236.1–2 (2000), pp. 133–178.
- [13] M. Avanzini and G. Moser. “Polynomial Path Orders”. In: *LMCS* 9.4 (2013).
- [14] M. Avanzini and G. Moser. “Dependency Pairs and Polynomial Path Orders”. In: *RTA ’09*. LNCS 5595. 2009, pp. 48–62.
- [15] M. Avanzini and G. Moser. “A Combination Framework for Complexity”. In: *IC* 248 (2016), pp. 22–55.
- [16] M. Avanzini and G. Moser. “Complexity of Acyclic Term Graph Rewriting”. In: *FSCD ’16*. LIPIcs 52. 2016, 10:1–10:18.
- [17] M. Avanzini, G. Moser, and M. Schaper. “TCT: Tyrolean Complexity Tool”. In: *TACAS ’16*. LNCS 9636. 2016, pp. 407–423.
- [18] R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. “PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis”. In: *CoRR* abs/cs/0512056 (2005).
- [19] A. M. Ben-Amram and S. Genaim. “Ranking Functions for Linear-Constraint Loops”. In: *Journal of the ACM* 61.4 (2014), 26:1–26:55.
- [20] *KoAT Benchmarks*. 2014. URL: <https://github.com/s-falke/kittel-koat/tree/master/koat-evaluation/examples> (visited on 02/05/2018).
- [21] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. “ABC: Algebraic Bound Computation for Loops”. In: *LPAR ’10*. LNCS 6355. 2010, pp. 103–118.
- [22] C. Bouchard, K. A. Gero, C. Lynch, and P. Narendran. “On Forward Closure and the Finite Variant Property”. In: *FroCoS ’13*. LNCS 8152. 2013, pp. 327–342.
- [23] R. S. Boyer and J. Moore. *A Computational Logic*. Academic Press, 1979.
- [24] M. Bozga, R. Iosif, and F. Konečný. “Fast Acceleration of Ultimately Periodic Relations”. In: *CAV ’10*. LNCS 6174. 2010, pp. 227–242.
- [25] A. R. Bradley, Z. Manna, and H. B. Sipma. “Linear Ranking with Reachability”. In: *CAV ’05*. LNCS 3576. 2005, pp. 491–504.
- [26] M. Brockschmidt, B. Cook, and C. Fuhs. “Better Termination Proving Through Cooperation”. In: *CAV ’13*. LNCS 8044. 2013, pp. 413–429.
- [27] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. “Analyzing Runtime and Size Complexity of Integer Programs”. In: *ACM TOPLAS* 38.4 (2016), 13:1–13:50.

BIBLIOGRAPHY

- [28] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. “Automated Termination Proofs for Java Programs with Cyclic Data”. In: *CAV ’12*. LNCS 7358. 2012, pp. 105–122.
- [29] M. Brockschmidt, C. Otto, and J. Giesl. “Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting”. In: *RTA ’11*. LIPIcs 10. 2011, pp. 155–170.
- [30] H. J. S. Bruggink, B. König, D. Nolte, and H. Zantema. “Proving Termination of Graph Transformation Systems Using Weighted Type Graphs over Semirings”. In: *ICGT ’15*. LNCS 9151. 2015, pp. 52–68.
- [31] H. J. S. Bruggink, B. König, and H. Zantema. “Termination Analysis for Graph Transformation Systems”. In: *IFIP TCS ’14*. LNCS 8705. 2014, pp. 179–194.
- [32] J. Burnim, S. Juvekar, and K. Sen. “WISE: Automated Test Generation for Worst-Case Complexity”. In: *ICSE*. 2009, pp. 463–473.
- [33] P. Cameron. *Polynomials taking integer values*. 2017. URL: <https://cameroncounts.wordpress.com/2017/01/31/polynomials-taking-integer-values/> (visited on 02/02/2018).
- [34] Q. Carbonneaux, J. Hoffmann, T. W. Reps, and Z. Shao. “Automated Resource Analysis with Coq Proof Objects”. In: *CAV ’17*. LNCS 10427. 2017, pp. 64–85.
- [35] Q. Carbonneaux, J. Hoffmann, and Z. Shao. “Compositional Certified Resource Bounds”. In: *PLDI ’15*. 2015, pp. 467–478.
- [36] J. Cassaigne, N. Ollinger, and R. Torres-Avilés. “A small minimal aperiodic reversible Turing machine”. In: *Journal of Computer and System Sciences* 84 (2017), pp. 288–301.
- [37] J. Christ, J. Hoenicke, and A. Nutz. “SMTInterpol: An Interpolating SMT Solver”. In: *SPIN ’12*. LNCS 7385. 2012, pp. 248–254.
- [38] M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J. Giesl. “SAT Solving for Argument Filterings”. In: *LPAR ’06*. LNAI 4246. 2006, pp. 30–44.
- [39] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS ’08*. LNCS 4963. 2008, pp. 337–340.
- [40] S. Debray, P. López-García, M. V. Hermenegildo, and N. Lin. “Lower Bound Cost Estimation for Logic Programs”. In: *ILPS ’97*. 1997, pp. 291–305.
- [41] F. Emmes, T. Enger, and J. Giesl. “Proving Non-Looping Non-Termination Automatically”. In: *IJCAR ’12*. LNCS 7364. 2012, pp. 225–240.
- [42] F. Emmes and L. Noschinski. *Oops*. 2010. URL: <http://www.termination-portal.org/wiki/Tools:Oops>.

