master thesis in computer science

by

Manuel Schneckenreither, MSc

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Assoc. Prof. Dr. Georg Moser, Department of Computer Science

Innsbruck, 17 July 2018

Master Thesis

Amortized Resource Analysis for Term Rewrite Systems

Manuel Schneckenreither, MSc (1117198)
manuel.schneckenreither@student.uibk.ac.at

17 July 2018

Supervisor: Assoc. Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

Based on earlier work on amortised resource analysis, we establish two novel automated amortised resource analyses for term rewrite systems. On one hand the worst-case analysis gives rise to polynomial bounds on the innermost runtime complexity of the analysed term rewrite system while on the other hand we present a best-case analysis for term rewrite systems. Both methods are presented in an inference system akin to a type system and build upon similar concepts. The worst-case analysis does not restrict the input rewrite system in any way, which facilitates integration in a general framework for resource analysis of programs. In contrast to that the best-case analysis is designed for first-order eagerly evaluated term rewrite systems and thus provides novel methods to fully automatically infer lower bounds. More precisely, we establish univariate amortised resource analyses based on the potential method which gives rise to polynomial bounds of the term rewrite system investigated. Due to the invocation of small-step semantics, the methods do not presuppose termination. This complements earlier work on automated amortised resource analysis. We have implemented the methods and provide ample evidence of their viability.

Contents

Part I

Worst-Case Upper Bounds for Term Rewrite Systems

1 Introduction

Being able to perform quantitative analyses of programs, e.g. inferring the runtime complexity in relation to the program input, is important for comparing algorithms, designing efficient programs or to identify performance bottlenecks in software [1]. Besides these obvious implications of program analysis there are other applications as well. For instance, in schedulability analyses of safety critical real-time computer systems worst-case execution times can be used to test whether a given set of tasks will meet the timing requirements of the application on a given target system and thus is schedulable or not [37]. This information can be used to ensure timely responses from interrupts which might be crucial and failure could lead to a disaster, sometimes including the loss of human life [10]. Similarly, the applications of the analysis of lower bounds include performance debugging, program optimisation and verification. But, in the literature one also finds the automation of parallelisation as application area, cf. [3, 9].

Most often we are concerned with time or space complexities of algorithms [1]. These complexities are measured in a quantity, called *size*, of input data, which is usually represented as an integer value. For example, the size for an algorithm which reverses an input list could be the number of elements in the list. Similarly, for a graph problem the number of nodes could specify it's size.

The time needed for the evaluation of an algorithm expressed as a function of it's size is called the *time complexity* [1]. The *asymptotic time complexity* is the limiting behavior of the complexity as the problem's size increases. To be more precise, if a problem of size n is processed by an algorithm in time cn^x for some constant c, then the algorithm is in the class $\mathsf{O}(n^x)$. Analogous (asymptotic) space complexity can be defined.

The *worst-case complexity* is the maximum complexity over all inputs of a given size [1]. In contrast, if for a given size the complexity is taken as the minimum complexity over all inputs of a that size, then the complexity is called *best-case complexity* [44]. Clearly, once the best-case and worst-case complexities of algorithms are known, the (asymptotic) time or space complexities are predetermined and algorithms can be compared to each other. This allows choosing of appropriate algorithms when designing applications. For instance, it might be crucial for security reasons to not provide location information of database entries to external users via the application interface. Thus, if a database request takes considerably more time once the needed information is located at the end of the database table as compared to the beginning, then the users could imply the location of the requested information according to the respond time. An algorithm with the same or similar (asymptotic) best-case and worst-case complexities could be chosen to prevent such an attack.

One way of finding and proving such properties is utilizing an amortised resource analysis originated by Sleator and Tarjan [40, 41]. It is a powerful method to assess the

overall complexity of a sequence of operations precisely. Amortised resource analysis has been established in the context of self-balancing data structures, which sometimes require costly operations that however balance out in the long run. For this observe that many data structures and programs require a sequence of operations rather than a single operation, e.g. when data is added in a balanced binary tree the algorithm has to perform a lookup, an insert, and might need to balance the tree. One of these operations might be slow, but the sequence of these operations still can be fast. When analysing such structures for the worst-case complexity the sum of the worst-case times of the individual operations may be unduly pessimistic as it ignores the correlation effects of the data structure. On the other hand, the sum of average-case analyses over the sequence may be simply false. Thus amortised analysis averages the running times of operations in a sequence over the sequence [41]. This is done by using a potential function Φ which maps the data structure to numbers, commonly elements of $\mathbb{N}$. Then the (worst-case) amortised cost of the i^{th} operation with output d_i and actual cost t_i is defined as $a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$. Therefore, the sum of amortised costs a_i over the sequence of operations is an upper bound of the actual resource consumption of the program [41].

Hofmann started with a different approach for program complexity analysis. They used annotated resource types which control the number of constructor symbols in recursive functions to ensure linear heap space of programs built upon the data structures of lists, binary trees and pairs [25]. Their technique was automated to a compilation process for a linear typed first-order functional programming language into malloc()-free C code. Later this technique of annotated resource types for a data structure-limited functional programming language was reused in [27]. They combined it with amortised resource analysis to count heap space allocations and deallocations.

Jost et al. [31] were able to gerneralise this approach to arbitrary recursive data structures for algorithms written in the simple, purely functional programming language Schopenhauer. Their experimental results show astonishing accurate predictions when compared to actual measured resource usage but the analysis method was still limited to linearly worst-case bounded input programs. The amortised worst-case complexity analysis was later lifted to a strict, higher-order, polymorphic, recursive Schopenhauer variant [30]. Finally in 2012 the work was extended to lazy functional programming by using a notion of lazy potential, which does not require defunctionalisation or other program transformations [38].

However, already in 2010 Hoffmann et al. extended the amortised resource analysis to use polynomial potential [24]. This was the first work to allow the automatic analysis to find univariate polynomial bounds. The analysis was based on a functional first-order programming language named Resource Aware ML (RaML). In [21] Hoffmann et al. presented an analysis which was capable of handling multivariate bounds. However, in their approach the input programs were again restricted to a first-order fragment of ML, featuring integers, lists, binary trees, and recursion. Nonetheless, later inspired by the methods introduced in [30] RaML and its amortised analysis were extended to incorporate higher-order constructs [23].

As the RaML prototype has grown into a highly sophisticated analysis tool for (higher-

order) functional programs, cf. [23], resource analysis tools for imperative programs like COSTA [2], CoFloCo [13] and LOOPUS [39] have integrated amortised reasoning.

Some of the amortised techniques were converted to term rewrite systems (TRS) which can be used as a intermediate compilation step for a complexity analysis from various types of programming languages, like object-oriented or functional ones. In general TRSs do not specify type structures and thus the method does not only work on typed programming languages, but is type independent. However, first the linear worst-case runtime analysis was converted to typed TRSs [28]. Typed TRSs extend TRSs by specifying data types declarations. Similarly to [21] the method allows recursive data types and is limited to TRS with linear worst-case runtime complexity. Then, Hofmann et al. adapted the techniques of the univariate analysis to enable polynomial worst-case complexity bounds of typed TRS [28], before lifting the analysis to multivariate bounds for (non-typed) TRS in [29]. The most important improvement to [31] and [21] is that arbitrary data types can be analysed with the demonstrated procedures, whereas the former ones rely on special constructs. Additionally, by removing the necessity of types the generality of TRSs could be regained.

While the study of upper bounds on the (worst-case) runtime complexity of programs is well-developed, the automated analysis of lower bounds for runtime complexity is somewhat underdeveloped. The literature mainly focuses on lower bound analysis for imperative and logic programs [3, 9]. In the setting of functional programming (or term rewriting for that matter) we are only aware of the analysis provided by Chang Ngo et. al. [33].

Rather than following the focus in [23, 33] on a particular programming language, we follow in this thesis the general approach of static program analysis, where peculiarities of specific programming languages are suitable abstracted to give way to more general constructions like *recurrence relations*, *cost relations*, *transition systems*, *term rewrite systems*, etc. Thus we seek a more general discussion of program analysis in (first-order) functional programming, where we choose term rewrite systems as suitable abstraction.

Therefore, in this thesis we study *worst-case* runtime complexity as well as *best-case* complexity for first-order eagerly evaluated functional programs and establish novel amortised resource methods to automatically infer runtime bounds. Taking inspirations from [28, 29], the amortised analyses are based on the potential method. They employ the standard (small-step) semantics of innermost rewriting and exploit a *footprint* relation in order to facilitate the extension to TRSs. For the latter, we suit a corresponding notion of Avanzini et al. [4] to our context. Due to the small-step semantics we immediately obtain analyses which do not presuppose termination. The incorporation of the footprint relations allows the immediate adaption of the proposed method to general rule-based languages.

Consider the rewrite system $\mathcal{R}_1$ in Figure 1.1 encoding a variant of an example by Okasaki [36, Section 5.2]. $\mathcal{R}_1$ encodes an efficient implementation of a queue in functional programming. A queue is represented as a pair of two lists $\mathsf{que}(f, r)$, encoding the initial part f and the reversal of the remainder r. The invariant of the algorithm is that the first list never becomes empty, which is achieved by reversing r if necessary. Should the invariant ever be violated, an exception (err_head or err_tail) is raised. To exemplify

$$
\begin{array}{rrcl}
1\colon & \mathsf{chk}(\mathsf{que}(\mathsf{nil}, r)) & \to & \mathsf{que}(\mathsf{rev}(r), \mathsf{nil}) \\
2\colon & \mathsf{chk}(\mathsf{que}(x \sharp xs, r)) & \to & \mathsf{que}(x \sharp xs, r) \\
3\colon & \mathsf{tl}(\mathsf{que}(x \sharp f, r)) & \to & \mathsf{chk}(\mathsf{que}(f, r)) \\
4\colon & \mathsf{snoc}(\mathsf{que}(f, r), x) & \to & \mathsf{chk}(\mathsf{que}(f, x \sharp r)) \\
5\colon & \mathsf{rev}'(x \sharp xs, ys) & \to & \mathsf{rev}'(xs, x \sharp ys) \\
6\colon & \mathsf{enq}(\mathsf{s}(n)) & \to & \mathsf{snoc}(\mathsf{enq}(n), n)
\end{array}
\qquad
\begin{array}{rrcl}
7\colon & \mathsf{enq}(0) & \to & \mathsf{que}(\mathsf{nil}, \mathsf{nil}) \\
8\colon & \mathsf{rev}'(\mathsf{nil}, ys) & \to & ys \\
9\colon & \mathsf{rev}(xs) & \to & \mathsf{rev}'(xs, \mathsf{nil}) \\
10\colon & \mathsf{hd}(\mathsf{que}(x \sharp f, r)) & \to & x \\
11\colon & \mathsf{hd}(\mathsf{que}(\mathsf{nil}, r)) & \to & \mathsf{err\_head} \\
12\colon & \mathsf{tl}(\mathsf{que}(\mathsf{nil}, r)) & \to & \mathsf{err\_tail}
\end{array}
$$

Figure 1.1: Queues in Rewriting

the physicist's method of amortised analysis [41] we assign to every queue $\mathsf{que}(f, r)$ the length of r as *potential*. Then the amortised cost for each operation is constant, as the costly reversal operation is only executed if the potential can pay for the operation, cf. [36]. Thus, based on an amortised analysis, we may deduce the optimal linear runtime complexity for $\mathcal{R}$.

Our tool is able to derive the linear worst-case and best-case runtime complexities for $\mathcal{R}_1$. The method derives signatures of the program's functions (this includes constructors). Each signature specifies the amortised cost for using the function, as well as how to calculate the potentials according to the result type annotation and parameter type annotations. E.g. the signature $\mathsf{rev}\colon [1] \xrightarrow{4} 0$ for the reverse function is inferred by the tool. Here the application of rev costs 4 units and the potential before the operation has to be evaluated with 1, whereas the potential after the operation is evaluated with 0. In case the input is a list with signatures $\sharp\colon [0 \times 1] \xrightarrow{1} 1$ and $\mathsf{nil}\colon [] \xrightarrow{0} 1$, then the potential of the input is recursively calculated by summing the costs of the function signatures. For instance, consider an input list of length two and the shape $\sharp\,(x, \sharp\,(y, \mathsf{nil}))$. Further, as the annotation of the parameter of rev is 1 the signature for the outermost function $\sharp$ fits. For this observe that the return type annotation of $\sharp$ is 1 as well. The potential $\Phi(t\colon c)$ of a term t with annotation c is found by recursively unfolding the data structure and adding the costs: $\Phi(\sharp\,(x, \sharp\,(y, \mathsf{nil}))\colon 1) = 1 + \Phi(x\colon 0) + \Phi(\sharp\,(y, \mathsf{nil})\colon 1) = 1+0+1+\Phi(y\colon 0)+\Phi(\mathsf{nil}\colon 1) = 1+0+1+0+0 = 2$, where the potential for the variables x and y are set to 0. Note that the return annotations of the underlying functions and the corresponding parameter annotations of the unfolded functions are alaways equal. Further only the costs, written above the arrow, are accumulated. In this example the length of the list equals it's potential. The output of rev will be $\sharp\,(y, \sharp\,(x, \mathsf{nil}))$ and the potential of the output structure with signatures $\sharp\colon [0 \times 0] \xrightarrow{0} 0$ and $\mathsf{nil}\colon [] \xrightarrow{0} 0$ is 0. Therefore, the amortised costs a_i of this operation is $a_i = t_i + \Phi(d_i) - \Phi(d_{i-1}) = 4 + 0 - 2 = 2$, where t_i is the actual cost and d_i the output of operation i. Even though the actual cost of the rev-operation is 4, in this sequence its amortised cost is 2. This makes sense as the potential has to pay for the seldom reversal of the list. The required potential is acquired in the previous operations. Therefore, the amortised analysis averages the costs of operations in a sequence over the sequence.

However, the difficulty is not to evaluate the signatures to potentials, but rather to find signatures which correctly balance the potential for a given problem. The hereafter presented methods are designed to infer constraint problems which when solved, provide the required signatures.

For the rewrite system $\mathcal{R}_1$ our tool automatically derives the following signatures for the worst-case complexity analysis:

$\mathsf{0}\colon [] \xrightarrow{0} 15$	$\mathsf{s}\colon [15] \xrightarrow{15} 15$	$\mathsf{err\_head}\colon [] \xrightarrow{0} 6$	$\mathsf{err\_tail}\colon [] \xrightarrow{0} 7$
$\sharp\colon [0 \times 0] \xrightarrow{0} 0$	$\sharp\colon [0 \times 1] \xrightarrow{1} 1$	$\sharp\colon [0 \times 7] \xrightarrow{7} 7$	
$\mathsf{nil}\colon [] \xrightarrow{0} 0$	$\mathsf{nil}\colon [] \xrightarrow{0} 1$	$\mathsf{nil}\colon [] \xrightarrow{0} 7$	$\mathsf{nil}\colon [] \xrightarrow{0} 15$
$\mathsf{que}\colon [0 \times 7] \xrightarrow{7} 7$	$\mathsf{que}\colon [0 \times 8] \xrightarrow{8} 8$	$\mathsf{que}\colon [0 \times 11] \xrightarrow{11} 11$	
$\mathsf{chk}\colon [7] \xrightarrow{5} 7$	$\mathsf{tl}\colon [11] \xrightarrow{3} 1$	$\mathsf{snoc}\colon [7 \times 0] \xrightarrow{14} 7$	$\mathsf{rev}\colon [1] \xrightarrow{4} 0$
$\mathsf{rev}'\colon [1 \times 0] \xrightarrow{2} 0$	$\mathsf{enq}\colon [15] \xrightarrow{12} 7$	$\mathsf{hd}\colon [11] \xrightarrow{9} 0$	

Note that we allow multiple signature declarations for constructor symbols which are presented in the upper half. The vector length of the annotations used in the signatures then give rise to the asymptotic complexity of the TRS. In this case only scalar annotations (vectors of length $\mathbf{1}$) are used, in comparisons to e.g. vectors of length two like in a possible signature $\mathsf{s}\colon [(1,1)] \xrightarrow{1} (0,1)$. Thus, as we will see, we can infer linear worst-case complexity, that is $\mathsf{O}(n^{\mathbf{1}})$, for the TRS $\mathcal{R}_1$.

Worst-Case Analysis

The most significant extension of the worst-case analysis is the extension to standard TRSs. TRSs form a universal model of computation that underlies much of declarative programming. In the context of functional programming, TRSs form a natural abstraction of strictly typed programming languages like RaML, but natively form foundations of non-strict languages and non-typed languages as well.

Our interest in an amortised analysis for TRSs is motivated by the use of TRSs as abstract program representation within our uniform resource analyse tool T$_\mathsf{C}$T [6]. Incorporating a transformational approach the latter provides a state-of-the-art tool for the resource analysis of pure OCaml programs, but more generally allows the analysis of general programs. In this spirit we aim at an amortised resource for TRSs in its standard form: untyped, not necessarily left-linear, confluent, or constructor-based. Technically, the main contributions of the worst-case analysis presented in the first part of the thesis are as follows.

- Employing the standard rewriting semantics in the context of amortised resource analysis. This standardises the results and simplifies the presentations contrasted

to related results on amortised analysis of TRSs cf. [28]. We emphasise that our analysis does not presuppose termination.

- We overcome earlier restrictions to sorted, completely defined, orthogonal and constructor TRSs, that is, we establish an amortised analysis for standard first-order rewrite system, that is, the only restrictions required are the standard restrictions that (i) the left-hand side of a rule must not be a variable and (ii) no extra variables are introduced in rules.

- The analysis is lifted to relative rewriting, that is, the runtime complexity of a relative TRS $\mathcal{R}/\mathcal{S}$ is measured by the number of rule applications from $\mathcal{R}$, only. This extension is mainly of practical relevance, as required to obtain an automation of significant strength.

- Finally, the analysis has been implemented and integrated into T$_\mathsf{C}$T. We have assessed the viability of the method in context of the TPDB as well as on an independent benchmark.

This thesis is split into two parts. The first part concentrates on the worst-case complexity analysis, whereas the second part on the best-case complexity. Nonetheless the best-case analysis heavily adapts ideas and concept from the worst-case analysis. The rest of this part of the thesis is structured as follows. In the next section we cover basics. In Section 3 we introduce the inference system and prove soundness. In Section 4 we detail the implementation of the method and remark on challenges posed by automation. Section 5 provides a full numerical example of the worst-case analysis. Section 6 provides the experimental assessment of the method. Then in part two we shortly motivate the best-case analysis in Section 7 before delineate a workable definition of best-case complexity in the context of (non-deterministic) TRSs and introduce the definition of lower bounds formally. The best-case amortised analysis and the inference system embodying this analysis is given in Section 9. In this section we state soundness of the method (Theorem 9.11). In Section 10 we introduce refinements, and detail the implementation and remark on challenges posed by automation. Before providing the experimental assessment in Section 12 we again present a numerical example of the method in Section 11. Finally we conclude in Chapter 13, where we also sketch future work.

2 Preliminaries and Runtime Complexity

We assume familiarity with term rewriting [7, 42] but briefly review basic concepts and notations.

Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature, such that $\mathcal{F}$ contains at least one constant. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $\mathcal{V}\mathsf{ar}(t)$ to denote the set of variables occurring in term t.

We suppose $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{C}$ denotes a finite, non-empty set of *constructor symbols*, $\mathcal{D}$ is a finite set of *defined function symbols*, and $\uplus$ denotes disjoint union. Defined function symbols are sometimes referred to as *operators*. A term t is *linear* if every variable in t occurs only once. A term t' is the *linearisation* of a non-linear term t if the variables in t are renamed apart such that t' becomes linear. The notion generalises to sequences of terms.

A term $t = f(t_1, \ldots, t_k)$ is called *basic*, if f is defined, and all $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. If $t \in \mathcal{T}(\mathcal{C}, \varnothing)$ then t is called a *ground constructor term* or a *value*. The set of values is denoted $\mathcal{V}\mathsf{al}$. The n-fold iteration of a unary function symbol f to a term t is denoted as $f^n(t)$. This notion generalises to non-unary function symbols as in $g(c, \cdot)^n(t)$, where $\cdot$ denotes the iteration point. E.g. $g(c, \cdot)^2(t) = g(c, g(c, t))$.

The size of t, that is, the number of symbols in t, is denoted $|t|$. Furthermore, the depth of a term t, denoted as $\mathsf{depth}(t)$, is defined as the height of the tree of positions of t. For example $\mathsf{depth}(\mathsf{f}(\mathsf{a}, \mathsf{c})) = 2$.

In the following, function symbols are denoted as f, g, h; variables are denoted as x, y, z; terms are denoted by s, t, u; values are denoted by v (possibly extended by subscripts). Sequences of terms $t_1, \ldots, t_n$ are denotes as vectors $\vec{t}$.

A *substitution* σ is a mapping from variables to terms. Substitutions are denoted as sets of assignments: $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We write $\mathsf{dom}(\sigma)$ ($\mathsf{rg}(\sigma)$) to denote the domain (range) of σ.

Let $\to \subseteq S \times S$ be a binary relation. We denote by $\to^+$ the transitive and by $\to^*$ the transitive and reflexive closure of $\to$. By $\to^n$ we denote the n-fold application of $\to$. If t is in normal form with respect to $\to$, we write $s \to^! t$.

We say that $\to$ is *well-founded* or *terminating* if there is no infinite sequence $s_0 \to s_1 \to \ldots$. It is *finitely branching* if the set $\{t \mid s \to t\}$ is finite for each $s \in S$.

For two binary relations $\to_A$ and $\to_B$, the relation of $\to_A$ *relative* to $\to_B$ is defined by $\to_A / \to_B := \to_B^* \cdot \to_A \cdot \to_B^*$

A *rewrite rule* is a pair $l \to r$ of terms, such that (i) the root symbol of l is defined, and (ii) $\mathcal{V}\mathsf{ar}(l) \supseteq \mathcal{V}\mathsf{ar}(r)$. A *term rewrite system* (TRS) over $\mathcal{F}$ is a finite set of rewrite rules.

Definition 2.1. Let $\mathcal{R}$ be a TRS. We define the *innermost rewrite relation* as follows

$$\frac{f(l_1, \dots, l_n) \to r \in \mathcal{R},\ \sigma \colon \mathcal{V} \to \mathcal{V}\mathsf{al}}{f(l_1\sigma, \dots, l_n\sigma) \xrightarrow{\mathrm{i}}_{\mathcal{R}} r\sigma} \qquad \frac{s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t}{f(\dots, s, \dots) \xrightarrow{\mathrm{i}}_{\mathcal{R}} f(\dots, t, \dots)}$$

A TRS is *left-linear* if all rules are left-linear, it is *non-overlapping* if there are no critical pairs, that is, no ambiguity exists in applying rules. A TRS is *orthogonal* if it is left-linear and non-overlapping. A TRS is *completely defined* if all ground normal-forms are values[1]. Note that an orthogonal TRS is confluent. A TRS is *constructor* if all arguments of left-hand sides are basic.

Observe that TRSs need not be constructor systems, that is, arguments of left-hand sides of rules may contain defined symbols. Such function symbols are called *constructor-like*, as below they will be sometimes subject to similar restrictions as constructor symbols.

The set of normal forms of a TRS $\mathcal{R}$ is denoted as $\mathsf{NF}(\mathcal{R})$, or NF for short. We call a substitution σ *normalised with respect to* $\mathcal{R}$ if all terms in the range of σ are ground normal forms of $\mathcal{R}$. Typically $\mathcal{R}$ is clear from context, so we simply speak of a *normalised* substitution.

In the sequel we are concerned with *innermost* rewriting, that is, an eager evaluation strategy. Furthermore, we consider relative rewriting.

The *innermost rewrite relation* $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is defined on terms as follows: $s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t$ if there exists a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ such that $s = C[l\sigma]$, $t = C[r\sigma]$, and all proper subterms of $l\sigma$ are normal forms of $\mathcal{R}$.

In order to generalise the innermost rewriting relation to relative rewriting, we introduce the slightly technical construction of the *restricted* rewrite relation [43]. The *restricted rewrite relation* $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ is the restriction of $\to_{\mathcal{R}}$ where all arguments of the redex are in normal form with respect to the TRS $\mathcal{Q}$. We define the *innermost rewrite relation*, dubbed $\xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}}$, of a relative TRS $\mathcal{R}/\mathcal{S}$ as follows.

$$\xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}} := \xrightarrow{\mathcal{R}\cup\mathcal{S}}{}^{*}_{\mathcal{S}} \cdot \xrightarrow{\mathcal{R}\cup\mathcal{S}}_{\mathcal{R}} \cdot \xrightarrow{\mathcal{R}\cup\mathcal{S}}{}^{*}_{\mathcal{S}} .$$

Observe that $\xrightarrow{\mathrm{i}}_{\mathcal{R}} = \xrightarrow{\mathrm{i}}_{\mathcal{R}/\varnothing}$ holds.

Let s and t be terms, such that t is in normal-form. Then a *derivation* $D \colon s \to^*_{\mathcal{R}} t$ with respect to a TRS $\mathcal{R}$ is a finite sequence of rewrite steps.

Definition 2.2. The *derivation height* of a term s with respect to a well-founded, finitely branching relation $\to$ is defined as

$$\mathsf{dh}(s, \to) = \max\{n \mid \exists t\ s \to^n t\}.$$

[1] In the literature, often a more general definition of a *completely defined* TRS is given: any normal-form is a constructor term. However, in the presence of at least one constant the provided simpler definition is equivalent in our context.

Runtime Complexity

Definition 2.3. We define the *innermost runtime complexity* (with respect to $\mathcal{R}/\mathcal{S}$): $\mathsf{rc}_{\mathcal{R}}(n) := \max\{\mathsf{dh}(t, \xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}}) \mid t \text{ is basic and } |t| \leqslant n\}$.

Intuitively the innermost runtime complexity w.r.t. $\mathcal{R}/\mathcal{S}$ counts the maximal number of eager evaluation steps in $\mathcal{R}$ in a derivation over $\mathcal{R} \cup \mathcal{S}$. In the definition, we tacitly assume that $\xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}}$ is terminating and finitely branching.

For the rest of this part the relative TRS $\mathcal{R}/\mathcal{S}$ and its signature $\mathcal{F}$ are fixed. In the sequel of the thesis, substitutions are assumed to be normalised with respect to $\mathcal{R} \cup \mathcal{S}$.

3 Worst-Case Amortised Analysis

In this section we establish a novel amortised resource analysis for TRSs, which is based on the potential method and coached in an inference system. Firstly, we annotate the (untyped) signature by the prospective resource usage (Definition 3.2). Secondly, we define a suitable inference system, akin to a type system. Based on this inference system we delineate a class of *resource bounded* TRSs (Definition 3.10) for which we deduce polynomial bounds on the innermost runtime complexity for a suitably chosen class of annotations, cf. Theorem 3.16.

Definition 3.1. A *resource annotation* $\vec{p}$ is a vector $\vec{p} = (p_1, \ldots, p_k)$ over non-negative rational numbers, typically natural numbers. The vector $\vec{p}$ is also simply called an *annotation*.

Resource annotations are denoted by $\vec{p}$, $\vec{q}$, $\vec{u}$, $\vec{v}$, ..., possibly extended by subscripts and we write $\mathcal{A}$ for the set of such annotations. For resource annotations (p) of length 1 we write p. A resource annotation does not change its meaning if zeroes are appended at the end, so, conceptually, we can identify $()$ with (0) and also with 0. If $\vec{p} = (p_1, \ldots, p_k)$ we set $k := |\vec{p}|$ and $\max \vec{p} := \max\{p_i \mid i = 1, \ldots, k\}$. We define the notations $\vec{p} \leqslant \vec{q}$ and $\vec{p} + \vec{q}$ and $\lambda\vec{p}$ for $\lambda \geqslant 0$ component-wise, filling up with 0s if needed. So, for example $(1,2) \leqslant (1,2,3)$ and $(1,2) + (3,4,5) = (4,6,5)$. Furthermore, we recall the additive shift [24] given by $\lhd(p_1, \ldots, p_k) = (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$. We also define the interleaving $\vec{p} \,|||\, \vec{q}$ by $(p_1, q_1, p_2, q_2, \ldots, p_k, q_k)$ where, as before the shorter of the two vectors is padded with 0s. Finally, we use the notation $\Diamond\vec{p} = p_1$ for the first entry of an annotation vector.

Definition 3.2. Let f be a function symbol of arity n. We annotate the arguments and results of f by *resource annotations*. A (resource) annotation for f, decorated with $k \in \mathbb{Q}^+$, is denoted as $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$. The set of annotations is denoted $\mathcal{F}_{\mathsf{pol}}$.

We lift signatures $\mathcal{F}$ to *annotated signatures* $\mathcal{F} \colon \mathcal{C} \cup \mathcal{D} \to (\mathcal{P}(\mathcal{F}_{\mathsf{pol}}) \setminus \varnothing)$ by mapping a function symbol to a non-empty set of resource annotations. Hence for any function symbol we allow multiple types. In the context of operators this is also referred to as *resource polymorphism*. The inference system, presented below, mimics a type system, where the provided annotations play the role of types. If the annotation of a constructor or constructor-like symbol f results in $\vec{q}$, there must only be exactly one declaration of the form $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$ in $\mathcal{F}(f)$, that is, the annotation has to be *unique*. Moreover, annotations for constructor and constructor-like symbols f must satisfy the *superposition principle*: If f admits the annotations $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$ and $[\vec{p'_1} \times \cdots \times \vec{p'_n}] \xrightarrow{k'} \vec{q'}$ then it also has the annotations $[\lambda\vec{p_1} \times \cdots \times \lambda\vec{p_n}] \xrightarrow{\lambda k} \lambda\vec{q}$ ($\lambda \in \mathbb{Q}^+$, $\lambda \geqslant 0$) and $[\vec{p_1} + \vec{p'_1} \times \cdots \times \vec{p_n} + \vec{p'_n}] \xrightarrow{k+k'} \vec{q} + \vec{q'}$.

Example 3.3. Consider the sets $\mathcal{D} = \{\mathsf{enq}, \mathsf{rev}, \mathsf{rev}', \mathsf{snoc}, \mathsf{chk}, \mathsf{hd}, \mathsf{tl}\}$ and $\mathcal{C} = \{\mathsf{nil}, \sharp, \mathsf{que}, \mathsf{0}, \mathsf{s}\}$, which together make up the signature $\mathcal{F}$ of the motivating example $\mathcal{R}_1$ in Figure 3.1. Annotations of the constructors nil and $\sharp$ would for example be as follows. $\mathcal{F}(\mathsf{nil}) = \{[] \xrightarrow{0} k \mid k \geqslant 0\}$ and $\mathcal{F}(\sharp) = \{[0 \times k] \xrightarrow{k} k \mid k \geqslant 0\}$. These annotations are unique and fulfill the superposition principle. □

Note that, in view of superposition and uniqueness, the annotations of a given constructor or constructor-like symbol are uniquely determined once we fix the resource annotations for result annotations of the form $(0, \dots, 0, 1)$ (remember the implicit filling up with 0s). An annotated signature $\mathcal{F}$ is simply called signature, where we sometimes write $f\colon [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q}$ instead of $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)$.

The next definition introduces the notion of the potential of a normal form. For rules $f(l_1, \dots, l_n) \to r$ in non-constructor TRSs the left-hand side $f(l_1, \dots, l_n)$ need not necessarily be basic terms. However, the arguments l_i are deconstructed in the rule (app) that we will see in Figure 3.1. This deconstruction may free potential, which needs to be well-defined. This makes it necessary to treat defined function symbols in l_i similar to constructors in the inference system (see Definition 3.7).

Definition 3.4. Let $v = f(v_1, \dots, v_n)$ be a normal form and let $\vec{q}$ be a resource annotation. We define the *potential* of v with respect to $\vec{q}$, written $\Phi(v\colon \vec{q})$ by cases. First suppose v contains only constructors or constructor-like symbols. Then the potential is defined recursively as

$$\Phi(v\colon \vec{q}) := k + \Phi(v_1\colon \vec{p}_1) + \cdots + \Phi(v_n\colon \vec{p}_n)\ ,$$

where $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)$. Otherwise, we set $\Phi(v\colon \vec{q}) := 0$.

The *sharing relation* $\curlyvee(\vec{p} \,|\, \vec{p_1}, \vec{p_2})$ holds if $\vec{p_1} + \vec{p_2} = \vec{p}$.

Lemma 3.5. *Let v be a a normal form. If $\curlyvee(\vec{p} \,|\, \vec{p_1}, \vec{p_2})$ then $\Phi(v\colon \vec{p}) = \Phi(v\colon \vec{p_1}) + \Phi(v\colon \vec{p_2})$. Furthermore, if $\vec{p} \leqslant \vec{q}$ then $\Phi(v\colon \vec{p}) \leqslant \Phi(v\colon \vec{q})$.*

Proof. We distinguish two cases. Either v is only build from constructor symbols or constructor-like symbols. Then by superposition together with uniqueness the additivity property propagates to the argument types. For example, if we have the annotations $\mathsf{s} : [6] \xrightarrow{3} 12$, $\mathsf{s} : [10] \xrightarrow{5} 20$, and $\mathsf{s} : [x] \xrightarrow{8} y$ then we can conclude $x = 16$, $y = 32$, for this annotation must be present by superposition and there can only be one by uniqueness. Otherwise v contains at least one $f \in \mathcal{D}$, but f does not occur as argument of a left-hand side in $\mathcal{R}$. By definition $\Phi(v\colon \vec{q}) = 0$. Thus the lemma holds trivially. The second claim follows from the first one and non-negativity of potentials. □

A *(variable) context* is a partial mapping from variables $\mathcal{V}$ to annotations. Contexts are denoted by upper-case Greek letters and depicted as sequences of pairs $x\colon \vec{q}$ of variables and annotations, where $x\colon \vec{q}$ in a variable context means that the resource $\vec{q}$ can be distributed over all occurrences of the variable x in the term.

$$\frac{f \in \mathcal{C} \cup \mathcal{D} \quad [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)}{x_1 \colon \vec{p}_1, \ldots, x_n \colon \vec{p}_n \mid\!\!\overset{k}{\vdash} f(x_1, \ldots, x_n) \colon \vec{q}}\ (\mathsf{app}) \qquad \frac{\Gamma \mid\!\!\overset{k}{\vdash} t \colon \vec{q} \quad k' \geqslant k}{\Gamma \mid\!\!\overset{k'}{\vdash} t \colon \vec{q}}\ (\mathsf{w}_1)$$

$$\frac{\begin{array}{ll} \text{all } x_i \text{ are fresh} & k = \sum_{i=0}^{n} k_i \\ x_1 \colon \vec{p}_1, \ldots, x_n \colon \vec{p}_n \mid\!\!\overset{k_0}{\vdash} f(x_1, \ldots, x_n) \colon \vec{q} & \Gamma_1 \mid\!\!\overset{k_1}{\vdash} t_1 \colon \vec{p}_1 \ \cdots \ \Gamma_n \mid\!\!\overset{k_n}{\vdash} t_n \colon \vec{p}_n \end{array}}{\Gamma_1, \ldots, \Gamma_n \mid\!\!\overset{k}{\vdash} f(t_1, \ldots, t_n) \colon \vec{q}}\ (\mathsf{comp})$$

$$\frac{\Gamma \mid\!\!\overset{k}{\vdash} t \colon \vec{q}}{\Gamma, x \colon \vec{p} \mid\!\!\overset{k}{\vdash} t \colon \vec{q}}\ (\mathsf{w}_4) \qquad \frac{\Gamma, x \colon \vec{r}, y \colon \vec{s} \mid\!\!\overset{k}{\vdash} t[x, y] \colon \vec{q} \quad \curlyvee(\vec{p} \,|\, \vec{r}, \vec{s}) \quad x,\, y \text{ are fresh}}{\Gamma, z \colon \vec{p} \mid\!\!\overset{k}{\vdash} t[z, z] \colon \vec{q}}\ (\mathsf{share})$$

$$\frac{\Gamma, x \colon \vec{r} \mid\!\!\overset{k}{\vdash} t \colon \vec{q} \quad \vec{p} \geqslant \vec{r}}{\Gamma, x \colon \vec{p} \mid\!\!\overset{k}{\vdash} t \colon \vec{q}}\ (\mathsf{w}_2) \qquad \frac{}{x \colon \vec{q} \mid\!\!\overset{0}{\vdash} x \colon \vec{q}}\ (\mathsf{var}) \qquad \frac{\Gamma \mid\!\!\overset{k}{\vdash} t \colon \vec{s} \quad \vec{s} \geqslant \vec{q}}{\Gamma \mid\!\!\overset{k}{\vdash} t \colon \vec{q}}\ (\mathsf{w}_3)$$

Figure 3.1: Inference System for Term Rewrite Systems.

Definition 3.6. Our potential based amortised analysis is coached in an inference system whose rules are given in Figure 3.1. Let t be a term and $\vec{q}$ a resource annotation. The inference system derives judgements of the form $\Gamma \mid\!\!\overset{k}{\vdash} t \colon \vec{q}$, where Γ is a variable context and $k \in \mathbb{Q}^+$ denotes the amortised costs at least required to evaluate t.

Furthermore, we define a subset of the inference rules, free of weakening rules, dubbed the *footprint* of the judgement, denoted as $\Gamma \mid\!\!\overset{k}{\underset{\mathsf{fp}}{\vdash}} t \colon \vec{q}$. Thus for the footprint we only consider the inference rules (app), (comp), (share), and (var).

Occasionally we omit the amortised costs from both judgements using the notations $\Gamma \mid\!\!\vdash t \colon \vec{q}$ and $\Gamma \mid\!\!\underset{\mathsf{fp}}{\vdash} t \colon \vec{q}$.

To ease the presentation we have omitted certain conditions, like the pairwise disjointedness of $\Gamma_1, \ldots, \Gamma_n$ in the rule (comp), that make the inference rules deterministic. However, the implementation (see Section 4) is deterministic, which removes redundancy in constraint building and thus improves performance. A substitution is called *consistent with* Γ if for all $x \in \mathsf{dom}(\sigma)$ if $\Gamma \mid\!\!\vdash x \colon \vec{q}$, then $\Gamma \mid\!\!\vdash x\sigma \colon \vec{q}$. Recall that substitutions are assumed to be normalised. Let Γ be a context and let σ be a substitution consistent with Γ. Then $\Phi(\sigma \colon \Gamma) := \sum_{x \in \mathsf{dom}(\Gamma)} \Phi(x\sigma \colon \Gamma(x))$.

Definition 3.7. Let $f(l_1, \ldots, l_n) \to r$, $n \geqslant 1$, be a rule in the TRS $\mathcal{R}/\mathcal{S}$. Further suppose $f \colon [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q}$ is a resource annotation for f and let $V := \{y_1, \ldots, y_m\}$ denote the set of variables in the left-hand side of the rule. The potential *freed* by the rule is a pair consisting of a variable context $y_1 \colon \vec{r}_1, \ldots, y_m \colon \vec{r}_m$ and an amortised cost ℓ, defined as follows:

- The sequence $l'_1, \ldots, l'_n$ is a linearisation of $l_1, \ldots, l_n$. Set $Z := \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(l'_i)$ and let $Z = \{z_1, \ldots, z_{m'}\}$, where $m' \geqslant m$.

- There exist annotations $\vec{s}_1, \dots, \vec{s}_{m'}$ such that for all i there exist costs ℓ_i such that $z_1 \colon \vec{s}_1, \dots, z_{m'} \colon \vec{s}_{m'} \mid\!\!\frac{\ell_i}{\mathsf{fp}}\; l'_i \colon \vec{p}_i$.
- Let $y_j \in V$ and let $\{z_{j_1}, \dots, z_{j_o}\} \subseteq Z$ be all renamings of y_j. Define annotations $\vec{r}_j := \vec{s}_{j_1} + \cdots + \vec{s}_{j_o}$.
- Finally, $\ell := \sum_{i=1}^{n} \ell_i$.