BIBLIOGRAPHY

- [43] J. Endrullis and H. Zantema. “Proving Non-Termination by Finite Automata”. In: *RTA ’15*. LIPIcs 36. 2015, pp. 160–176.
- [44] S. Falke, D. Kapur, and C. Sinz. “Termination Analysis of C Programs Using Compiler Intermediate Languages”. In: *RTA ’11*. LIPIcs 10. 2011, pp. 41–50.
- [45] S. Falke, D. Kapur, and C. Sinz. “Termination Analysis of Imperative Programs Using Bitvector Arithmetic”. In: *VSTTE ’12*. LNCS 7152. 2012, pp. 261–277.
- [46] A. Farzan and Z. Kincaid. “Compositional Recurrence Analysis”. In: *FMCAD ’15*. 2015, pp. 57–64.
- [47] G. Feuillade, T. Genet, and V. V. T. Tong. “Reachability Analysis over Term Rewriting Systems”. In: *JAR* 33.3–4 (2004), pp. 341–383.
- [48] A. Flores-Montoya. “Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations”. In: *FM ’16*. LNCS 9995. 2016, pp. 254–273.
- [49] A. Flores-Montoya. “Cost Analysis of Programs Based on the Refinement of Cost Relations”. PhD thesis. Darmstadt University of Technology, 2017.
- [50] A. Flores-Montoya and R. Hähnle. “Resource Analysis of Complex Programs with Cost Equations”. In: *APLAS ’14*. LNCS 8858. 2014, pp. 275–295.
- [51] F. Frohn and J. Giesl. “Complexity Analysis for Java with AProVE”. In: *iFM ’17*. LNCS 10510. 2017, pp. 85–101.
- [52] F. Frohn. *Evaluation Website*. 2018. URL: <https://ffrohn.github.io/thesis-evaluation/>.
- [53] F. Frohn and J. Giesl. “Analyzing Runtime Complexity via Innermost Runtime Complexity”. In: *LPAR ’17*. EPiC Series in Computing 46. 2017, pp. 249–268.
- [54] F. Frohn, J. Giesl, F. Emmes, T. Ströder, C. Aschermann, and J. Hensel. “Inferring Lower Bounds for Runtime Complexity”. In: *RTA ’15*. LIPIcs 36. 2015, pp. 334–349.
- [55] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. “Lower Bounds for Runtime Complexity of Term Rewriting”. In: *JAR* 59.1 (2017), pp. 121–163.
- [56] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. “Lower Runtime Bounds for Integer Programs”. In: *IJCAR ’16*. LNCS 9706. 2016, pp. 550–567.
- [57] C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. “Proving Termination by Dependency Pairs and Inductive Theorem Proving”. In: *JAR* 47.2 (2011), pp. 133–160.