Example 3.8. Consider the rule $\mathsf{enq}(\mathsf{s}(n)) \to \mathsf{snoc}(\mathsf{enq}(n), n)$ in the running example, together with the annotated signature $\mathsf{enq} \colon [15] \xrightarrow{12} 7$. The left-hand side contains the subterm $\mathsf{s}(n)$. Using the generic annotation $\mathsf{s} \colon [k] \xrightarrow{k} k$, the footprint $n \colon k \mid\!\!\frac{k}{\mathsf{fp}}\; \mathsf{s}(n) \colon k$ is derivable for any $k \geqslant 0$. Thus, in particular the rule frees the context $n \colon 15$ and cost 15. □

Lemma 3.9. *Let $f(l_1, \dots, l_n) \to r \in \mathcal{R}/\mathcal{S}$ and let $c \colon [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{0} \vec{q}$ denote a fresh, cost-free constructor. Let $y_1 \colon \vec{r}_1, \dots, y_m \colon \vec{r}_m$ and ℓ be freed by the rule. We obtain:*

$$y_1 \colon \vec{r}_1, \dots, y_m \colon \vec{r}_m \mid\!\!\frac{\ell}{\mathsf{fp}}\; c(l_1, \dots, l_n) \colon \vec{q}.$$

Proof. By assumption there exists a linearisation $l'_1, \dots, l'_n$ of the arguments of the left-hand side of the rule. By definition no variable occurs twice in the sequence $l'_1, \dots, l'_n$. Furthermore, for every $i = 1, \dots, n$, the following judgement is derivable:

$$z_1 \colon \vec{s}_1, \dots, z_{m'} \colon \vec{s}_m \mid\!\!\frac{\ell_i}{\mathsf{fp}}\; l'_i \colon \vec{p}_i \;, \tag{3.1}$$

where $\ell = \sum_{i=1}^{n} \ell_i$. Observe that the definition of the annotations $\vec{r}_j$ embodies a repeated application of the sharing rule (share). Thus in order to prove the lemma, it suffices to derive $z_1 \colon \vec{s}_1, \dots, z_{m'} \colon \vec{s}_{m'} \mid\!\!\frac{\ell}{\mathsf{fp}}\; c(l'_1, \dots, l'_n) \colon \vec{q}$. However, to derive the latter a single composition rule, together with the assumed derivations (3.1) suffices. □

Based on Definition 3.7 we can now succinctly define resource boundedness of a TRS. The definition constitutes a non-trivial generalisation of Definition 11 in [28]. First the input TRS need no longer be sorted. Second the restriction on constructor TRSs has been dropped and finally, the definition has been extended to handle relative rewriting.

Definition 3.10. Let $\mathcal{R}/\mathcal{S}$ be a relative TRS, let $\mathcal{F}$ be a signature and let $f \in \mathcal{F}$. An annotation $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)$ is called *resource bounded* if for any rule $f(l_1, \dots, l_n) \to r \in \mathcal{R} \cup \mathcal{S}$, we have

$$y_1 \colon \vec{r}_1, \dots, y_l \colon \vec{r}_l \mid\!\!\frac{k+\ell-K^{\mathrm{rule}}}{}\; r \colon \vec{q}\,,$$

where $y_1 \colon \vec{r}_1, \dots, y_l \colon \vec{r}_l$ and ℓ are freed by the rule if $n \geqslant 1$ and $\ell = 0$ otherwise. Here, the cost K^{rule} for the application of the rule is defined as follows:

$$K^{\mathrm{rule}} := \begin{cases} 0 & \text{if } f(l_1, \dots, l_n) \to r \in \mathcal{S} \\ 1 & \text{if } f(l_1, \dots, l_n) \to r \in \mathcal{R} \end{cases}$$

We call an annotation *cost-free* resource bounded if the cost K^{rule} is always set to zero.

A function symbol f is called *(cost-free) resource bounded* if any resource annotation in $\mathcal{F}(f)$ is (cost-free) resource bounded. Finally, $\mathcal{R}/\mathcal{S}$ is called *resource bounded*, or simply *bounded* if any $f \in \mathcal{F}$ is resource bounded. Observe that boundedness of $\mathcal{R}/\mathcal{S}$ entails that the application of rules in the strict part $\mathcal{R}$ is counted, while the weak part $\mathcal{S}$ is not counted.

In a nutshell, the method works as follows: Suppose the judgement $\Gamma \mid\!\!\overset{k'}{\vdash} t\colon \vec{q}$ is derivable and suppose σ is consistent with Γ. The constant k' is an upper-bound to the amortised cost required for reducing t to normal form. Below we will prove that the derivation height of $t\sigma$ (with respect to innermost rewriting) is bounded by the difference in the potential before and after the evaluation plus k'. Thus if the sum of the potentials of the arguments of $t\sigma$ is in $\mathsf{O}(n^k)$, where n is the size of the arguments and k the maximum length of the resource annotations, then the innermost runtime complexity of $\mathcal{R}/\mathcal{S}$ lies in $\mathsf{O}(n^k)$.

More precisely consider the comp rule. First note that this rule is only applicable if $f(t_1, \ldots, t_n)$ is linear, which can always be obtained by the use of the sharing rule. Now the rule embodies that the amortised costs k' required to evaluate $t\sigma$ can be split into those costs k'_i $(i \geqslant 1)$ required for the normalisation of the arguments and the cost k'_0 of the evaluation of the operator f. Furthermore the potential provided in the context $\Gamma_1, \ldots, \Gamma_n$ is suitably distributed. Finally the potential which remains after the evaluation of the arguments is made available for the evaluation of the operator f.

Before we proceed with the formal proof of this intuition, we exemplify the method on the running example.

Example 3.11 (continued from Example 3.3)**.** TCT derives the following annotations for the operators in the running example.

$$\begin{array}{lll} \mathsf{enq}\colon [15] \xrightarrow{12} 7 & \mathsf{rev}\ \colon [1] \xrightarrow{4} 0 & \mathsf{rev}'\colon [1 \times 0] \xrightarrow{2} 0 \\ \mathsf{snoc}\colon [7 \times 0] \xrightarrow{14} 7 & \mathsf{hd}\ \colon [11] \xrightarrow{9} 0 & \mathsf{tl}\colon [11] \xrightarrow{3} 1 \end{array}$$

□

We consider resource boundedness of $\mathcal{R}_1$ with respect to the given (monomorphic) annotated signatures of Example 3.11. For simplicity we restrict to boundedness of enq. We leave it to the reader to check the other cases. In addition to the annotations for constructor symbols (cf. Example 3.3) we can always assume the presence of zero-cost annotations, e.g. $\sharp\colon [0 \times 0] \xrightarrow{0} 0$. Observe that Rule 6 frees the context $n\colon 15$ and cost 15. Thus, we obtain the following derivation.

$$\dfrac{\dfrac{\dfrac{\mathsf{snoc}\colon [7 \times 0] \xrightarrow{14} 7}{q\colon 7, m\colon 0 \mid\!\!\overset{14}{\vdash} \mathsf{snoc}(q,m)\colon 7}\ (\mathsf{app}) \quad \dfrac{}{n_2\colon 0 \mid\!\!\overset{0}{\vdash} n_2\colon 0}\ (\mathsf{var}) \quad \dfrac{\mathsf{enq}\colon [15] \xrightarrow{12} 7}{n_1\colon 15 \mid\!\!\overset{12}{\vdash} \mathsf{enq}(n_1)\colon 7}\ (\mathsf{app})}{n_1\colon 15, n_2\colon 0 \mid\!\!\overset{26}{\vdash} \mathsf{snoc}(\mathsf{enq}(n_1), n_2)\colon 7}\ (\mathsf{comp})}{n\colon 15 \mid\!\!\overset{26}{\vdash} \mathsf{snoc}(\mathsf{enq}(n), n)\colon 7}\ (\mathsf{share})$$

In comparison to [28, Example 13], where the annotations were found manually, we note that the use of the interleaving operation [28] has been avoided. This is due to the more general class of annotations considered in our prototype implementation (see Section 4).

The footprint relation forms a restriction of the judgement $\vdash$ without the use of weakening. Hence the footprint allows a precise control of the resources stored in the substitutions, as indicated by the next lemma.

Lemma 3.12. *Let t be a normal form w.r.t. $\mathcal{R}$, where t consists of constructor or constructor-like symbols only. If $\Gamma \mid\!\!\frac{k}{\mathsf{fp}}\, t\colon \vec{q}$, then $\Phi(t\sigma\colon \vec{q}) = \Phi(\sigma\colon \Gamma) + k$.*

Proof. Let Π denote the derivation of the footprint $\Gamma \mid\!\!\frac{k}{\mathsf{fp}}\, t\colon \vec{q}$ and let $t = f(t_1, \dots, t_n)$. We proceed by induction on Π. We restrict our attention to the cases where Π amounts to a rule application or ends in a comp rule. The other cases are treated similarly.

Suppose Π has the following form, so that $t = f(x_1, \dots, x_n)$.

$$\frac{f \in \mathcal{C} \cup \mathcal{D} \quad [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)}{x_1\colon \vec{p}_1, \dots, x_n\colon \vec{p}_n \mid\!\!\frac{k}{\vdash}\, f(x_1, \dots, x_n)\colon \vec{q}} \quad .$$

By assumption the annotation $f\colon [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q}$ is unique. Hence, we obtain:

$$\Phi(f(x_1, \dots, x_n)\sigma\colon \vec{q}) = k + \Phi(x_1\sigma\colon \vec{p}_1) + \cdots + \Phi(x_n\sigma\colon \vec{p}_n) = \Phi(\sigma\colon \Gamma) + k \;,$$

from which the claim follows.

Suppose Π ends in a comp rule and thus has the following form.

$$\frac{\overbrace{x_1\colon \vec{p}_1, \dots, x_n\colon \vec{p}_n}^{=:\Delta} \mid\!\!\frac{k_0}{\vdash}\, f(x_1, \dots, x_n)\colon \vec{q} \quad \Gamma_i \mid\!\!\frac{k_i}{\vdash}\, t_i\colon \vec{p}_i \quad \text{for all } i = 1, \dots, n}{\Gamma_1, \dots, \Gamma_n \mid\!\!\frac{k}{\vdash}\, f(t_1, \dots, t_n)\colon \vec{q}} \quad .$$

Wlog. t is linear. By induction hypothesis we have for all $i = 1, \dots, n$: $\Phi(t_i\sigma\colon \vec{p}_i) = \Phi(\sigma\colon \Gamma_i) + k_i$. We set $\rho := \{x_i \mapsto t_i\sigma \mid i = 1, \dots, n\}$. Again by induction hypothesis we conclude that $\Phi(f(x_1, \dots, x_n)\rho\colon \vec{q}) = \Phi(\rho\colon \Delta) + k_0$. Now the claim follows as (i) $\Phi(t\sigma\colon \vec{q}) = \Phi(f(x_1, \dots, x_n)\rho\colon \vec{q})$, (ii) $\Phi(\rho\colon \Delta) = \sum_{i=1}^{n} (\Phi(\sigma\colon \Gamma_i) + k_i)$ and (iii) $k = \sum_{i=0}^{n} k_i$. Suppose Π has the following form.

$$\frac{\Gamma, x\colon \vec{r}, y\colon \vec{s} \mid\!\!\frac{k}{\vdash}\, t[x, y]\colon \vec{q} \quad \Upsilon(\vec{p} \mid \vec{r}, \vec{s}) \quad x,\, y \text{ are fresh}}{\Gamma, z\colon \vec{q} \mid\!\!\frac{k}{\vdash}\, t[z, z]\colon \vec{q}} \quad .$$

Then the claim follows by induction hypothesis in conjunction with Lemma 3.9.

Suppose Π has the following form.

$$\frac{}{x\colon \vec{q} \mid\!\!\frac{0}{\vdash}\, x\colon \vec{q}}$$

The lemma holds trivially. □

We state the following substitution lemma. The lemma follows by simple induction on t.

Lemma 3.13. *Let Γ be a context and let σ be a substitution consistent with Γ. Then $\Gamma \vdash t\colon \vec{q}$ implies $\vdash t\sigma\colon \vec{q}$.*

We establish soundness with respect to relative innermost rewriting.

Theorem 3.14. *Let $\mathcal{R}/\mathcal{S}$ be a resource bounded TRS and let σ be a normalised such that σ is consistent with the context Γ. Suppose $\Gamma \overset{k}{\vdash} t\colon \vec{p}$ and $t\sigma \xrightarrow{\mathrm{i}}{}^{K}_{\mathcal{R}} u\tau$, $K \in \{0,1\}$ for a normalising substitution τ. Then there exists a context Δ such that $\Delta \overset{\ell}{\vdash} t\colon \vec{q}$ is derivable and $\Phi(\sigma\colon\Gamma) + k - \Phi(\tau\colon\Delta) - \ell \geqslant K$.*

Proof. Let Π denote the derivation of the judgement $\Gamma \overset{k}{\vdash} t\colon \vec{q}$. The proof proceeds by case distinction on derivation $D\colon t\sigma \xrightarrow{\mathrm{i}}{}^{K}_{\mathcal{R}} u\tau$ and side-induction on Π.

Suppose D is empty, that is, $t\sigma$ is a normal form w.r.t. $\mathcal{R} \cup \mathcal{S}$. We distinguish two subcases. Either (i) $t\sigma$ contains only constructor or constructor-like symbols or (ii) $t\sigma$ contains at least one defined function symbol which does not occur as argument of the left-hand side of a rule in $\mathcal{R} \cup \mathcal{S}$.

For subcase (i), it suffices to show that $\Phi(\sigma\colon\Gamma) + k \geqslant \Phi(v\colon\vec{q})$ holds even under the assumption that no weakening rules are applied in Π. However, due to Lemma 3.9, $\Gamma \overset{k}{\underset{\mathsf{fp}}{\vdash}} t\colon\vec{q}$ implies that $\Phi(\sigma\colon\Gamma) + k = \Phi(v\colon\vec{q})$. Thus the theorem follows. Now consider subcase (ii). By definition $\Phi(v\colon\vec{q}) = 0$ and the theorem follows as potentials and amortised costs are non-negative.

Now suppose $D\colon t\sigma \xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}} u\tau$, that is D is non-empty. We exemplify the proof on three subcases.

For subcase (i), we assume that Π has the following form.

$$\dfrac{f \in \mathcal{C} \cup \mathcal{D} \quad [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)}{\underbrace{x_1\colon\vec{p}_1, \ldots, x_n\colon\vec{p}_n}_{=\Gamma} \overset{k}{\vdash} f(x_1, \ldots, x_n)\colon\vec{q}} \quad .$$

Then $\sigma = \{x_i \mapsto v_i \mid i = 1, \ldots, n\}$, where $v_i \in \mathsf{NF}$. By assumption on D there exists a rule $f(l_1, \ldots, l_n) \to r$ and a normalised substitution τ such that $f(l_1, \ldots, l_n)\tau = t\sigma$ and $t\tau = u$. Wlog. $f(l_1, \ldots, l_n) \in \mathcal{R}$. As $\mathcal{R}/\mathcal{S}$ is bounded there exist variables $y_1, \ldots, y_m$, resource annotation $\vec{r}_1, \ldots, \vec{r}_m$ and an amortised costs ℓ such that the following judgement is derivable.

$$\overbrace{y_1\colon\vec{r}_1, \ldots, y_l\colon\vec{r}_l}^{=:\Delta} \overset{k+\ell-1}{\vdash} r\colon\vec{q}\,.$$

Due to Lemmata 3.9 and 3.12 we obtain $\Phi(\sigma\colon\Gamma) + k = \Phi(\tau\colon\Delta) + \ell$. The theorem follows.

For subcase (ii), we assume that Π has the following form.

$$\dfrac{\overbrace{x_1\colon\vec{p}_1, \ldots, x_n\colon\vec{p}_n}^{=:\Delta_0} \overset{k_0}{\vdash} f(x_1, \ldots, x_n)\colon\vec{p} \quad \Gamma_i \overset{k_i}{\vdash} t_i\colon\vec{p}_i \quad \text{for all } i = 1, \ldots, n}{\Gamma_1, \ldots, \Gamma_n \overset{k}{\vdash} f(t_1, \ldots, t_n)\colon\vec{p}} \quad .$$

Wlog. t is linear. As $t\sigma \xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}} u\tau$, there exist (potentially empty) subderivations $D_i\colon t_i\sigma \xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}} u_i\tau$ for all $i = 1, \dots, n$. We set $D_0 := f(x_1, \dots, x_k)\rho \xrightarrow{\mathrm{i}}_{\mathcal{R}/\mathcal{S}} u\tau$.

By side induction hypothesis we conclude the existence of contexts Δ_i and annotations $\vec{q}_i$ such that $\Delta_i \vdash^{\ell_i} u_i\colon \vec{q}_i$ and $\Phi(\sigma\colon\Gamma_i) + k_i \geqslant \Phi(\tau\colon\Delta_i) + \ell_i$. Further let $\rho := \{x_i \mapsto u_i \mid i = 1, \dots, n\}$. Then again by induction hypothesis, there exists context Δ' and annotation $\vec{q}'$ such that $\Delta' \vdash^{\ell_0} u\colon\vec{q}'$ is derivable and $\Phi(\rho\colon\Delta) + k_0 \geqslant \Phi(\tau\colon\Delta') + \ell_0$. Observe that at most one of the inequalities in the potentials is strict. By distinguishing all possible subcases, the theorem follows.

For subcase (iii), we assume Π has the following form.

$$\frac{\Gamma, x\colon\vec{r}, y\colon\vec{s} \vdash^{k} t[x,y]\colon\vec{q} \quad \Upsilon(\vec{p}\,|\,\vec{r},\vec{s}) \quad x,\, y \text{ are fresh}}{\Gamma, z\colon\vec{q} \vdash^{k} t[z,z]\colon\vec{q}}\,.$$

Then the theorem follows from the side induction hypothesis in conjunction with Lemma 3.5. □

The next corollary is an immediate consequence of the theorem, highlighting the connection to similar soundness results in the literature.

Corollary 3.15. *Let $\mathcal{R}/\mathcal{S}$ be a bounded TRS and let σ be a normalising substitution consistent with the context Γ. Suppose $\Gamma \vdash^{k} t\colon\vec{q}$ and $D\colon t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}/\mathcal{S}} v \in \mathsf{NF}$. Then (i) $\vdash v\colon\vec{q}$ and (ii) $\Phi(\sigma\colon\Gamma) - \Phi(v\colon\vec{q}) + k \geqslant |D|$ hold.* □

The next theorem defines suitable constraints on the resource annotations to deduce polynomial innermost runtime from Theorem 3.14. Its proof follows the pattern of the proof of Theorem 14 in [28].

Theorem 3.16. *Suppose that for each constructor c with $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k'} \vec{q} \in \mathcal{F}(c)$, there exists $\vec{r}_i \in \mathcal{A}$ such that $\vec{p}_i \leqslant \vec{q} + \vec{r}_i$ where $\max\vec{r}_i \leqslant \max\vec{q} =: r$ and $p \leqslant r$ with $|\vec{r}_i| < |\vec{q}| =: k$. Then $\Phi(v\colon\vec{q}) \leqslant r|v|^k$, and thus the innermost runtime complexity of the TRS under investigation is in $\mathcal{O}(n^k)$.*

Proof. We proceed by induction on v. Observe that if $k = 0$ then $\Phi(v\colon\vec{q}) = 0$. Otherwise, we have

$$\begin{aligned}
\Phi(c(v_1, \dots, v_n)\colon\vec{q}) &\leqslant r + \Phi(v_1\colon\vec{p}_1 + \cdots + \Phi(v_n\colon\vec{p}_n)) \\
&\leqslant r + \Phi(v_1\colon\vec{q} + \vec{r}_1) + \cdots + \Phi(v_n\colon\vec{q} + \vec{r}_n) \\
&= r + \Phi(v_1\colon\vec{q}) + \Phi(v_1\colon\vec{r}_1) + \cdots + \Phi(v_n\colon\vec{q}) + \Phi(v_n\colon\vec{r}_n) \\
&\leqslant r(1 + |v_1|^k + |v_1|^{k-1} + \cdots + |v_n|^k + |v_n|^{k-1})\,,
\end{aligned}$$

where we have applied the induction hypothesis in conjunction with Lemma 3.5. The last expressed is bounded by $r(1 + |v_1| + \cdots + |v_n|)^k = r|v|^k$ due to the multinomial theorem. □

We note that our running example satisfies the premise of Theorem 3.16. Thus the linear bound on the innermost runtime complexity of the running example $\mathcal{R}_1$ follows. The next example clarifies that without further assumptions potentials are not restricted to polynomials.

Example 3.17. Consider that we annotate the constructors for natural numbers as $\mathsf{0}\colon[] \xrightarrow{0} \vec{p}$ and $\mathsf{s}\colon[2\vec{p}] \xrightarrow{p_1} \vec{p}$, where $\vec{p} = (p_1, \ldots, p_k)$. We then have, for example, $\Phi(t\colon 1) = 2^v - 1$, where v is the value represented by t. □

4 Implementation

In this section we describe the details of important implementation issues. The realisations of the presented method can be seen twofold. On one hand we have a standalone program which tries to directly annotate the input TRS without utilizing relative rewriting. While on the other hand we have an integration into T$_C$T [6] which uses relative rewriting. Clearly, as an integration into T$_C$T was planned from the beginning, the language used for the implementation of the amortised resource analysis module is Haskell[1]. The modular design of T$_C$T eased the integration tremendously.

The central idea of the implementation is the collection of all signatures and arising constraints occurring in the inference tree derivations. To guarantee resource boundedness further constraints are added such that uniqueness and superposition of constructors (cf. Section 3) is demanded and polynomial bounds on the runtime complexity are guaranteed (cf. Theorem 3.16).

Inference Tree Derivation and Resource Boundedness.

To be able to apply the inference rules the expected outcome judgement of each rule is generated (as in Example 3.11) by the program and the inference rules of Figure 3.1 are applied. To gain determinism the inference rules are ordered in the following way. The (share)-rule has highest priority, followed by (app), (var), (comp) and (w_4). In each step the first applicable rule is used while the remaining weakening rules (w_1), (w_2) and (w_3) are integrated in the aforementioned ones. For each application of an inference rule the emerging constraints are collected.

To ensure monomorphic typing of function signatures we keep track of a list of signatures. It uses variables in lieu of actual vectors. For each signature occurrence of defined function symbols the system refers to the corresponding entry in the list of signatures. Therefore, for each defined function symbol only one signature is added to the list of signatures. If the function occurs multiple times, the same references are used. Unlike defined function symbols multiple signature declarations of constructors are allowed, and thus each occurrence adds one signature to the list.

For the integration into T$_C$T we utilise the relative rewriting formulation. Instead of requiring all strict rules to be resource bounded, we weaken this requirement to have at least one strict rule being actually resource bounded, while the other rules may be annotated cost-free resource bounded. The SMT solver chooses which rule will be resource bounded. Clearly, this eases the constraint problem which is given to the SMT solver and thus improves performance.

[1] See `http://haskell.org/`.

Superposition of Constructors.

Recall that constructor and constructor-like symbols f must satisfy the superposition principle. Therefore, for each annotation $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$ of f it must be ensured that there is no annotation $[\lambda \cdot \vec{p_1} \times \cdots \times \lambda \cdot \vec{p_n}] \xrightarrow{\lambda \cdot k} \vec{q'}$ with $\lambda \in \mathbb{Q}^+$ and $q \neq \lambda \cdot q'$ in the corresponding set of annotated signatures. Therefore, for every pair (q, q') with $q' \geqslant q$ and $q > 0$ either for every $\lambda > 0 :\ q' \neq \lambda \cdot q$ or if $q' = \lambda \cdot q$ then the annotation must be of the form $[\lambda \cdot \vec{p_1} \times \cdots \times \lambda \cdot \vec{p_n}] \xrightarrow{\lambda \cdot k} \lambda \cdot \vec{q}$.

A naive approach is adding corresponding constraints for every pair of return annotations of a constructor symbol. This leads to universal quantifiers due to the scalar multiplication, which however, are available as binders in modern SMT solvers [8]. Early experiments revealed their bad performance. Overcoming this issue using Farkas' Lemma [11] is not possible here. It states that there either exists a $\vec{x} \geqslant 0$ (componentwise) with $\mathbf{A}\vec{x} = \vec{b}$ or there exists a $\vec{y}$ such that $\mathbf{A}^T\vec{y} \geqslant 0$ and $\vec{b}^T\vec{y} < 0$ [16]. For positive values this means that either $\neg(\forall x \colon \neg(\mathbf{A}\vec{x} = \vec{b}))$ must be true or there exists a $\vec{y}$ such that $\mathbf{A}^T\vec{y} \geqslant 0$. As the logical negation operator is natural in SMT solving this formulation could be used to transform a constraint with forall-quantifiers to the former existential quantified one if the constraint has the correct form. Unfortunately though the arising constraints are of the shape $\forall n_1, n_2 \in \mathbb{N}$: if $n_1 \cdot q_1 = n_2 \cdot q'$ then $n_1 \cdot p_i = n_2 \cdot p'_i$ for all argument annotations p_i and $n_1 \cdot k = n_2 \cdot k'$. The reason for the need n_1 and n_2 in comparison to just one variable as in Farkas' Lemma is caused by the fact that q may not be a multiple of q', e.g. $q = 5$ and $q' = 3$. Nonetheless, the number of required constraints increase exponentially by the amount of constructor annotations if every pair of annotations needs to be considered. Clearly, this thus not scale.

Thus, we developed a heuristic of spanning up a vector space using unit vectors for the annotation of the return types for each constructor. Each annotated signature of such a symbol must be a linear combination of the base signatures.

Both methods, universal quantifiers and base signatures lead to non-linear constraint problems. However, these can be handled by some SMT solvers[2]. Thus, in contrast to the techniques presented in [20,22,23], which restrict the potential function to pre-determined data structures, like lists or binary trees, our method allows any kind of data structure to be annotated.

Example 4.1. Consider the base constructor annotations $\sharp_1 \colon [(0,0) \times (1,0)] \xrightarrow{1} (1,0)$ and $\sharp_2 \colon [(0,0) \times (2,1)] \xrightarrow{1} (0,1)$ for a constructor $\sharp$. An actual instance of an annotated signature is $n_1 \cdot \sharp_1 + n_2 \cdot \sharp_2$ with $n_1, n_2 \in \mathbb{N}$. As the return types can be seen as unit vectors of a Cartesian coordinate system, the superposition and uniqueness properties hold. Note that $\sharp_1$ represents the linear part of the signature, whereas $\sharp_2$ the quadratic portion. □

[2] We use the SMT Solvers z3 (`https://github.com/Z3Prover/z3/wiki`) and MiniSmt (`http://cl-informatik.uibk.ac.at/software/minismt/`).

$$\frac{\begin{array}{cc} f \in \mathcal{C} \cup \mathcal{D} & y_1{:}\,\vec{r}_1, \ldots, y_l{:}\,\vec{r}_l \text{ and } \ell \text{ freed} \\ & \forall\, f(l_1, \ldots, l_n) \to r \in \mathcal{R}\colon\ y_1{:}\,\vec{r}_1, \ldots, y_l{:}\,\vec{r}_l \left|\frac{\vec{k}^{cf}+\ell}{\mathsf{cf}}\right. r{:}\,\vec{q} \\ [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f) & [\vec{p}_1^{\,cf} \times \cdots \times \vec{p}_n^{\,cf}] \xrightarrow{\vec{k}^{cf}} \vec{q}^{\,cf} \in \mathcal{F}^{\,cf}(f) \end{array}}{x_1{:}\,\vec{p}_1, \ldots, x_n{:}\,\vec{p}_n \left|\frac{k+\vec{k}^{cf}}{}\right. f(x_1, \ldots, x_n){:}\,\vec{q} + \vec{q}^{\,cf}}\ (\mathsf{app}_{cf})$$

Figure 4.1: Additional Function Application Inference Rule for Cost-Free Derivation.

Cost-Free Function Symbols.

Inspired by Hoffmann [20, p.93ff] we additionally implemented a cost-free inference tree derivation when searching for non-linear bounds. The idea is that for many non-tail recursive functions the freed potential must be the one of the original function call plus the potential that gets passed on.

The inference rules are extended by an additional (app_{cf})-rule, which separates the function signature into two parts, cf. Figure 4.1. On the left there are the monomorphic and cost-free signatures while on the right a cost-free part is added. For every application of the rule the newly generated cost-free signature annotation must be cost-free resource bounded, for this the cost-free type judgement indicated has to be derived for any rule $f(l_1, \ldots, l_n) \ \to \ r$ and freed context $y_1{:}\,\vec{r}_1, \ldots, y_l{:}\,\vec{r}_l$ and cost ℓ. Thus, the new set of annotations for a defined function symbols f is given by the following set, cf. [20, p. 93].

$$\{[\vec{p}_1 + \lambda \cdot \vec{p}_1^{\,cf} \times \cdots \times \vec{p}_n + \lambda \cdot \vec{p}_n^{\,cf}] \xrightarrow{k+\lambda\cdot\vec{k}^{cf}} \vec{q} + \lambda \cdot \vec{q}^{\,cf} \mid \lambda \in \mathbb{Q}^+, \lambda \geqslant 0\}\ .$$

The decision of which application rules, (app) or (app_{cf}), is applied utilises the strongly connected components (SCC) of the call graph analysis as done in [20, p.93ff].

Suppose f is the function being analysed. Whenever, a currently deriving function g, where g is not constructor-like, resides in the SCC of f and the current derivation is not the cost-free analysis of f, the cost-free application rule of Figure 4.1 is used. In all other cases, the original function application rule of Figure 3.1 is used.

We have experimented with this heuristic, by pruning the need of the SCC requirement, such that the cost-free inference rule for application is not just allowed when g resides in the SCC of f, but also when g is reachable from f in the call graph. This adaption however, increased the constraint problem size tremendously on one hand which obviously resulted in longer execution times but on the other hand could only infer one new cubic example and move one example from a cubic polynomial upper bound to quadratic. This makes sense, as the motivation behind this extension lies in the restricted feasibility of monomorphically annotations of recursive calls.

The implementation of the cost-free derivations uses a new list of signatures. Within this defined function symbol signatures and constructor symbol signatures can occur multiple times. However, clearly for each inference tree derivation only a single instance of a signature for each defined function symbol may be used. Nonetheless, for each cost-free derivation different annotations of cost-free signatures of defined function symbols are allowed.

Alternative Implementation of the Superposition Principle.

Similar to [20,28] we integrated the additive shift $\lhd(\vec{p})$ and interleaving $\vec{p} \,|||\, \vec{q}$ for constructors when type information is given. Here $\lhd(p_1, \ldots, p_k) := (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$ and $\vec{p} \,|||\, \vec{q} := (p_1, q_1, p_2, q_2, \ldots, p_k, q_k)$, where the shorter of the two vectors is padded with 0s. These heuristics are designed such that the superposition principle holds, without the need of base annotations. Therefore, the constraint problem automatically becomes linear as there is no need for base signatures whenever these heuristics are used. This tremendously reduces the problem size and with that the execution times.

Example 4.2. For instance, the additive shift $\lhd(\vec{p})$ for constructor signatures is integrated and ensures superposition by design when enabled. It is defined as $\lhd(\vec{p}) := (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$ for a vector $\vec{p}$ with length k. □

However, according to the experiments these heuristics are only rarely applicable and often require comprehensive type information. This additional information allows to separate constructors named alike but with different types. For instance, a list of lists can then have different base annotations compared to a simple list, even though the constructors have the same name. The rather poor performance of these heuristics in the presence of only generic type information came as a surprise to us. However, in hindsight it clearly showcases the importance of comprehensive type information (as e.g. demanded by RaML) for the efficiency of automation of resource analysis in functional programming.

5 Numerical Example

In this section we look at a full numerical example to clarify the aforementioned concepts. For this we consider the following rewrite system mult3 from the TPDB.

$$\begin{array}{ll} 1\colon \quad \mathsf{0} + y \to y & 3\colon \quad \mathsf{0} \cdot y \to \mathsf{0} \\ 2\colon \mathsf{s}(x) + y \to \mathsf{s}(x + y) & 4\colon \mathsf{s}(x) \cdot y \to x + (x \cdot y) \end{array}$$

The first two rules recursively implement addition for natural numbers in the standard way, while the others utilise this implementation to (falsely) implement multiplication. The flaw in the program is located in rule four. The translation of the successor function to standard equational notation makes it easy to find the reason for the unexpected behavior of the program. The rule translates to $(1 + x) \cdot y \to \mathbf{x} + (x \cdot y)$ whereas the correct rule for multiplication ought to be $(1 + x) \cdot y \to \mathbf{y} + (x \cdot y)$. Nonetheless, it will be fully analysed in this section to exemplify the application of the method. For the rest of this section we will not make use of the cost-free application rule (app_{cf}) but concentrate on demonstrating the method using the inference rules given in Figure 3.1.

Our tool derives the following annotations. Recall that functions are monomorphically typed, whereas the constructor symbols are build using base constructor annotations ($\mathsf{0}_1, \mathsf{0}_2$ and $\mathsf{s}_1, \mathsf{s}_2$ respectively) and the implicit filling up with 0s of annotations. Thus, all constructor annotations for e.g. s are linear combinations of the given base constructor annotations $n_1 \cdot \mathsf{s}_1 + \ n_2 \cdot \mathsf{s}_2$ with fresh variables n_1 and n_2.

$$\begin{array}{ll} +\colon [15 \times 0] \xrightarrow{12} 0 & \cdot\colon [(0,15) \times 0] \xrightarrow{8} 0 \\ \mathsf{0}_1\colon [] \xrightarrow{0} 1 & \mathsf{0}_2\colon [] \xrightarrow{0} (0,1) \\ \mathsf{s}_1\colon [1] \xrightarrow{1} 1 & \mathsf{s}_2\colon [(1,1)] \xrightarrow{1} (0,1) \end{array}$$

As demanded by Definition 3.10 all rules must be resource bounded. Therefore, for rule one the judgement $y\colon 0 \mid\!\!\xrightarrow{12-1+0} y\colon 0$ must be derivable. For the first argument the typing $\mathsf{0}\colon 15$ with freed cost 0 is used, thus $\mid\!\!\xrightarrow[\mathsf{fp}]{0} \mathsf{0}\colon 15$ has to be derivable as well. Both requirements are met by our annotations as can be seen by the following derivations.

$$\dfrac{\dfrac{\dfrac{}{y\colon 0 \mid\!\!\overset{0}{\vdash} y\colon 0}\ (\mathsf{var})}{y\colon 0 \mid\!\!\overset{11}{\vdash} y\colon 0}\ (\mathsf{w}_1)} \qquad\qquad \dfrac{\mathsf{0}\colon [] \overset{0}{\to} 15}{\mid\!\!\overset{0}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon 15}\ (\mathsf{app})$$

The same way the third rule can be shown to be resource bounded. The required judgements are $y\colon 0 \mid\!\!\overset{8-1+0}{\vdash} \mathsf{0}\colon 0$ for the rule and $\mid\!\!\overset{8}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon (0, 15)$ for the first argument with the following inference derivations.

$$\dfrac{\dfrac{\dfrac{\mathsf{0}\colon [] \overset{0}{\to} 0}{\mid\!\!\overset{0}{\vdash} \mathsf{0}\colon 0}\ (\mathsf{app})}{\mid\!\!\overset{7}{\vdash} \mathsf{0}\colon 0}\ (\mathsf{w}_1)}{y\colon 0 \mid\!\!\overset{7}{\vdash} \mathsf{0}\colon 0}\ (\mathsf{w}_4) \qquad\qquad \dfrac{\mathsf{0}\colon [] \overset{0}{\to} (0, 15)}{\mid\!\!\overset{0}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon (0, 15)}\ (\mathsf{app})$$

As the constructor signatures $\mathsf{0}\colon [] \overset{0}{\to} 15$, $\mathsf{0}\colon [] \overset{0}{\to} 0$ and $\mathsf{0}\colon [] \overset{0}{\to} (0, 15)$ are all in the set of annotations both rules are resource bounded. However the more comprehensive recursive rules still have to be investigated.

By using the annotation signatures $+\colon [15 \times 0] \overset{12}{\longrightarrow} 0$ and $\mathsf{s}\colon [15] \overset{15}{\longrightarrow} 15$ the judgements for proofing resource boundedness of the second rule are $x\colon 15, y\colon 0 \mid\!\!\overset{12-1+15}{\vdash} \mathsf{s}(x + y)\colon 0$ and $x\colon 15 \mid\!\!\overset{15}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon 15$. Note that the typing $x\colon 15$ is obtained by using the aforementioned signature of s on the corresponding argument of the left hand side of the rewrite rule.

$$\dfrac{\dfrac{\mathsf{s}\colon [0] \overset{0}{\to} 0}{z_1\colon 0 \mid\!\!\overset{0}{\vdash} \mathsf{s}(z_1)\colon 0}\ (\mathsf{app}) \quad \dfrac{\dfrac{+\colon [15 \times 0] \overset{12}{\longrightarrow} 0}{x\colon 15, y\colon 0 \mid\!\!\overset{12}{\vdash} x + y\colon 0}\ (\mathsf{app})}{x\colon 15, y\colon 0 \mid\!\!\overset{26}{\vdash} x + y\colon 0}\ (\mathsf{w}_1)}{x\colon 15, y\colon 0 \mid\!\!\overset{26}{\vdash} \mathsf{s}(x + y)\colon 0}\ (\mathsf{comp}) \qquad \dfrac{\mathsf{s}\colon [15] \overset{15}{\longrightarrow} 15}{x\colon 15 \mid\!\!\overset{15}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon 15}\ (\mathsf{app})$$

Note that all annotations of the derivations of the + operator have length 1. This means that the addition can be bounded by a linear function from above.

However when rule four is added to the TRS annotations having at least the length of 2 are needed to show resource boundedness. For rule four we utilise the signatures $\cdot\colon [(0, 15) \times 0] \overset{8}{\to} 0$ and $\mathsf{s}\colon [(15, 15)] \overset{15}{\longrightarrow} (0, 15)$ to gain the judgements $x\colon (15, 15), y\colon 0 \mid\!\!\overset{8-1+15}{\vdash} x + (x \cdot y)\colon 0$ and $x\colon (15, 15) \mid\!\!\overset{15}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}(x)\colon (0, 15)$. The following inference tree derivations proof resource boundedness of the rewrite rule.

$$\dfrac{\dfrac{\dfrac{+\colon [15\times 0]\xrightarrow{12} 0}{z_1\colon 15, z_2\colon 0 \mid\!\overset{12}{\vdash} z_1+z_2\colon 0}\ (\mathsf{app}) \quad \dfrac{}{x_1\colon 15 \mid\!\overset{0}{\vdash} x_1\colon 15}\ (\mathsf{var}) \quad \dfrac{\dfrac{\cdot\colon [(0,15)\times 0]\xrightarrow{8} 0}{x_2\colon (0,15), y\colon 0 \mid\!\overset{8}{\vdash} x_2\cdot y\colon 0}\ (\mathsf{app})}{x_2\colon (0,15), y\colon 0 \mid\!\overset{10}{\vdash} x_2\cdot y\colon 0}\ (\mathsf{w_1})}{x_1\colon 15, x_2\colon (0,15), y\colon 0 \mid\!\overset{22}{\vdash} x_1+(x_2\cdot y)\colon 0}\ (\mathsf{comp})}{x\colon (15,15), y\colon 0 \mid\!\overset{22}{\vdash} x+(x\cdot y)\colon 0}\ (\mathsf{share})$$

$$\dfrac{\mathsf{s}\colon [(15,15)]\xrightarrow{15}(0,15)}{x\colon (15,15) \mid\!\overset{15}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon (0,15)}\ (\mathsf{app})$$

Here the (share) rule is applied to split the potential of $x\colon (15,15)$ into the parts $x_1\colon (15,0)$ and $x_2\colon (0,15)$. Remember the implicit filling up with 0s and the dropping of parenthesis for annotations of length 1, thus the typings $x_1\colon 15$ and $x_1\colon (15,0)$ coincide. Further, note that the costs before applying the (comp) rule and the sum of the costs after applying it is equal.

As all rules are resource bounded and as superposition as well as uniqueness hold by design we conclude that the TRS mult3 is in $\mathsf{O}(n^2)$.

6 Experimental Results

In this Section we have a look at how the amortised analysis deals with some selected examples including the thesis part's running example queue. Furthermore we presented evaluation results to show the viability of the method.