BIBLIOGRAPHY

- [58] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. “Proving Termination of Integer Term Rewriting”. In: *RTA '09*. LNCS 5595. 2009, pp. 32–47.
- [59] C. Fuhs, C. Kop, and N. Nishida. “Verifying Procedural Programs via Constrained Rewriting Induction”. In: *ACM TOCL* 18.2 (2017), 14:1–14:50.
- [60] T. Genet. *Reachability Analysis of Rewriting for Software Verification*. 2009.
- [61] T. Genet and V. V. T. Tong. “Reachability Analysis of Term Rewriting Systems with Timbuk”. In: *LPAR '01*. LNAI 2250. 2001, pp. 695–706.
- [62] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. “Analyzing Program Termination and Complexity Automatically with AProVE ”. In: *JAR* 58.1 (2017), pp. 3–31.
- [63] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. “Automated Termination Proofs for Haskell by Term Rewriting”. In: *ACM TOPLAS* 33.2 (2011), 7:1–7:39.
- [64] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. “Proving Termination of Programs Automatically with AProVE ”. In: *IJCAR '14*. LNCS 8562. 2014, pp. 184–191.
- [65] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. “Symbolic Evaluation Graphs and Term Rewriting: A General Methodology for Analyzing Logic Programs”. In: *PPDP '12*. 2012, pp. 1–12.
- [66] J. Giesl, R. Thiemann, and P. Schneider-Kamp. “The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs”. In: *LPAR '04*. LNAI 3452. 2004, pp. 301–331.
- [67] J. Giesl, R. Thiemann, and P. Schneider-Kamp. “Proving and Disproving Termination of Higher-Order Functions”. In: *FroCoS '05*. LNCS 3717. 2005, pp. 216–231.
- [68] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. “Mechanizing and Improving Dependency Pairs”. In: *JAR* 37.3 (2006), pp. 155–203.
- [69] L. Gonnord and N. Halbwachs. “Combining Widening and Acceleration in Linear Relation Analysis”. In: *SAS '06*. LNCS 4134. 2006, pp. 144–160.
- [70] B. Gramlich. “Abstract Relations between Restricted Termination and Confluence Properties of Rewrite Systems”. In: *Fundamenta Informaticae* 24.1/2 (1995), pp. 2–23.
- [71] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. “SPEED: Precise and Efficient Static Estimation of Program Computational Complexity”. In: *POPL '09*. 2009, pp. 127–139.

BIBLIOGRAPHY

- [72] S. Gulwani. “SPEED: Symbolic Complexity Bound Analysis”. In: *CAV ’09*. LNCS 5643. 2009, pp. 51–62.
- [73] G. Hardy and J. Littlewood. “Some Problems of Diophantine Approximation, Part II”. In: *Acta Mathematica* 37.1 (1914), pp. 193–239.
- [74] A. Heck. *Introduction to Maple (2. ed.)* Springer, 1996.
- [75] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. “Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution”. In: *SEFM ’16*. LNCS 9763. 2016, pp. 234–252.
- [76] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. “Termination and Complexity Analysis for Programs with Bitvector Arithmetic by Symbolic Execution”. In: *JLAMP* 97 (2018), pp. 105–130.
- [77] N. Hirokawa, A. Middeldorp, and H. Zankl. “Uncurrying for Termination and Complexity”. In: *JAR* 50.3 (2013), pp. 279–315.
- [78] N. Hirokawa and A. Middeldorp. “Tyrolean Termination Tool: Techniques and Features”. In: *IC* 205.4 (2007), pp. 474–511.
- [79] N. Hirokawa and G. Moser. “Automated Complexity Analysis Based on the Dependency Pair Method”. In: *IJCAR ’08*. LNCS 5195. 2008, pp. 364–379.
- [80] D. Hofbauer and J. Waldmann. *Constructing Lower Bounds on the Derivational Complexity of Rewrite Systems*. Slides of a talk at the *2nd Workshop on Proof Theory and Rewriting*. 2010. URL: <http://www.imn.htwk-leipzig.de/~waldmann/talk/10/pr/main.pdf> (visited on 02/15/2018).
- [81] D. Hofbauer and C. Lautemann. “Termination Proofs and the Length of Derivations”. In: *RTA ’89*. LNCS 355. 1989, pp. 167–177.
- [82] J. Hoffmann, A. Das, and S. Weng. “Towards Automatic Resource Bound Analysis for OCaml”. In: *POPL ’17*. 2017, pp. 359–373.
- [83] J. Hoffmann. “Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis”. PhD thesis. Ludwig Maximilians University Munich, 2011.
- [84] J. Hoffmann, K. Aehlig, and M. Hofmann. “Multivariate Amortized Resource Analysis”. In: *ACM TOPLAS* 34.3 (2012), 14:1–14:62.
- [85] J. Hoffmann and Z. Shao. “Type-Based Amortized Resource Analysis with Integers and Arrays”. In: *Journal of Functional Programming* 25 (2015).
- [86] M. Hofmann and G. Moser. “Amortised Resource Analysis and Typed Polynomial Interpretations”. In: *RTA ’14*. LNCS 8560. 2014, pp. 272–286.
- [87] M. Hofmann and G. Moser. “Multivariate Amortised Resource Analysis for Term Rewrite Systems”. In: *TLCA ’15*. LIPIcs 38. 2015, pp. 241–256.