All experiments[1] were conducted on a machine with an Intel Xeon CPU E5-2630 v3 @ 2.40GHz (32 threads) and 64GB RAM. The timeout was set to 60 seconds. For benchmarking we use the runtime complexity innermost rewriting folder of the TPDB[2] as well as a collection consisting of 140 TRSs representing first-order functional programs [14, 18], transformations from higher-order programs [5], or RaML programs [22] and interesting examples from the TPDB. We compared the competition version of TCT 2016 to the current version of TCT with and without (w/o) the amortised resource analysis (ARA), as well as the output of AProVE as presented in [32][3]. Figure 6.1 gives an overview of interesting examples and summarises some of the TRSs discussed below.

#3.42 – Binary representation. Given a number n in unary encoding as input, the TRS computes the binary representation $(n)_2$ by repeatedly halving n and computing the last bit, see Figure 6.2. The optimal runtime complexity of $\mathcal{R}_1$ is linear in n. For this, first observe that the evaluation of $\mathsf{half}(\mathsf{s}^m(\mathsf{0}))$ and $\mathsf{lastbit}(\mathsf{s}^m(\mathsf{0}))$ requires about m steps in total. Secondly, n is halved in each iteration and thus the number of steps can be estimated by $\sum_{i=0}^{k} 2^i$, where $k := |(n)_2|$. As the geometric sum computes to $2 \cdot 2^k - 1$, the claim follows. Such a precise analysis is enabled by an amortised analysis, which takes the sequence of subsequent function calls and their respective arguments into account. Compared to former versions of TCT which reported $\mathsf{O}(n^2)$ we find this optimal linear bound of $\mathsf{O}(n)$ when ARA is enabled. Furthermore, the best-case analysis of ARA shows that this bound is tight by returning $\Omega(n)$. Similarly AProVE [17] yields the tight bound employing a size abstraction to *integer transition systems* (*ITSs* for short), cf. [32]. The resulting ITSs are then solved with CoFloCo [12], which also embodies an amortisation analysis.

bfs.raml – Depth/Breadth-First Search. This TRS is a translation of depth-first search (DFS) and breadth-first search (BFS) from RaML syntax, see Listing 6.3, and can be found in the TPDB. Note that the TRS uses strict rules for the equality check which

[1] Detailed data is available at `http://cl-informatik.uibk.ac.at/software/tct/experiments/ara_flops/`.

[2] We refer to Version 10.4 of the Termination Problem Database, available from `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB`.

[3] See `https://aprove-developers.github.io/trs_complexity_via_its/` for detailed results of AProVE. Timeout: 300 seconds, Intel Xeon with 4 cores at 2.33 GHz and 16GB of RAM.

Example	TCT '16	TCT w/o ARA	TCT with ARA	AProVE
queue	$O(n^2)$	$O(n^2)$	$O(n^1)$	$O(n^2)$
#3.42	$O(n^2)$	$O(n^3)$	$O(n^1)$	$O(n^1)$
bfs.raml			$O(n^3)$	
insertionsort			$O(n^3)$	$O(n^2)$
4.33	$O(n^3)$	$O(n^3)$	$O(n^3)$	
tpa2			$O(n^2)$	$O(n^2)$
splitandsort.raml			$O(n^5)$	$O(n^3)$
matrix.raml			$O(n^7)$	
#4.27.trs	$O(n^3)$	$O(n^3)$	$O(n^2)$	$O(n^2)$

Figure 6.1: Experimental evaluation of selected TRSs.

$$
\begin{aligned}
&1\colon\ \mathsf{conv}(0) \to \mathsf{nil} && 5\colon\ \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{s}(\mathsf{half}(x))\\
&2\colon\ \mathsf{conv}(\mathsf{s}(x)) \to \mathsf{conv}(\mathsf{half}(\mathsf{s}(x))) \sharp\, \mathsf{lastbit}(\mathsf{s}(x)) && 6\colon\ \mathsf{lastbit}(0) \to 0\\
&3\colon\ \mathsf{half}(0) \to 0 && 7\colon\ \mathsf{lastbit}(\mathsf{s}(0)) \to \mathsf{s}(0)\\
&4\colon\ \mathsf{half}(\mathsf{s}(0)) \to 0 && 8\colon\ \mathsf{lastbit}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{lastbit}(x)
\end{aligned}
$$

Figure 6.2: Example #3.42 of the TPDB.

recurses on the given data structure. In DFS a binary tree is searched one branch after the other for a matching entry while BFS uses two lists to keep track of nodes of a binary tree to be visited. The first one is used to traverse on the nodes of the current depth, whereas the second list collects all nodes of the next depth to visit. After each iteration the futurelist is reversed. Further, note that BFS is called twice in the function bfs2. TCT with ARA is the only tool which is able to infer a complexity bound of $O(n^3)$.

insertionsort.raml/splitandsort.raml – Sorting. Insertionsort has quadratic runtime complexity, although TCT with ARA using the default setup can only find a cubic upper bound, as it handles the trade off between execution time and tightness of the bound. If TCT is triggered to find the best bound within the timeout, it will infer $O(n^2)$ as AProVE does. This bound is tight [20, p.158ff]. The best-case analysis finds a linear lower bound for this implementation of insertionsort. splitandsort.raml first groups the input by a specified key and then sorts each grouped list using quicksort. The optimal runtime complexity for this program is $O(n^2)$ [20, 158ff]. Although far from being optimal, TCT with ARA is able to find the worst-case upper bound $O(n^5)$, whereas AProVE infers a cubic bound.

tpa2 – Multiple Subtraction. This TRS from the TPDB iterates subtraction until no more rules can be applied. The latest version of TCT with ARA is in comparison to an older version able to solve the problem. The inferred quadratic worst-case bound

```
dfs (queue,x) = match queue with
| [] -> leaf
| (t::ts) -> match t with
| leaf -> dfs(ts,x)
| node(a,t1,t2) ->
if a == x then t else dfs(t1::t2::ts,x);
dodfs (t,x) = dfs([t],x);
bfs(queue,futurequeue,x) = match queue with
| [] -> match futurequeue with
| [] -> leaf
| (t::ts) -> bfs(reverse(t::ts),[],x)
| (t::ts) -> match t with
| leaf -> bfs(ts,futurequeue,x)
| node(y,t1,t2) -> if x==y then node(y,t1,t2)
else bfs(ts,t2::t1::futurequeue,x);
dobfs(t,x) = bfs([t],[],x);
bfs2(t,x) = let t' = dobfs(t,x) in dobfs(t',x);
```

Figure 6.3: DFS and BFS in RaML Syntax [20, p.70] of which the translation to an TRS can be found in the TPDB.

coincides with the bounds provided by AProVE.

matrix.raml – Matrix Operations. This TRS implements transposing of matrices and matrix multiplications for a list of matrices, three matrices and two matrices, see Listing 6.4 for an excerpt in RaML syntax of the implemented matrix multiplication for two matrices, of which the second one is already transposed. The program maps over the matrix m1 line by line, for each line mapping over matrix m2 calling mult on the corresponding entries. Clearly, if the $*$-function is seen as one operation, as in the TRS, this program has cubic worst-case runtime complexity. Due to ARA, the latest version of T$_C$T can now handle this TRS and returns a complexity bound of $O(n^7)$ in the default setup, but when the best bound is looked for, T$_C$T returns the asymptotically optimal upper bound defined by the list matrix multiplication of $O(n^4)$. Neither the older version of T$_C$T nor AProVE is able to find any upper bound for this TRS.

Experimental Evaluation

We have conducted several further experiments on the TPDB, as well as on the smaller testbed composed of interesting examples with the focus on program translations. Over the last year the strategy of T$_C$T was adapted to focus on TRSs which were translated from functional programs. Thus, the examples which can be solved are distinct from the T$_C$T competition strategy of 2016 to a great extent. Due to ARA the latest competition

```
matrixMult' (m1,m2) = match m1 with
| nil -> nil
| (l::ls) -> (lineMult (l,m2))::(matrixMult' (ls,m2));
lineMult (l,m2) = match m2 with
| nil -> nil
| (x::xs) -> (mult (l,x))::(lineMult (l,xs));
mult (l1,l2) = match l1 with
| [] -> +0
| (x::xs) -> match l2 with
| [] -> +0
| (y::ys) -> x*y + (mult (xs,ys));
```

Figure 6.4: Excerpt of the original RaML code translated to matrix.raml from the TPDB.

strategy of TCT can solve 5 more examples of the TPDB than without ARA and for 14 examples a better bound can be inferred. On the small testbed TCT with ARA can find better bounds for 5 examples in contrast to TCT without ARA and additionally bfs.raml can be solved.

Figure 6.5 shows the results on the TPDB. The upper half shows the number of analyses, whereas the average execution times are listed on the lower half. The columns represent the former TCT competition strategy of 2016, the latest TCT competition without using ARA, the latest TCT competition strategy including ARA and finally for reference the results of AProVE. Note that the latest versions of TCT with ARA is able to find better bounds as without ARA. Further, even though TCT Comp 2016 finds better bounds compared to the latest competition strategy the examples lost by the translation of the competition strategy are mostly unpractical TRSs, see the detail results. The results of the evaluations on the independent testbed are listed in Figure 6.6. Again the upper half shows the number of analysed TRSs according to the result of the evaluation, whereas the lower half lists the average execution times. The first three result columns are as above, however the last column lists the results for using the heuristics. The TRSs in this testbed provide generic type information which are needed by the heuristics.

Constructor Heuristics. To gain a better insight in the method we experimented with the shift and interleaving heuristics as discussed in Section 3, cf. Figure 6.6. Clearly the use of heuristics reduce execution times. For instance, the example typed-pairsp has a cubic worst-case upper bound, which can be inferred using all given variants of ARA. However, using heuristics the execution times are reduced by a factor between 2 to 3. Nonetheless, the applicability of the heuristics is rather limited, especially when the type information given is not comprehensive enough, as discussed earlier. Further experiments show that when only simple type inference is used only 7.0% of the examples of the TPDB can be solved using the heuristics, whereas the full method is able to solve 17.1% of the TPDB problems.

	Number of Analyses			
Result	TcT Comp 2016	TcT No ARA	TcT ARA	AProVE
O(1)	40	39	39	53
O(n^1)	234	227	232	326
O(n^2)	85	61	63	127
O(n^3)	24	49	44	35
O(n^4)	4	6	5	2
O(n^5)	1	2	4	4
O(n^6)	0	0	1	0
O(n^7)	0	0	1	0
O(n^{10})	0	1	1	1
MAYBE	595	629	617	125
Timeout	39	8	15	344
Sum Success	388	385	390	548
	Execution Time (in seconds)			
Result	TcT Comp 2016	TcT No ARA	TcT ARA	AProVE
O(1)	0.22	0.19	0.93	2.03
O(n^1)	0.50	0.60	1.85	34.10
O(n^2)	5.75	4.74	6.50	151.77
O(n^3)	10.02	4.30	8.00	190.17
O(n^4)	26.62	24.49	24.29	258.31
O(n^5)	15.08	7.14	20.19	299.98
O(n^6)	0.00	0.00	46.97	0.00
O(n^7)	0.00	0.00	22.73	0.00
O(n^{10})	0.00	2.04	2.12	300.00
MAYBE	37.47	56.21	56.21	119.33
Timeout	60.05	60.05	60.01	300.00
Avgerage	25.06	35.85	36.28	154.27

Figure 6.5: Experimental Evaluation on the TPDB.

Additional Experiments. Further we integrated the amortised resource analysis method twofold into TcT. On one hand we replaced the polynomial interpretation methods while on the other hand we integrated the method such that it is additionally used in parallel when the polynomial interpretations are used as well. Both variants of ARA are able to solve more examples than the polynomial interpretation methods. Adding to this, we observed that ARA performs better when the SMT solver Z3 is used in contrast to MiniSMT.[4]

[4]These results can be found at `http://cl-informatik.uibk.ac.at/software/tct/experiments/ara/`.

	Number of Analyses			
Result	TCT Comp 2016	TCT No ARA	TCT ARA	ARA Heuristics
O(1)	2	2	2	1
O(n^1)	58	54	59	14
O(n^2)	37	21	17	4
O(n^3)	7	21	21	0
O(n^4)	2	7	7	0
O(n^5)	0	1	1	0
MAYBE	34	34	31	121
Timeout	0	0	2	0
Sum Success	106	106	107	19

	Execution Time (in seconds)			
Result	TCT Comp 2016	TCT No ARA	TCT ARA	ARA Heuristics
O(1)	0.14	0.04	0.05	0.06
O(n^1)	0.30	0.37	0.54	0.10
O(n^2)	5.87	0.74	2.86	0.16
O(n^3)	15.62	1.44	5.06	0.00
O(n^4)	30.29	7.32	8.95	0.00
O(n^5)	0.00	17.6	18.47	0.00
MAYBE	34.08	58.28	58.19	0.28
Timeout	0.00	0.00	60.00	0.00
Average	11.17	15.12	15.66	0.25

Figure 6.6: Experimental evaluation on the independent testbed.

Part II

Best-Case Lower Bounds for Term Rewrite Systems

7 Motivation

In this part of the thesis we study best-case complexity for first-order eagerly evaluated functional programs and establish novel methods to automatically infer lower bounds on the best-case complexity. The traditional applications of the analysis of lower bounds include performance debugging, program optimisation and verification. But in the literature one also finds the automation of parallelisation as application area, cf. [3, 9]. Yet another possible application, in conjunction with automated inference of tight worst-case upper bounds, is cyber security, in particular information leakage vulnerabilities may manifest itself in an input-dependent difference between (worst-case) upper and (best-case) lower bound runtime complexity, cf. [33].

While the study of upper bounds on the (worst-case) runtime complexity of programs is well-developed, the automated analysis of lower bounds for runtime complexity is somewhat underdeveloped. The literature mainly focuses on lower bound analysis for imperative and logic programs [3, 9]. In the setting of functional programming (or term rewriting for that matter) we are only aware of the analysis provided by Chang Ngo et. al. [33].

Rather than following the focus in [33] on a particular programming language, we follow in this thesis the general approach of static program analysis, where peculiarities of specific programming languages are suitable abstracted to give way to more general constructions like *recurrence relations*, *cost relations*, *transition systems*, *term rewrite systems*, etc. Thus we seek a more general discussion of lower bound analysis in (first-order) functional programming, where we choose term rewrite systems as suitable abstraction. Based on earlier work on automating amortised complexity analysis, we study univariate amortised resource analysis in the context of rewrite systems and provide experimental data of a prototype implementation. Furthermore, yet another theoretical highlight is the invocation of a small-step semantics, resulting in an analysis which does not presupposes termination.

Term rewrite systems provide a formal model of computation that underlies declarative programming and which is heavily used in symbolic computation in mathematics and theorem proving. Our interest in term rewriting stems from its natural focus on general data types and non-deterministic computation on the one hand and the ability to naturally represent inexact size constraints and multiple arguments on the other hand. Furthermore, the thus obtained results can be immediately applied to the best-case lower bound analysis of term rewrite systems (TRSs for short), while no prior results in this direction exist.

Let us now look at a concrete example, which we will use as running example throughout this part of the thesis.

Example 7.1. Consider the following simple TRS reverse, encoding the reversal of a list.

$$\begin{array}{lll} 1\colon & \mathsf{nil} \mathbin{@} xs \to xs & 3\colon \quad \mathsf{reverse}(\mathsf{nil}) \to \mathsf{nil} \\ 2\colon & (x \mathbin{\sharp} xs) \mathbin{@} ys \to x \mathbin{\sharp} (xs \mathbin{@} ys) & 4\colon \mathsf{reverse}(x \mathbin{\sharp} xs) \to \mathsf{reverse}(xs) \mathbin{@} (x \mathbin{\sharp} \mathsf{nil}) \end{array}$$

The worst-case runtime complexity of the computation of $\mathsf{reverse}(ls)$ is quadratic in the length of the input list ls. Furthermore, the TRS is confluent or non-ambiguous. In combination with an innermost rewriting strategy this implies that worst-case and best-case complexity coincide. Indeed our prototype implementation can derive the quadratic best-case complexity of reverse fully automatically. □

Technically, the main contributions of this part of the thesis are as follows.

- Conceptualisation and definition of best-case lower bound analysis in the context of term rewriting and functional programming. In contrast to the definition of worst-case upper bound and lower bounds, a more contrived definition is required in the context of best-case bounds due to the non-deterministic nature of the underlying computation model.
- Development of an automated lower bound analysis of term rewrite systems based on amortised resource analysis. Our results are directly applicable to functional programming and do not presuppose typing or specific data structures.
- Incorporation of a small-step semantics suited to amortised analysis. Thus, our analysis does not presuppose termination.
- We have provided a prototype implementation. In this prototype we focus on easy usability, in particular no user interaction is required.

We briefly comment on some of these contributions. We focus on *asymptotic* lower bounds, although for some applications precise bounds would be better. This is a design choice, as precise bounds can be obtained if either the used potential functions are set more restrictive or more interaction with the user is requested. We consider these as orthogonal considerations. Furthermore, we emphasise that our basis computational model, term rewriting, does not require types and neither does our analysis. This is natural in the context of transformational resource analysis, where we do not want to exclude the analysis of untyped languages.

Related Work

With respect to related work, in particular in the work on resource analysis following the seminal approach by Wegbreit [44,45], lower bound analysis has been most prominent. In particular Debray et al. provide in [9] a lower bound analysis of logic programs, resting on a suitable representation of the program flow as cost relations. A conceptually similar approach, has been established by Albert et al. [3]. These works contrast with our

contribution mainly on the paramount focus on cost relations, while we directly work on an abstraction of functional programs, thus deriving lower bounds through a more direct translation of computations to constraints stemming from our amortised resource analysis.

The approach by Chan Ngo et. [33] to lower bound analysis within the tool RaML is, like ours, based on an amortised analysis. Thus conceptually these approaches are quite similar. However, we go beyond the results established in [33] as we not only allow a more direct representation of arbitrary data structures, but also show how an amortised lower bound analysis can be performed without reference to a typed programming language.

We contrast our findings to research on worst-case lower bounds as have been conducted in particular in the area of rewriting, cf. [15]. Worst-case lower bounds have mainly been motivated through the quest for tight (worst-case) upper bounds, but cannot be employed in the context of the application areas mentioned above. In particular the techniques are complementary to those presented in this thesis. Finally, we briefly cite related work on (automated) amortised resource analysis apart from Chan Ngo et al. [20, 22, 23, 26, 28–30].

The rest of this part of the thesis is structured as follows. In the next section we recall basics and delineate a workable definition of best-case complexity in the context of (non-deterministic) TRSs and introduce the definition of lower bound formally. The amortised analysis and the inference system embodying this analysis is given in Section 9. In this section we also state soundness of the method (Theorem 9.11). In Section 10 we introduce refinements and remark on challenges posed by automation. Section 12 provides the experimental assessment of the method. Finally we conclude in Section 13, where we also sketch future work.

8 Best-Case Complexity

In the following we shortly recall the most important preliminaries of Section 2 and then establish the small-step semantics for the best case lower bound analysis.

Recall that the size of a term t, that is, the number of symbols in t is denoted $|t|$. Furthermore, the depth of a term t, denoted as $\mathsf{depth}(t)$, is defined as the height of the tree of positions of t. For example $\mathsf{depth}(\mathsf{f}(\mathsf{a},\mathsf{c})) = 2$.

Let σ be a substitution and V be a set of variables; $\sigma \restriction V$ denotes the restriction of the domain of σ to V. The substitution τ is called an *extension* of substitution σ if $\tau \restriction \mathsf{dom}(\sigma) = \sigma$. Let σ, τ be substitutions such that $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\tau) = \varnothing$. Then we denote the disjoint union of σ and τ as $\sigma \uplus \tau$.

In the sequel we study *constructor* TRSs $\mathcal{R}$, that is, all arguments of left-hand sides are basic. That is, there are no constructor-like function symbols as introduced in part one. Furthermore, we restrict to *completely defined* and *orthogonal* systems. A TRS is completely defined if all ground normal forms are values.

Let s and t be terms, such that t is in normal-form. Then a *derivation* $D\colon s \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}} t$ with respect to a TRS $\mathcal{R}$ is a finite sequence of rewrite steps. For the rest of the thesis the *length* of a derivation D, that is, the number of rewrite steps, is denoted as $|D|$.

Recall that if $\mathcal{R}$ is completely defined, any derivation starting with a ground term ends in a value. In connection with innermost rewriting this yields a *call-by-value* strategy. In addition in this part we assume $\mathcal{R}$ to be orthogonal. Hence the reduction strategy is *non-ambiguous* in the following strong sense. See [4] for a proof of the lemma.

Lemma 8.1. *All normalising reductions of a term t have the same length and yield the same result, that is, if $t \xrightarrow{\mathrm{i}}{}^{m}_{\mathcal{R}} u$ and $t \xrightarrow{\mathrm{i}}{}^{n}_{\mathcal{R}} v$, then $m = n$ and $u = v$.*

Lemma 8.1 allows us to recast innermost rewriting into an operational small-step semantics instrumented with resource counters, cf. Figure 8.1. Its definition is suited to the proof of the soundness Theorem 9.11. The transitive closure of the judgement $\vdash^{m} \langle s,\sigma\rangle \to \langle t,\tau\rangle$ is defined as follows, where σ,ρ,τ denote (normalised) substitutions.

- $\vdash^{m} \langle s,\sigma\rangle \to^{+} \langle t,\tau\rangle$ if $\vdash^{m} \langle s,\sigma\rangle \to \langle t,\tau\rangle$ $(m \in \{0,1\})$
- $\vdash^{m_1+m_2} \langle s,\sigma\rangle \to^{+} \langle u,\rho\rangle$ if $\vdash^{m_1} \langle s,\sigma\rangle \to \langle t,\tau\rangle$ and $\vdash^{m_2} \langle t,\tau\rangle \to^{+} \langle u,\rho\rangle$.

The next proposition clarifies that the introduced operational small-step semantics faithfully represents the "standard" semantics of rewriting. We emphasise that we do *not* assume termination of the underlying TRS $\mathcal{R}$.

Proposition 8.2. *Let f be a defined function symbol of arity n, σ be a substitution, v a value and τ an extension of σ. Then $D\colon t\sigma \xrightarrow{\mathrm{i}}{}^{m}_{\mathcal{R}} u\tau$ iff $\vdash^{m} \langle t,\sigma\rangle \to^{+} \langle u,\tau\rangle$.*

$$\frac{x\sigma = v}{\mid\!\!\overset{0}{\vdash} \langle x, \sigma\rangle \to \langle v, \sigma\rangle} \qquad \frac{c \in \mathcal{C} \quad x_1\sigma = v_1 \quad \cdots \quad x_n\sigma = v_n}{\mid\!\!\overset{0}{\vdash} \langle c(x_1, \ldots, x_n), \sigma\rangle \to \langle c(v_1, \ldots, v_n), \sigma\rangle}$$

$$\frac{\forall i\colon v_i \in \mathcal{V}\mathsf{al} \quad \rho = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \quad f \text{ defined, all } x_i \text{ fresh}}{\mid\!\!\overset{0}{\vdash} \langle f(v_1, \ldots, v_n), \sigma\rangle \to \langle f(x_1, \ldots, x_n), \sigma \uplus \rho\rangle}$$

$$\frac{f(l_1, \ldots, l_n) \to r \in \mathcal{R} \quad \forall i\colon x_i\sigma = l_i\tau}{\mid\!\!\overset{1}{\vdash} \langle f(x_1, \ldots, x_n), \sigma\rangle \to \langle r, \sigma \uplus \tau\rangle} \qquad \frac{\mid\!\!\overset{1}{\vdash} \langle t_i, \sigma\rangle \to \langle u, \sigma'\rangle}{\mid\!\!\overset{1}{\vdash} \langle f(\ldots, t_i, \ldots), \sigma\rangle \to \langle f(\ldots, u, \ldots), \sigma'\rangle}$$

Figure 8.1: Operational Small-Step Semantics

Proof. In the proof of the direction form left to right, we employ induction on the length of the derivation D while for the opposite direction we employ induction on the derivation of $\mid\!\!\overset{m}{\vdash} \langle t, \sigma\rangle \to^+ \langle u, \tau\rangle$. □

Best-Case Lower Bounds

Essentially the cost measure for the best case w.r.t. a relation $\to$ and term t is the minimal derivation length to a normal form. However this intuitive definition excludes the case of non-terminating TRSs. We have chosen the following, slightly more technical definition. Recall that $|D|$ denotes the number of (innermost) steps in a rewrite derivation D.

Definition 8.3. The cost measure for the best case w.r.t. a relation $\to$ and term t is defined as follows: $\mathsf{sp}(t, \to) := \inf\{|D| \mid D\colon t \to^* v,\ v \in \mathcal{V}\mathsf{al}\}$.

The worst-case runtime complexity of a TRS is defined as the maximal derivation length to normal form as a function of the term size of the starting term, cf. [19]. The next example clarifies that this simple definition is not suitable for the definition of *best-case complexity*.

Example 8.4. Consider the following simple TRS $\mathsf{mult1}$, encoding an (inefficient) variant of multiplication.

$$\begin{aligned} 1\colon &\quad \mathsf{0} + y \to y & 3\colon &\quad \mathsf{0} \cdot y \to \mathsf{0} \\ 2\colon &\quad \mathsf{s}(x) + y \to \mathsf{s}(x + y) & 4\colon &\quad \mathsf{s}(x) \cdot y \to x \cdot y + y \end{aligned}$$

The worst-case runtime complexity of $\mathsf{mult1}$ is cubic in n, which is due to the recursive call of multiplication in the first (recursive) argument of addition. On the other hand, the best-case complexity of multiplication is linear in the size of the term $t_n := \mathsf{s}^n(\mathsf{0}) \cdot \mathsf{0}$, as $\mathsf{sp}(t_n, \to)$ is linear in n. □

We define the *best-case complexity* of a TRS $\mathcal{R}$ as the best-case complexity of a single *fixed* function, which represents the main function of $\mathcal{R}$. Furthermore, we express the complexity as a multi-dimensional function of the sizes of the *arguments* of main. Finally, we parametrise the definition in a suitable chosen *size measure*. All induced possible starting terms are embodied in the set S, consisting only of basic terms.

Definition 8.5. We define the *best case complexity function* bc with respect to a relation $\to$, a set of starting terms S in n arguments and a size measure $\|\cdot\|$, as follows: $\mathsf{bc}(\to, S, m_1, \ldots, m_n) := \inf\{\mathsf{sp}(f(\vec{v}), \to) \mid f(\vec{v}) \in S, \|v_i\| \geqslant m_i\}$.

If not mentioned otherwise, we set $\|t\| := |t|$, for any term t. Let $\mathcal{R}$ be a TRS whose main defined function is denoted as main and has n arguments.

Definition 8.6. We define the multi-dimensional *best case complexity* of $\mathcal{R}$ with respect to the function main, as follows: $\mathsf{bc}_{\mathcal{R}}(\vec{m}) := \mathsf{bc}(\to_{\mathcal{R}}, S, \vec{m})$.

We tacitly assume the existence of an infinite set of starting terms S with arguments of unbounded size, that is, for each number n there exist infinitely many terms t, such that $\|t\| \geqslant n$. It suffices to assume that there exists at least one argument position i of the main function such that for any number n, $\mathsf{main}(v_1, \ldots, v_k) \in S$ and $\|v_i\| \geqslant n$. Otherwise, we can principally only derive a trivial constant lower bound. In the sequel of the thesis, we fix $S := \{\mathsf{main}(v_1, \ldots, v_n) \mid v_i \text{ a value}\}$.

Example 8.7 (continued from Example 7.1)**.** Let us consider our motivating example $\mathsf{reverse}$. The main function of the program is the $\mathsf{reverse}$ function and thus the set of starting terms S is suitably defined as follows: $S := \{(x \sharp \cdot)^n(\mathsf{nil}) \mid n \geqslant 0\}$. Clearly for any n there exist infinitely many basic terms $t \in S$, such that $\|t\| \geqslant n$. Then $\mathsf{bc}_{\mathsf{reverse}}(n)$ is bounded from below by a quadratic function in the length of the input list. □

Let $f\colon \mathbb{N}^m \to \mathbb{N}$ and $g\colon \mathbb{N}^n \to \mathbb{N}$ be monotone functions of arity m and n respectively. Suppose $m \leqslant n$. We say that f is *covered* by g as a complexity function, denoted as $f \leqslant g$, if f is asymptotically bounded by the restriction of g to m arguments. I.e., for $m = n$, there exists a threshold N, such that for all $x_1, \ldots, x_n \geqslant N$, we have: $f(x_1, \ldots, x_n) \leqslant c \cdot g(x_1, \ldots, x_n)$, where $c \in \mathbb{R}^+$. Furthermore, for $m < n$, we define a restriction g' on a selection of argument positions $i_1, \ldots, i_m$, where $1 \leqslant i_1 < \ldots i_m \leqslant n$, as follows $g'(x_{i_1}, \ldots, x_{i_m}) := g(\ldots, x_{i_1}, \ldots, x_{i_m}, \ldots)$. The missing arguments are suitably set to zero. Then f is *covered* by g, if there exists a threshold N, such that for all $x_1, \ldots, x_n \geqslant N$, we have $f(x_1, \ldots, x_m) \leqslant c \cdot g'(x_1, \ldots, x_m)$, where $c \in \mathbb{R}^+$.

Definition 8.8. A complexity function $f\colon \mathbb{N}^n \to \mathbb{N}$ is a *(best case) lower bound function* on $\mathcal{R}$ if for all i, there exists a constant N_i, such that for all $m_i \geqslant N_i$, we have $f(m_i, \ldots, m_n) \leqslant c \cdot \mathsf{bc}_{\mathcal{R}}(m_1, \ldots, m_n)$, for some constant $c \in \mathbb{R}^+$. We say that f is *asymptotically optimal*, if there exists no lower bound function which covers it.

We observe that $\Omega(1)$ is the trivial lower bound and that we can employ standard notation for asymptotic lower bound, for example $\Omega(n^k)$, in the usual way. Let f be function in n arguments then by definition f covers any restriction with less arguments.

In concluding the section, we emphasise that our analysis is restricted to *completely defined* TRSs. While natural in programming, completely defined TRSs are unnatural in rewriting and contradict the aspect of generalisation. The next example, clarifies the need for this restriction in our setting.

Example 8.9. Consider the following simple TRS pairs, generating a paired list from the input list. The example is due to Hoffmann [20].

$$\begin{aligned}
\mathsf{nil} \mathbin{@} y &\to y & (n \sharp x) \mathbin{@} y &\to n \sharp (x \mathbin{@} y)\\
\mathsf{attach}(x, \mathsf{nil}) &\to \mathsf{nil} & \mathsf{attach}(x, y \sharp ys) &\to \mathsf{pair}(x, y) \sharp \mathsf{attach}(x, ys)\\
\mathsf{pairs}(\mathsf{nil}) &\to \mathsf{nil} & \mathsf{pairs}(x \sharp xs) &\to \mathsf{attach}(x, xs) \mathbin{@} \mathsf{pairs}(xs))
\end{aligned}$$

□

The TRS pairs is non-ambiguous in the strong sense of Lemma 8.1. Hence best-case and worst-case complexities ought to be equal. However, the TRS is not completely defined. In particular $\mathsf{pairs}(\mathsf{pair}(\mathsf{nil}, \mathsf{nil}))$ is a normal form. Thus, there exists an infinite set of starting terms such that the best-case complexity would be constant with respect to S.

9 Best-Case Amortised Analysis

In this section we establish a novel amortised best case resource analysis for TRS, which is based on the potential method and coached in an inference system. In order to obtain an automated procedure, we coach amortised analysis into an inference system, which is modelled according to a type system. As our studied programs (term rewrite systems) are untyped, the "type system" focuses on assigning suitable resource annotations to function arguments and function results. In the following we might redefine important definitions of part one, in particular Section 2, to ensure a holistic section. Already seen definitions, that is completely unchanged definitions of part one, are marked by $\star$ and can thus safely be skipped. Recall that in this part there are no constructor-like symbols, therefore many definitions are only slightly different from their counterparts in part one. The here presented method is restricted to left-linear constructor TRS. Note that this restrictions are quite natural in functional programming.

Definition 9.1 ($\star$)**.** A *resource annotation* $\vec{p}$ is a vector $\vec{p} = (p_1, \ldots, p_k)$ over non-negative rational numbers, typically natural numbers. The vector $\vec{p}$ is also simply called an *annotation*.

Definition 9.2 ($\star$)**.** Let f be a function symbol of arity n. We annotate the arguments and results of f by *resource annotations*. A (resource) annotation for f, decorated with $k \in \mathbb{Q}^+$, is denoted as $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$. The set of annotations is denoted $\mathcal{F}_{\mathsf{pol}}$.

We lift signatures $\mathcal{F}$ to *annotated signatures* $\mathcal{F} \colon \mathcal{C} \cup \mathcal{D} \to (\mathcal{P}(\mathcal{F}_{\mathsf{pol}}) \setminus \varnothing)$ by mapping a function symbol to a non-empty set of resource annotations. Hence for any function symbol we allow multiple types. In the context of operators this is also referred to as *resource polymorphism*. The inference system, presented below, mimics a type system, where the provided annotations play the role of types. If the annotation of a constructor symbol f results in $\vec{q}$, there must only be exactly one declaration of the form $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q}$ in $\mathcal{F}(f)$, that is, the annotation has to be *unique*. Moreover, constructor annotations are to satisfy the *superposition principle*: If constructor c admits the annotations $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{p} \vec{q}$, $[\vec{p'_1} \times \cdots \times \vec{p'_n}] \xrightarrow{p'} \vec{q'}$, then c also has the annotations $[\lambda\vec{p_1} \times \cdots \times \lambda\vec{p_n}] \xrightarrow{\lambda p} \lambda\vec{q}$, $\lambda \geqslant 0$, $\lambda \in \mathbb{N}$ and $[\vec{p_1} + \vec{p'_1} \times \cdots \times \vec{p_n} + \vec{p'_n}] \xrightarrow{p+p'} \vec{q} + \vec{q'}$. In view of superposition and uniqueness, the annotations of a given constructor are determined, once we fix the annotated types for result annotations of the form $(0, \ldots, 0, 1)$. For notational convenience we often write $f \colon [\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{p} \vec{q}$ instead of $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{p} \vec{q} \in \mathcal{F}(f)$. An annotated signature $\mathcal{F}$ is simply called *signature*.

Example 9.3 (continued from Example 8.7)**.** We consider the signatures for the constructor symbols $\mathcal{C} = \{\sharp, \mathsf{nil}\}$ in the TRS $\mathsf{reverse}$: $\sharp \colon \{[0 \times (k_1 + k_2, k_2)] \xrightarrow{k_1+k_2} (k_1, k_2) \mid$

$$\frac{f \in \mathcal{C} \cup \mathcal{D} \quad [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)}{x_1{:}\,\vec{p}_1, \ldots, x_n{:}\,\vec{p}_n \mid^{k} f(x_1, \ldots, x_n){:}\,\vec{q}}\ (\mathsf{app}) \qquad \frac{\Gamma \mid^{k} t{:}\,\vec{q} \quad k' \leqslant k}{\Gamma \mid^{k'} t{:}\,\vec{q}}\ (\mathsf{w}_1)$$

$$\frac{\begin{array}{ll}\text{all } x_i \text{ are fresh} & k = \sum_{i=0}^{n} k_i \\ x_1{:}\,\vec{p}_1, \ldots, x_n{:}\,\vec{p}_n \mid^{k_0} f(x_1, \ldots, x_n){:}\,\vec{q} & \Gamma_1 \mid^{k_1} t_1{:}\,\vec{p}_1 \ \cdots \ \Gamma_n \mid^{k_n} t_n{:}\,\vec{p}_n\end{array}}{\Gamma_1, \ldots, \Gamma_n \mid^{k} f(t_1, \ldots, t_n){:}\,\vec{q}}\ (\mathsf{comp})$$

$$\frac{\Gamma \mid^{k} t{:}\,\vec{q}}{\Gamma, x{:}\,0 \mid^{k} t{:}\,\vec{q}}\ (\mathsf{w}_4) \qquad \frac{\Gamma, x{:}\,\vec{r}, y{:}\,\vec{s} \mid^{k} t[x, y]{:}\,\vec{q} \quad \Upsilon(\vec{q} \mid \vec{r}, \vec{s}) \quad x,\, y \text{ are fresh}}{\Gamma, z{:}\,\vec{q} \mid^{k} t[z, z]{:}\,\vec{q}}\ (\mathsf{share})$$

$$\frac{\Gamma, x{:}\,\vec{r} \mid^{k} t{:}\,\vec{q} \quad \vec{p} \leqslant \vec{r}}{\Gamma, x{:}\,\vec{p} \mid^{k} t{:}\,\vec{q}}\ (\mathsf{w}_2) \qquad \frac{}{x{:}\,\vec{q} \mid^{0} x{:}\,\vec{q}}\ (\mathsf{var}) \qquad \frac{\Gamma \mid^{k} t{:}\,\vec{s} \quad \vec{s} \leqslant \vec{q}}{\Gamma \mid^{k} t{:}\,\vec{q}}\ (\mathsf{w}_3)$$

Figure 9.1: Inference System for Term Rewrite Systems.

$k_1, k_2 \geqslant 0\}$, $\mathsf{nil}\colon \{[] \xrightarrow{0} (k_1, k_2) \mid k_1, k_2 \geqslant 0\}$. Uniqueness of the annotated signatures is immediate. For any fixed numbers k_1, k_2 acting as resulting annotation of the constructors $\sharp$ and nil respectively, there exists only the signature with resulting annotations (k_1, k_2).

For superposition of $\sharp$ consider the base vectors $[0 \times (1, 0)] \xrightarrow{1} (1, 0)$ and $[0 \times (1, 1)] \xrightarrow{1} (0, 1)$, that is setting $k_1 = 1$ and $k_2 = 0$, and vice versa. Thus, superposition holds, as the signature $[0 \times (k_1 + k_2, k_2)] \xrightarrow{k_1+k_2} (k_1, k_2)$ is obtained by multiplying the base vectors with the scalars k_1 and k_2 respectively, and then adding them. □

Definition 9.4. Let $v = c(v_1, \ldots, v_n) \in \mathcal{V}\mathsf{al}$ and let $\vec{q}$ be a resource annotation. The *potential* of v under $\mathcal{C}$, written $\Phi(v{:}\,\vec{q})$, is defined recursively by:

$$\Phi(v{:}\,\vec{q}) := p + \Phi(v_1{:}\,\vec{p}_1) + \cdots + \Phi(v_n{:}\,\vec{p}_n)\ ,$$

whenever $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{p} \vec{q} \in \mathcal{F}(c)$.

The *sharing relation* $\Upsilon(\vec{p} \mid \vec{p_1}, \vec{p_2})$ holds if $\vec{p_1} + \vec{p_2} = \vec{p}$.

Lemma 9.5 ([28])**.** *If $\Upsilon(\vec{p} \mid \vec{p_1}, \vec{p_2})$ then $\Phi(v{:}\,\vec{p}) = \Phi(v{:}\,\vec{p_1}) + \Phi(v{:}\,\vec{p_2})$ holds for any value of annotation A. Furthermore, if $\vec{p} \geqslant \vec{q}$, then $\Phi(v{:}\,\vec{p}) \geqslant \Phi(v{:}\,\vec{q})$.*

A *(variable) context* is a mapping from variables $\mathcal{V}$ to annotations. Contexts are denoted by upper-case Greek letters and depicted as sequence of pairs $x{:}\,\vec{q}$ of variables and annotations. Our potential based amortised analysis is coached in an inference system whose rules are given in Figure 9.1. Let t be a term and $\vec{q}$ a resource annotation. The inference system derives judgements of the form $\Gamma \mid^{k} t{:}\,\vec{q}$, where Γ is variable context and $k \in \mathbb{Q}^+$ denotes the amortised costs at most required to evaluate t.