BIBLIOGRAPHY

- [88] P. K. Hooper. “The Undecidability of the Turing Machine Immortality Problem”. In: *Journal of Symbolic Logic* 31.2 (1966), pp. 219–234.
- [89] J. M. Hullot. “Canonical Forms and Unification”. In: *CADE ’80*. LNCS 87. 1980, pp. 318–334.
- [90] B. Jeannet and A. Miné. “APRON: A Library of Numerical Abstract Domains for Static Analysis”. In: *CAV ’09*. LNCS 5643. 2009, pp. 661–667.
- [91] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. “Abstract Acceleration of General Linear Loops”. In: *POPL ’14*. 2014, pp. 529–540.
- [92] D. E. Knuth. “Big Omicron and Big Omega and Big Theta”. In: *ACM SIGACT News* 8.2 (1976), pp. 18–23.
- [93] D. E. Knuth. “Johann Faulhaber and Sums of Powers”. In: *Mathematics of Computation* 61.203 (1993), pp. 277–294.
- [94] C. Kop and N. Nishida. “Constrained Term Rewriting tool”. In: *LPAR ’15*. LNCS 9450. 2015, pp. 549–557.
- [95] D. Kroening, M. Lewis, and G. Weissenbacher. “Under-Approximating Loops in C Programs for Fast Counterexample Detection”. In: *FMSD* 47.1 (2015), pp. 75–92.
- [96] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. “Proving Termination of Imperative Programs Using Max-SMT”. In: *FMCAD ’13*. 2013, pp. 218–225.
- [97] D. Lea, J. Bloch, A. van Hoff, and N. Gafter. *HashMap.java*. 2014. URL: <http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java> (visited on 01/31/2018).
- [98] C. Lynch and B. Morawska. “Basic Syntactic Mutation”. In: *CADE ’02*. LNCS 2392. 2002, pp. 471–485.
- [99] K. Madhukar, B. Wachter, D. Kroening, M. Lewis, and M. K. Srivas. “Accelerating Invariant Generation”. In: *FMCAD ’15*. 2015, pp. 105–111.
- [100] R. Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375.
- [101] G. Moser and M. Schaper. *From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation*. Preprint. 2016. URL: <http://cbr.uibk.ac.at/publications/ic16.pdf> (visited on 01/30/2018).
- [102] M. Naaf et al. *LoAT*. 2016. URL: <https://github.com/aprove-developers/LoAT> (visited on 02/05/2018).
- [103] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. “Complexity Analysis for Term Rewriting by Integer Transition Systems”. In: *FroCoS ’17*. LNCS 10483. 2017, pp. 132–150.