Furthermore, we define a subset of the inference rules, free of weakening rules, dubbed the *footprint* of the judgement, denoted as $\Gamma \mid^{k}_{\mathsf{fp}} t{:}\,\vec{q}$. Thus, for the footprint we only

consider the the inference rules: $(\mathsf{app}), (\mathsf{comp}), (\mathsf{share}), (\mathsf{var})$. Occasionally we omit the amortised costs from both judgements using the notations $\Gamma \vdash t\colon \vec{q}$ and $\Gamma \vdash_{\mathsf{fp}} t\colon \vec{q}$.

A substitution is called *consistent with* Γ if for all $x \in \mathsf{dom}(\sigma)$, if $\Gamma \vdash x\colon \vec{q}$, then $\Gamma \vdash x\sigma\colon \vec{q}$. Recall that substitutions are assumed to be normalising. Let Γ be a context and let σ be a substitution consistent with Γ. Then $\Phi(\sigma\colon\Gamma) := \sum_{x\in\mathsf{dom}(\Gamma)} \Phi(x\sigma\colon\Gamma(x))$.

Definition 9.6. Let $f(l_1,\dots,l_n) \to r$, $n \geqslant 1$, be a rule in the TRS $\mathcal{R}$. Further suppose $f\colon [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q}$ is a resource annotation for f, let $\mathcal{V}\mathsf{ar}(l_i) = \{y_{i_1},\dots,y_{i_m}\}$, and let $\biguplus_{i=1}^n \mathcal{V}\mathsf{ar}(l_i) = \{y_1,\dots,y_m\}$. The potential *freed* by the rule is a pair consisting of a variable context $y_1\colon\vec{r}_1,\dots,y_l\colon\vec{r}_l$ and an amortised cost ℓ, such that $y_{i_1}\colon\vec{r}_{i_1},\dots,y_{i_m}\colon\vec{r}_{i_m} \vdash_{\mathsf{fp}}^{\ell_i} l_i\colon\vec{p}_i$, whence $\ell := \sum_{i=1}^n \ell_i$ and $\biguplus_{i=1}^n\{\vec{r}_{i_1},\dots,\vec{r}_{i_m}\} = \{\vec{r}_1,\dots,\vec{r}_m\}$.

Based on Definition 9.6 we can succinctly define resource boundedness of a TRS. The definition constitutes an analogue of Definition 11 in [28] with respect to best-case complexity.

Definition 9.7. Let $\mathcal{R}$ be a TRS, let $\mathcal{F}$ be a signature and let $f \in \mathcal{F}$. An annotation $[\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f)$ is called *resource bounded* if for any rule $f(l_1,\dots,l_n) \to r \in \mathcal{R}$, we have $y_1\colon\vec{r}_1,\dots,y_l\colon\vec{r}_l \vdash_{\mathsf{fp}}^{k+\ell-K^{\mathrm{rule}}} r\colon\vec{q}$ where $y_1\colon\vec{r}_1,\dots,y_l\colon\vec{r}_l$ and ℓ are freed by the rule if $n \geqslant 1$ or $\ell = 0$ otherwise. Furthermore, K^{rule} denotes the cost of the application of the rule $f(l_1,\dots,l_n) \to r$, which defaults to 1. We call an annotation *cost-free* resource bounded, if the cost K^{rule} is always set to zero.

A function symbol f is called *(cost-free) resource bounded* if any resource annotation in $\mathcal{F}(f)$ is (cost-free) resource bounded. Finally, $\mathcal{R}$ is called *resource bounded*, or simply *bounded* if any $f \in \mathcal{F}$ is resource bounded.

In a nutshell, the method works as follows: Suppose the judgement $\Gamma \vdash^{k'} t\colon\vec{q}$ is derivable and suppose σ is consistent with Γ. The constant k' is a lower bound to the amortised cost required for reducing t to normal form. Below we will prove that the derivation length $|D|$ of any derivation D of $t\sigma$ to a normal form is bounded by the difference in the potential before and after the evaluation plus k'. Thus if the sum of the potential of the arguments of $t\sigma$ is in $\Omega(n^k)$, where n is the size of the arguments, then the $\mathsf{bc}_{\mathcal{R}}$ lies in $\Omega(n^k)$. Again k represents the length of the annotations.

More precisely, consider the composition rule (comp). First note that this rule is only applicable if $f(t_1,\dots,t_n)$ is linear, which can always be obtained by the use of the sharing rule (share). Now the rule embodies that the amortised costs k required to evaluate $t\sigma$ can be split into those costs k_i $(i \geqslant 1)$ required for the normalisation of the arguments and the cost k_0 of the evaluation of the operator f. Furthermore the potential provided in the context $\Gamma_1,\dots,\Gamma_n$ is suitably distributed. Finally the potential which remains after the evaluation of the arguments is made available for the evaluation of the operator f.

We explain the use of the inference system on (part of) the motivating example.

Example 9.8 (continued from Example 9.3)**.** In addition to the above given signatures for the constructor symbols, we consider the following signatures for the operators in

$$\dfrac{\dfrac{\sharp\colon [0\times 0]\xrightarrow{0} 0}{z_1\colon 0, z_2\colon 0 \mid\!\!\overset{0}{\vdash} z_1 \sharp z_2\colon 0}\ (\mathsf{app}) \quad \dfrac{}{x\colon 0 \mid\!\!\overset{0}{\vdash} x\colon 0}\ (\mathsf{var}) \quad \dfrac{@\colon [1\times 0]\xrightarrow{1} 0}{xs\colon 1, ys\colon 0 \mid\!\!\overset{1}{\vdash} xs @ ys\colon 0}\ (\mathsf{app})}{x\colon 0, xs\colon 1, ys\colon 0 \mid\!\!\overset{1}{\vdash} x \sharp (xs @ ys)\colon 0}\ (\mathsf{comp})$$

Figure 9.2: Resource boundedness of rewrite rule $(x \sharp xs) @ ys \to x \sharp (xs @ ys)$.

$\mathsf{reverse}$. $@\colon [1\times 0]\xrightarrow{1} 0$ and $\mathsf{reverse}\colon [(1,1)]\xrightarrow{1} (0,0)$. We focus on verifying resource boundedness for the rewrite rule $(x \sharp xs) @ ys \to x \sharp (xs @ ys)$ defining concatenation. Thus, we need to derive the judgement $x\colon 0, xs\colon 1, ys\colon 0 \mid\!\!\overset{1}{\vdash} x \sharp (xs @ ys)\colon 0$. Note that the context $xs\colon 1$ and amortised cost 1 is freed by the rule (see Definition 9.6). The derivation of this judgement is given in Figure 9.2. The base cases are left for the reader. The proof of boundedness of $\mathsf{reverse}$ is postponed to Section 10. □

Let t be a term and let σ be a substitution. The next technical lemma shows that the potential of the instance $t\sigma$ equals the potential of the term t plus the potential of the substitution σ. Here equality holds due as we do not allow weakenings.

Lemma 9.9. *Suppose t is a constructor term, not necessarily ground. If $\Gamma \mid\!\!\overset{k}{\vdash_{\mathsf{fp}}} t\colon \vec{q}$, then $\Phi(t\sigma\colon \vec{q}) = \Phi(\sigma\colon \Gamma) + k$.*

We state the following substitution lemma. The lemma follows by simple induction on t.

Lemma 9.10. *Let Γ be a context and let σ be a substitution consistent with Γ. Then $\Gamma \mid\!\!\vdash t\colon \vec{q}$ implies $\mid\!\!\vdash t\sigma\colon \vec{q}$.*

We arrive at the soundness theorem.

Theorem 9.11 (Soundness)**.** *Let $\mathcal{R}$ be a resource bounded TRS and σ a normalised substitution consistent with Γ, where Γ denotes a variable context. Suppose $\Gamma \mid\!\!\overset{k}{\vdash} t\colon \vec{q}$ and $\mid\!\!\overset{m}{\vdash} \langle t, \sigma\rangle \to \langle u, \tau\rangle$, where τ extends σ and $K \in \{0, 1\}$. Then there exists a context Δ such that $\Delta \mid\!\!\overset{\ell}{\vdash} u\colon \vec{q}$ is derivable and $\Phi(\sigma\colon \Gamma) + k - \Phi(\tau\colon \Delta) - \ell \leqslant m$ holds.*

Proof. Suppose Π derives $\Gamma \mid\!\!\overset{p}{\vdash} t\colon \vec{q}$, while Ξ derives $\mid\!\!\overset{K}{\vdash} \langle t, \sigma\rangle \to \langle u, \tau\rangle$. The proof proceed by in induction on Ξ with side-induction on Π. □

We rephrase the result of Theorem 9.11 in a more directly applicable form.

Corollary 9.12. *Let $\mathcal{R}$ be a resource bounded TRS with main function main returning the annotation 0 and let $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ be a normalising substitution consistent with Γ, where $\Gamma = x_1\colon \vec{p}_1, \ldots, x_n\colon \vec{p}_n$ denotes a variable context. Suppose further $x_1\colon \vec{p}_1, \ldots, x_n\colon \vec{p}_n \mid\!\!\overset{k}{\vdash} \mathsf{main}(x_1, \ldots, x_n)\colon \vec{q}$. Then $\Phi(\sigma\colon \Gamma) + k \leqslant \mathsf{bc}_{\mathcal{R}}(|v_1|, \ldots, |v_n|)$.* □

The next observation follows by straightforward induction on v, employing Lemma 9.5.

Lemma 9.13. *Suppose all annotations are vectors of length* 1*, that is, positive rational numbers. Suppose further, for all* $c \in \mathcal{C}$ *and all annotated signatures* $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q} \in \mathcal{F}(c)$*,* $p_i \geqslant q$ *holds for all* $i = 1, \ldots, n$*. Finally, suppose* $k \geqslant q$*. Then for all values* v *and all non-zero annotation* q*, we have* $\Phi(v\colon q) \geqslant q \cdot |v|$*.*

Corollary 9.14. *Let* $\mathcal{R}$ *be a resource bounded TRS, whose* main *function returns the annotation* 0*. Furthermore, suppose the conditions of Lemma 9.13 are fulfilled. Moreover,* $x_1\colon \vec{p}_1, \ldots, x_m\colon \vec{p}_m \vdash^{k} \mathsf{main}(x_1, \ldots, x_m)\colon \vec{q}$ *is derivable. Furthermore suppose there exists one argument of* main*, such that* $\vec{p}_i \neq 0$*. Then* $\mathsf{bc}_{\mathcal{R}}(|v_1|, \ldots, |v_m|) \in \Omega(v_i)$*.*

Proof. By assumption and due to Corollary 9.14 we have that $\Phi(\sigma\colon \Gamma) + k \leqslant \mathsf{bc}_{\mathcal{R}}(|v_1|, \ldots, |v_n|)$. In conjunction with Lemma 9.13, the inequality yields that there exists a constant $c \in \mathbb{N}$ such that $\mathsf{bc}_{\mathcal{R}}(|v_1|, \ldots, |v_n|) \geqslant q \cdot |v_i|$. □

Example 9.15. Consider the first two rules of TRS mult1, implementing the operator $+$. We denote this part as $\mathcal{R}_{add}$. (i) $\mathsf{0} + y \to y$, and (ii) $\mathsf{s}(x) + y \to \mathsf{s}(x + y)$. $\mathcal{R}_{add}$ requires linearly many steps in the size of first argument given. The TRS is resource bounded with respect to the signatures $+\colon [1 \times 0] \xrightarrow{1} 0$, $\mathsf{s}\colon \{[n] \xrightarrow{n} n \mid n \geqslant 0\}$ and $\mathsf{0}\colon \{[] \xrightarrow{0} n \mid n \geqslant 0\}$. This signature fulfils the requirements of Corollary 9.14, which can be verified fully automatically by our prototype. □

Example 9.16. However, let us consider the simple TRS $\mathcal{R}_{app}$, a subset of the running example reverse: (i) $\mathsf{nil} \mathbin{@} ys \to ys$, and (ii) $(x \mathbin{\sharp} xs) \mathbin{@} ys \to x \mathbin{\sharp} (xs \mathbin{@} ys)$. Clearly $\mathcal{R}_{app}$ requires at most (and at least) linearly many steps in the *length* of the list xs given as first argument. Conclusively the TRS is resource bounded with respect to the following signature

$$\sharp\colon \{[0 \times n] \xrightarrow{n} n \mid n \geqslant 0\} \qquad\qquad @\colon [1 \times 0] \xrightarrow{1} 0\ .$$

Nonetheless, this signature does not fulfil the conditions of Corollary 9.14 as the first argument of the constructor $\sharp$ violates Lemma 9.13. □

The above example motivates the following definition, refining the notion of size of a data structure to a less restrictive measure.

Definition 9.17. Let $\pi\colon \mathcal{C} \to \mathbb{N}$ denote a mapping from constructor symbols to argument positions. Suppose for each $f \in \mathcal{F}$ there exists a designated argument position $\pi(f)$. Then the *trace depth* of a term t is defined as follows:

$$\mathsf{tdepth}_{\pi}(t) := \begin{cases} 0 & t \text{ is a constant or a variable} \\ 1 + \mathsf{tdepth}_{\pi}(t_i) & t = f(t_1, \ldots, t_n), \text{ and } \pi(f) = i\ . \end{cases}$$

If the projection function π is clear from context, we write $\mathsf{tdepth}(t)$ instead of $\mathsf{tdepth}_{\pi}(t)$.

The next lemma suitably bounds the potential of a value v, taking into account the notion of *trace*. The proof proceeds by induction on v.

Lemma 9.18. *Suppose for all constructor symbols c of non-zero arity, there exists a designated argument $\pi(c)$ such that for all $[\vec{p_1} \times \cdots \times \vec{p_n}] \xrightarrow{k} \vec{q} \in \mathcal{F}(c)$, where $\vec{q} \neq 0$, we have $k \geqslant 1$ and there exists $\vec{r} \in \mathcal{A}$ with $\vec{p_{\pi(c)}} \geqslant \vec{q} + \vec{r}$ and $|\vec{r}| \geqslant |\vec{q}| - 1$. For all values v and all non-zero annotations $\vec{q}$: $\Phi(v{:}\,\vec{q}) \geqslant \binom{\mathsf{tdepth}_\pi(v)}{|q|}$.*

Finally, we obtain the strongest result of our analysis. The corollary is a direct consequence of the lemma.

Corollary 9.19. *Let $\mathcal{R}$ be a resource bounded TRS, whose* main *function returns the annotation* 0*. Furthermore, suppose the conditions of Lemma 9.18 are fulfilled. Moreover, $x_1{:}\,\vec{p}_1, \ldots, x_m{:}\,\vec{p}_m \mid^k \mathsf{main}(x_1, \ldots, x_m){:}\,\vec{q}$ is derivable. Suppose there exists one argument of* main*, such that $\vec{p}_i \neq 0$. Then $\mathsf{bc}_\mathcal{R}(\mathsf{tdepth}\, v_1, \ldots, \mathsf{tdepth}\, v_m) \in \Omega(n^d)$, where $n = \mathsf{tdepth}\, v_i$ and d denotes the length of the resource annotations employed.*

Proof. Due to the lemma in conjunction with the fact that $\binom{n}{d} \in \Omega(n^d)$. which follows from the following estimation:

$$(\frac{n}{d})^d \leqslant \binom{n}{d} \leqslant (\frac{n \cdot e}{d})^d \,,$$

where e denotes Euler's e and holds for all numbers $n \geqslant 1$ and $d \in \{1, \ldots, n\}$. □

10 Refinements and Implementation

This section introduces and explains additional implementation issues specific to the best case analysis of the prototype implemented in Haskell. For the major explanation of implementation details see Section 4. In short, the implementation details for the collection of signatures and constraints, as well as the and uniqueness of constructors are identical. Furthermore, we also integrated the cost-free function application rule (app_{cf}) given in Figure 10.1. By utilizing this rule we are able to type the running example reverse. In Figure 10.2, the derivation of Rule 4 is presented. The rather simple derivations of the application of Rule 3 for the cost-free signature of reverse, the use of (var) in the cost-free derivation and the verification of the cost-free resource boundedness of @ are left over to the reader. As we have found a typing using vectors with length 2 we can conclude that reverse has a quadratic best case lower bound in the length of the input list, cf. Corollary 9.19.

Completely Defined Restriction

The established method only works for a restrictive set of rewrite systems, which is however natural in the context of functional programming. In particular, we require to completely defined, non-overlapping, left-linear, constructor TRSs. While natural in programming, completely defined TRSs are quite unnatural in rewriting and contradict the aspect of generalisation. Hence, our prototype implementation also accepts non completely defined programs as input and elaborates them to completely defined TRSs. This step underapproximates the best-case complexity and is only performed in order to provide a fully automatic method. A more refined analysis would require reachability analysis and is subject to future work.

However, these rules add derivation paths for all cases. Even those which were unintended in the first case. A sound alternative is to suitably restrict the set of starting terms, for example through a reachability analysis [34]. For instance, in Example 8.9 terms including the constructor pair might not be considered as valid starting terms.

$$\frac{\begin{array}{cc} f \in \mathcal{C} \cup \mathcal{D} & y_1\colon\vec{r}_1,\ldots,y_l\colon\vec{r}_l \text{ and } \ell \text{ freed} \\ & \forall\, f(l_1,\ldots,l_n) \to r \in \mathcal{R}\colon\ y_1\colon\vec{r}_1,\ldots,y_l\colon\vec{r}_l \mid^{\vec{k}^{cf}+\ell}_{\mathsf{cf}} r\colon\vec{q} \\ [\vec{p}_1 \times \cdots \times \vec{p}_n] \xrightarrow{k} \vec{q} \in \mathcal{F}(f) & [\vec{p}_1^{\,cf} \times \cdots \times \vec{p}_n^{\,cf}] \xrightarrow{\vec{k}^{cf}} \vec{q}^{\,cf} \in \mathcal{F}^{\,cf}(f) \end{array}}{x_1\colon\vec{p}_1,\ldots,x_n\colon\vec{p}_n \mid^{k+\vec{k}^{cf}} f(x_1,\ldots,x_n)\colon\vec{q}+\vec{q}^{\,cf}}\ (\mathsf{app}_{cf})$$

Figure 10.1: Additional Function Application Inference Rule for Cost-Free Derivation.

$$
\dfrac{
\dfrac{
(\star)\colon
\qquad
\dfrac{
\dfrac{
\dfrac{}{\mathsf{@}_{cf}\colon [1\times 1]\xrightarrow{0} 1}
\qquad
\dfrac{}{\mathsf{reverse}_{cf}\colon [1]\xrightarrow{0} 1}
}{z_1\colon 1, z_2\colon 1 \mid^{0}\!\vdash z_1 \mathbin{@} z_2\colon 1}
\qquad
\dfrac{
\dfrac{}{\sharp_{cf}\colon [0\times 1]\xrightarrow{1} 1}
\qquad
\dfrac{}{\mathsf{nil}_{cf}\colon []\xrightarrow{0} 1}
}{x\colon 0, z_5\colon 1 \mid^{1}\!\vdash x \sharp z_5\colon 1}
\qquad
\dfrac{}{\mid^{0}\!\vdash \mathsf{nil}\colon 1}
}{x\colon 0 \mid^{1}\!\vdash x \sharp \mathsf{nil}\colon 1}\ (\mathsf{comp})
}{xs\colon 1 \mid^{0}\!\vdash \mathsf{reverse}(xs)\colon 1}\ (\mathsf{comp})
}{x\colon 0, xs\colon 1 \mid^{1}\!\vdash \mathsf{reverse}(xs) \mathbin{@} (x \sharp \mathsf{nil})\colon 1}\ \text{Rule 4}
}{
\dfrac{
\mathsf{reverse}\colon [(1,1)]\xrightarrow{1} 0
\qquad
\mathsf{reverse}_{cf}\colon [1]\xrightarrow{0} 1
}{xs\colon (2,1) \mid^{1}\!\vdash \mathsf{reverse}(xs)\colon 1}\ (\mathsf{app}_{cf})
}
$$

$$
\dfrac{
\dfrac{
\dfrac{}{\mathsf{@}\colon [1\times 0]\xrightarrow{1} 0}
}{z_1\colon 1, z_2\colon 0 \mid^{1}\!\vdash z_1 \mathbin{@} z_2\colon 0}
\qquad
\dfrac{
\dfrac{
\dfrac{}{\sharp\colon [0\times 0]\xrightarrow{0} 0}
}{z_3\colon 0, z_4\colon 0 \mid^{0}\!\vdash (z_3 \sharp z_4)\colon 0}
\qquad
\dfrac{}{x\colon 0 \mid^{0}\!\vdash x\colon 0}\ (\mathsf{var})
\qquad
\dfrac{
\dfrac{}{\mathsf{nil}\colon []\xrightarrow{0} 0}
}{\mid^{0}\!\vdash \mathsf{nil}\colon 0}
}{x\colon 0 \mid^{0}\!\vdash x \sharp \mathsf{nil}\colon 0}\ (\mathsf{comp})
}{x\colon 0, xs\colon (2,1) \mid^{2}\!\vdash \mathsf{reverse}(xs) \mathbin{@} (x \sharp \mathsf{nil})\colon 0}\ (\star)\ (\mathsf{comp})
$$

Figure 10.2: Resource boundedness of rewrite rule $\mathsf{reverse}(x \sharp xs) \to \mathsf{reverse}(xs) \mathbin{@} (x \sharp \mathsf{nil})$. All unnamed inference steps are applications of the rule (app). Recall that all type annotations are vectors and can be appended by arbitrary many 0.