BIBLIOGRAPHY

- [104] F. Neurauter, H. Zankl, and A. Middeldorp. “Revisiting Matrix Interpretations for Polynomial Derivational Complexity of Term Rewriting”. In: *LPAR ’10*. LNCS 6397. 2010, pp. 550–564.
- [105] L. Noschinski, F. Emmes, and J. Giesl. “Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs”. In: *JAR* 51.1 (2013), pp. 27–56.
- [106] V. van Oostrom. “Random Descent”. In: *RTA ’07*. LNCS 4533. 2007, pp. 314–328.
- [107] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. “Automated Termination Analysis of Java Bytecode by Term Rewriting”. In: *RTA ’10*. LIPIcs 6. 2010, pp. 259–276.
- [108] É. Payet. “Loop Detection in Term Rewriting Using the Eliminating Unfoldings”. In: *TCS* 403.2–3 (2008), pp. 307–327.
- [109] A. Podelski and A. Rybalchenko. “A Complete Method for the Synthesis of Linear Ranking Functions”. In: *VMCAI ’04*. LNCS 2937. 2004, pp. 239–251.
- [110] C. Roux et al. *CAGE*. 2017. URL: github.com/draperlaboratory/Cage-Public (visited on 02/01/2018).
- [111] M. Sakai, K. Okamoto, and T. Sakabe. “Innermost Reductions Find All Normal Forms on Right-Linear Terminating Overlay TRSs”. In: *WRS ’03*. 2003.
- [112] A. Serrano, P. López-García, and M. V. Hermenegildo. “Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types”. In: *TPLP* 14.4–5 (2014), pp. 739–754.
- [113] M. Sinn, F. Zuleger, and H. Veith. “A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis”. In: *CAV ’14*. LNCS 8559. 2014, pp. 745–761.
- [114] M. Sinn, F. Zuleger, and H. Veith. “Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints”. In: *JAR* 59.1 (2017), pp. 3–45.
- [115] F. Spoto, F. Mesnard, and É. Payet. “A Termination Analyser for Java Bytecode Based on Path-Length”. In: *ACM TOPLAS* 32.3 (2010), 8:1–8:70.
- [116] A. Srikanth, B. Sahin, and W. R. Harris. “Complexity Verification Using Guided Theorem Enumeration”. In: *POPL ’17*. 2017, pp. 639–652.
- [117] T. Sternagel, A. Middeldorp, and C. Kop. “Complexity of Conditional Term Rewriting”. In: *LMCS* 13.1 (2017).
- [118] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, and P. Schneider-Kamp. “Proving Termination and Memory Safety for Programs with Pointer Arithmetic”. In: *IJCAR ’14*. LNCS 8562. 2014, pp. 208–223.

BIBLIOGRAPHY

- [119] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. “Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic”. In: *JAR* 58.1 (2017), pp. 33–65.
- [120] R. E. Tarjan. “Amortized Computational Complexity”. In: *SIAM Journal of Algebraic Discrete Methods* 6 (1985), pp. 306–318.
- [121] *Termination and Complexity Competition*. URL: http://termination-portal.org/wiki/Termination_Competition (visited on 02/05/2018).
- [122] *Termination Problems Data Base*. URL: <http://termination-portal.org/wiki/TPDB> (visited on 02/07/2018).
- [123] R. Thiemann, J. Giesl, and P. Schneider-Kamp. “Deciding Innermost Loops”. In: *RTA ’08*. LNCS 5117. 2008, pp. 366–380.
- [124] J. van de Pol and H. Zantema. “Generalized Innermost Rewriting”. In: *RTA ’05*. LNCS 3467. 2005, pp. 2–16.
- [125] J. Waldmann. “Matchbox: A Tool for Match-Bounded String Rewriting”. In: *RTA ’04*. LNCS 3091. 2004, pp. 85–94.
- [126] J. Waldmann. “Automatic Termination”. In: *RTA ’09*. LNCS 5595. 2009, pp. 1–16.
- [127] J. Waldmann. “Polynomially Bounded Matrix Interpretations”. In: *RTA ’10*. LIPIcs 6. 2010, pp. 357–372.
- [128] B. Wegbreit. “Mechanical Program Analysis”. In: *Communications of the ACM* 18 (1975), pp. 528–539.
- [129] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. “The Worst-Case Execution-Time Problem: Overview of Methods and Survey of Tools”. In: *ACM Transactions in Embedded Computing Systems* 7.3 (2008), 36:1–36:53.
- [130] S. Wolfram. “Mathematica: A System for Doing Mathematics by Computer”. In: *SIAM Review* 34.3 (1992), pp. 519–522.
- [131] H. Zankl and M. Korp. “Modular Complexity Analysis for Term Rewriting”. In: *LMCS* 10.1:19 (2014), pp. 1–33.
- [132] H. Zankl, C. Sternagel, D. Hofbauer, and A. Middeldorp. “Finding and Certifying Loops”. In: *SOFSEM ’10*. LNCS 5901. 2010, pp. 755–766.
- [133] H. Zantema. “Termination of Term Rewriting: Interpretation and Type Elimination”. In: *JSC* 17.1 (1994), pp. 23–50.
- [134] H. Zantema. “Termination of String Rewriting Proved Automatically”. In: *JAR* 34.2 (2005), pp. 105–139.