Thus, we allow the user to drop the generation of the completely defined constraints. Naturally, this may result in incorrect behaviour, if the missing definitions are intended in the specification.

Size Analysis

To be able to correctly prove a linear best case bound in the size of the arguments of the main function further constraints must be added. The main function must return 0, cf. Corollary 9.14, and at least one annotation of the arguments of the main rule can not be 0. (If no dedicated main function is given, we use the last given inference rule of the input TRS.) Further, the constraints of Lemma 9.13 for constructors are added and all annotations are restricted to scalars. Often a derivation of a TRS does not depend on the size of the input term, but rather on the depth of a specific argument. In, TRS $\mathcal{R}_{app}$ regardless of the size of the second argument, the TRS requires at least linearly many steps in the *length* of the list of the first argument. Thus, specifying a best case bound by the size of the input term hardly makes sense. Therefore, we developed a technique to specify the best case bound by the trace depth of at least one argument.

Trace Depth Analysis

This analysis has the same restriction on the entry point of the TRS as the size analysis, cf. Corollary 9.19. Further, for each constructor a designated argument is selected. This

can either be handled by the SMT solver or manually triggered. All non-zero arity base constructors must have cost $k \geqslant 1$. However, it is also mathematically possible to move the requirement of the costs $k \geqslant 1$ to the constants, such that only leafs in the trace depth are counted. Nonetheless, this alternative method can solve only very few examples and therefore was dropped.

11 Numerical Example

In this section we exemplify the method by applying the tdepth-variation to the already seen term rewrite system mult3 from the TPDB. This TRS was already used in the numerical example of the worst-case analysis to ensure the reader can easily compare the methods. The rules of the TRS are as follows.

$$\begin{array}{ll} 1\colon \quad \mathsf{0} + y \to y & 3\colon \quad \mathsf{0} \cdot y \to \mathsf{0} \\ 2\colon \mathsf{s}(x) + y \to \mathsf{s}(x + y) & 4\colon \mathsf{s}(x) \cdot y \to x + (x \cdot y) \end{array}$$

Remember that this TRS implements a (false) version of multiplication. For the rest of this section we will not make use of the cost-free application rule (app_{cf}) but concentrate on demonstrating the method using the inference rules given in Figure 8.1. Observe that the TRS is a completely defined left-linear constructor TRS.

Our tool derives the following annotations. Recall that functions are monomorphically typed, whereas the constructor symbols are build using base constructor annotations ($\mathsf{0}_1, \mathsf{0}_2$ and $\mathsf{s}_1, \mathsf{s}_2$ respectively) and the implicit filling up with 0s of annotations. Thus, all constructor annotations for e.g. s are linear combinations of the given base constructor annotations $n_1 \cdot \mathsf{s}_1 + \ n_2 \cdot \mathsf{s}_2$ with fresh variables n_1 and n_2.

$$\begin{array}{ll} +\colon [1 \times 0] \xrightarrow{1} 0 & \cdot\colon [(1,1) \times 0] \xrightarrow{1} 0 \\ \mathsf{0}_1\colon [] \xrightarrow{0} 1 & \mathsf{0}_2\colon [] \xrightarrow{0} (0,1) \\ \mathsf{s}_1\colon [1] \xrightarrow{1} 1 & \mathsf{s}_2\colon [(1,1)] \xrightarrow{1} (0,1) \end{array}$$

Observe that the resource annotation signature of $\cdot$ is not just a scaled variant of the corresponding worst-case annotation as presented in Section 5. In particular the annotation of the first parameter is different.

As demanded by Definition 9.7 all rules must be resource bounded. Therefore, for rule one the judgement $y\colon 0 \vdash^{1-1+0} y\colon 0$ must be derivable. For the first argument the typing $\mathsf{0}\colon 1$ with freed cost 0 is used, thus $\vdash^{0}_{\mathsf{fp}} \mathsf{0}\colon 1$ has to be derivable as well. Both requirements are met by our annotations as can be seen by the following derivations.

$$\frac{}{y\colon 0 \mid\!\overset{0}{\vdash} y\colon 0}\ (\mathsf{var}) \qquad \frac{\mathsf{0}\colon [] \overset{0}{\to} 1}{\mid\!\overset{0}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon 1}\ (\mathsf{app})$$

Similarly the third rule can be shown to be resource bounded. The required judgements are $y\colon 0 \mid\!\overset{1-1+0}{\vdash} \mathsf{0}\colon 0$ for the rule and $\mid\!\overset{0}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon (1,1)$ for the first argument with the following inference derivations.

$$\frac{\dfrac{\mathsf{0}\colon [] \overset{0}{\to} 0}{\mid\!\overset{0}{\vdash} \mathsf{0}\colon 0}\ (\mathsf{app})}{y\colon 0 \mid\!\overset{0}{\vdash} \mathsf{0}\colon 0}\ (\mathsf{w}_4) \qquad \frac{\mathsf{0}\colon [] \overset{0}{\to} (1,1)}{\mid\!\overset{0}{\underset{\mathsf{fp}}{\vdash}} \mathsf{0}\colon (1,1)}\ (\mathsf{app})$$

We arrive at the recursive rewrite rules, starting with rule two. By using the annotation signatures $+\colon [1 \times 0] \overset{1}{\to} 0$ and $\mathsf{s}\colon [1] \overset{1}{\to} 1$ the judgements for proofing resource boundedness of the second rule are $x\colon 1, y\colon 0 \mid\!\overset{1-1+1}{\vdash} \mathsf{s}(x+y)\colon 0$ and $x\colon 1 \mid\!\overset{1}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon 1$. The following derivations proof resource boundedness of rule two.

$$\frac{\dfrac{\mathsf{s}\colon [0] \overset{0}{\to} 0}{z_1\colon 0 \mid\!\overset{0}{\vdash} \mathsf{s}(z_1)\colon 0}\ (\mathsf{app}) \quad \dfrac{+\colon [1 \times 0] \overset{1}{\to} 0}{x\colon 1, y\colon 0 \mid\!\overset{1}{\vdash} x+y\colon 0}\ (\mathsf{app})}{x\colon 1, y\colon 0 \mid\!\overset{1}{\vdash} \mathsf{s}(x+y)\colon 0}\ (\mathsf{comp}) \qquad \frac{\mathsf{s}\colon [1] \overset{1}{\to} 1}{x\colon 1 \mid\!\overset{1}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon 1}\ (\mathsf{app})$$

For rule four we use the signatures $\cdot\colon [(1,1) \times 0] \overset{1}{\to} 0$ and $\mathsf{s}\colon [(1,1)] \overset{1}{\to} (0,1)$ to gain the judgements $x\colon (2,1), y\colon 0 \mid\!\overset{1-1+2}{\vdash} x + (x \cdot y)\colon 0$ and $x\colon (2,1) \mid\!\overset{2}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}(x)\colon (1,1)$. Note the increasing potential in the parameters of the constructor judgement. This ensues from the best-case analysis. The following inference tree derivations proof resource boundedness of the rewrite rule.

$$\frac{\dfrac{\dfrac{+\colon [1 \times 0] \overset{1}{\to} 0}{z_1\colon 1, z_2\colon 0 \mid\!\overset{1}{\vdash} z_1 + z_2\colon 0}\ (\mathsf{app}) \quad \dfrac{}{x_1\colon 1 \mid\!\overset{0}{\vdash} x_1\colon 1}\ (\mathsf{var}) \quad \dfrac{\cdot\colon [(1,1) \times 0] \overset{1}{\to} 0}{x_2\colon (1,1), y\colon 0 \mid\!\overset{1}{\vdash} x_2 \cdot y\colon 0}\ (\mathsf{app})}{x_1\colon 1, x_2\colon (1,1), y\colon 0 \mid\!\overset{2}{\vdash} x_1 + (x_2 \cdot y)\colon 0}\ (\mathsf{comp})}{x\colon (2,1), y\colon 0 \mid\!\overset{2}{\vdash} x + (x \cdot y)\colon 0}\ (\mathsf{share})$$

$$\frac{\mathsf{s}\colon [(2,1)] \overset{2}{\to} (1,1)}{x\colon (2,1) \mid\!\overset{2}{\underset{\mathsf{fp}}{\vdash}} \mathsf{s}\colon (1,1)}\ (\mathsf{app})$$

Here the (share) rule is applied to split the potential of $x\colon(2,1)$ into the parts $x_1\colon(1,0)$ and $x_2\colon(1,1)$. Further, note that the costs before applying the (comp) rule and the sum of the costs after applying it is equal.

We proved the TRS to be resource bounded. Furthermore, all constructor annotations are fulfilling Lemma 9.18. Note that constructor annotations without parameters are not considered for the lemma and that the base constructors are designed to hold the Lemma. Adding to this, as we set the main function to be the last rule in the TRS, that is rule four, Corollary 9.19 holds. For this observe that the first argument of the rule has a non-zero annotation. Thus, we conclude that the conclude that the TRS mult3 is in $\Omega(n^2)$.

12 Experimental Results

This section presents the main results of the conducted experiments. All experiments were run with a timeout of 60s on a machine with an Intel Core i7-3840QM CPU running at 2.80GHz, and 32GB of RAM. For benchmarking we used a collection consisting of 113 first-order TRSs, representing first-order functional programs [14, 18], transformations from higher-order programs [5], or RaML programs [23]. Finally, we considered additional examples from the TPDB [1]. We emphasise that not all examples are terminating. In Figure 12.1 and Figure 12.2 we summarise our experimental findings up to cubic bounds on the benchmark. CF means cost-free derivations are allowed and Heur specifies that heuristics are used instead of base signatures, cf. [20, 22, 23]. The upper half of the figures represent the number of successfully proven problems and below is a table of the average execution times in seconds listed. In case the tool could not derive a best case complexity it returns constant. The detailed results can be observed here

`http://cl-informatik.uibk.ac.at/software/tct/experiments/arabc/` .

The size method is only able to find 12 problems with a linear best-case bound, whereas tdepth can prove 33. As mentioned above, it is obvious that tdepth is able to prove more results, as the size method uses the size of the input term, whereas the tdepth method measures in the depth of a parameter. The cost-free derivations allow better bounds for few TRSs. Once the heuristics are enabled, the execution time decreases tremendously, but the number of solvable problems decreases as well. Exactly the same characteristics can be seen on the when there are no complete defined requirements. In the following we will investigate some of the programs of the testbed in detail.

mult1. The beforehand presented TRS, cf. Example 8.4, implements multiplication. Our tool can only provide a linear best-case lower bound, where the multiplier is set to zero. In contrast, RaML can automatically derive the correct cubic multivariate lower bound. In order to improve our prototype implementation, we need to incorporate a multivariate amortised analysis, cf. [29]. We leave this to future work.

intlookup and sp1. These TRS are non-terminating. In particular no polynomial can provide an optimal lower bound. Our tool can search for polynomials up to the degree of 7, and for both examples finds solutions with that degree.

[1] See `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB`.

	Number of Successful Analyses				
Result	size	tdepth	tdepth CF	tdepth Heur	tdepth CF H.
constant	89	68	65	83	81
linear	12	33	32	18	18
quadratic		4	9	3	8
cubic		3	5	2	3

	Execution Time (in seconds)				
Result	Size	tdepth	tdepth CF	tdepth Heur	tdepth CF H.
constant	1.19	1.16	3.38	0.74	2.06
linear	0.32	0.53	0.47	0.31	0.34
quadratic		2.08	4.36	0.47	0.39
cubic		3.94	26.54	0.63	0.57

Figure 12.1: Experimental evaluation with elaboration to completely defined TRSs.

reverse. The naive implementation of reverse has a worst-case complexity bound of $\mathsf{O}(n^2)$ as both @ and rev traverse over list once. For the same argument the best-case complexity must be $\Omega(n^2)$, which can be proven by our prototype with cost-free derivation enabled.

rev-foldl. This is an implementation of reverse by folding over the list, and simultaneously prepending the first element of the input list until there are no elements left. Clearly this yields $\Omega(n)$. Again our tool is able to prove that this bound is tight as well.

shuffle. This TRS has been a challenging problem for automation for some time. Nowadays tools like TCT [6] can prove that it has a worst-case complexity of $\mathsf{O}(n^3)$. The amortised analysis is able to provide a best-case lower bound of $\Omega(n^2)$.

#3.42. Given a number n in unary encoding as input, the TRS computes the binary representation $(n)_2$ by repeatedly halving n and computing the last bit. The best (and worst) case runtime complexity of this TRS is linear in n. For this firstly observe that the evaluation of $\mathsf{half}(\mathsf{s}^m(\mathsf{0}))$ and $\mathsf{lastbit}(\mathsf{s}^m(\mathsf{0}))$ requires about m steps in total. Secondly, n is halved in each iteration and thus the number of steps can be estimated by $\sum_{i=0}^{k} 2^i$, where $k := |(n)_2|$. As the geometric sum computes to $2 \cdot 2^k - 1$, the claim for the worst-case upper bound follows. Similarly one observes that any execution of conv needs at least linear runtime. The TRS is not completely defined as conv returns a list, but all rules expect a natural number. It is reasonable to assume that the start terms are only natural numbers and with this assumption the tool proves the tight bound $\Omega(n)$.

	Number of Successful Analyses				
Result	Size	tdepth	tdepth CF	tdepth Heur	tdepth CF H.
constant	87	30	33	68	70
linear	22	73	67	42	37
quadratic		9	16	12	15
cubic		6	5	6	10

	Execution Time (in seconds)				
Result	Size	tdepth	tdepth CF	tdepth Heur	tdepth CF H.
constant	4.26	5.78	6.00	0.43	2.43
linear	0.44	2.28	2.39	0.27	0.29
quadratic		1.01	4.80	0.38	1.49
cubic		2.09	12.12	0.23	0.41

Figure 12.2: Experimental evaluation without elaboration to completely defined TRSs.

quotient. This TRS computes the quotient if given two natural numbers. To make it completely defined rules which handle the cases of division by zero and non-zero remainders have to be added. Thus, the tool returns $\Omega(1)$ as division by 0 will only take one step to evaluate. Under the assumption that the given TRS evaluates to a value, the completely defined constraints can be dropped and the expected outcome of $\Omega(n)$ is returned.

13 Conclusion

In this thesis we have established two novel automated amortised cost analyses for term rewriting. In the first part of the thesis we presented an analysis for worst case runtime, whereas in the second part we investigated best-case lower bounds of first-order TRSs. In doing so we have not only implemented the methods detailed in earlier work [28], but also generalised the theoretical basis considerably. We have provided a prototype implementation and integrated it into T$_C$T.

More precisely, on one hand we have extended the method of worst-case amortised resource analysis to *unrestricted* term rewrite systems, thus overcoming typical restrictions of functional programs like left-linearity, pattern based, non-ambiguity, etc. This extension is non-trivial and generalises earlier results in the literature. Furthermore, we have lifted the method to relative rewriting. The latter is the prerequisite to a *modular* resource analysis, which we have provided through the integration into T$_C$T. The provided integration of amortised resource analysis into T$_C$T has led to an increase in overall strength of the tool (in comparison to the latest version without ARA and the current version of AProVE). Furthermore in a significant amount of cases we could find better bounds than before.

While on the other hand we established the first automated lower bound analysis for the best-case complexity of first-order TRSs. The method is based on an amortised resource analysis coached in a syntax-directed inference system. We have provided ample experimental data to showcase the viability of the approach. In future work we will also integrate this method in T$_C$T and investigate complexity preservation of existing transformations from programming languages in order to provide precise lower bound results with respect to the execution model of the programming languages considered as frontend in T$_C$T.

For part one we want to focus on lifting the provided amortised analysis in two ways. First we want to extend the provided univariate analysis to a multivariate analysis akin the analysis provided in RaML. The theoretical foundation for this has already been provided by Hofmann et al. [29]. However efficient automation of the method proposed in [29] requires some sophistication. Secondly, we aim to overcome the restriction to constant amortised analysis and provide an automated (or at least automatable) method establishing logarithmic amortised analysis. This aims at closing the significant gap of existing methods in contrast to the origin of amortised analysis [40, 41], compare also [35].

In this thesis we showed that amortised resource analysis is a flexible and powerful concept which can be applied to analyse generic computer programs in multiple ways. This allows a single tool to be able to use nearly the same ideas and code for worst-case runtime analyses, as well as best-case complexity analyses. With various applications,e.g. in cyber

security, this increasingly important research topic will provide the required methods and tools for secure and reliable programs in the future.

Bibliography

[1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *JAR*, 46(2), 2011.

[3] E. Albert, S. Genaim, and A. N. Masud. On the inference of resource usage upper and lower bounds. *TOCL*, 14(3):22, 2013.

[4] M. Avanzini and U. D. Lago. Automating sized-type inference for complexity analysis. *PACMPL*, 1(ICFP):43:1–43:29, 2017.

[5] M. Avanzini, U. D. Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *Proc. 20th ICFP*, pages 152–164. ACM, 2015.

[6] M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. 22nd TACAS*, LNCS, pages 407–423, 2016.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[9] S. Debray, P. López-García, M. V. Hermenegildo, and N. Lin. Lower bound cost estimation for logic programs. In *Proc. 1997 LP*, pages 291–305. MIT Press, 1997.

[10] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 163–174. IEEE, 2000.

[11] J. Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902.

[12] A. Flores-Montoya. CoFloCo: System description. In *15th International Workshop on Termination*, page 20.

[13] A. Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *Proc. 21st FM*, volume 9995 of *LNCS*, 2016.

[14] C. Frederiksen. Automatic runtime analysis for first order functional programs. Master's thesis, DIKU, 2002. DIKU TOPPS D-470.

[15] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. Lower bounds for runtime complexity of term rewriting. *JAR*, 59(1):121–163, 2017.

[16] D. Gale, H. W. Kuhn, and A. W. Tucker. Linear programming and the theory of games. *Activity analysis of production and allocation*, 13:317–335, 1951.

[17] 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. *JAR*, 58(1):3–31, 2017.

[18] A. Glenstrup. Terminator ii: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, 1999. Technical Report DIKU-TR-99/8.

[19] N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, number 5195 in LNAI, pages 364–380, 2008.

[20] J. Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[21] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):14, 2012.

[22] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14, 2012.

[23] J. Hoffmann, A. Das, and S. Weng. Towards automatic resource bound analysis for OCaml. In *Proc. 44th POPL*, pages 359–373. ACM, 2017.

[24] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *Proc. 19th ESOP*, volume 6012 of *LNCS*, pages 287–306, 2010.

[25] M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming*, pages 165–179. Springer, 2000.

[26] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th POPL*, pages 185–197. ACM, 2003.

[27] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, pages 22–37. Springer Berlin Heidelberg, 2006.

[28] M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Proc. of Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 272–286, 2014.

[29] M. Hofmann and G. Moser. Multivariate amortised resource analysis for term rewrite systems. In *Proc. 13th TLCA*, volume 38 of *LIPIcs*, pages 241–256, 2015.

[30] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proc. 37th POPL*, pages 223–236. ACM, 2010.

[31] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for resource-bounded computations using amortised analysis. In *Proc. 2nd FM*, volume 5850 of *LNCS*, pages 354–369. Springer Verlag, 2009.

[32] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. Complexity analysis for term rewriting by integer transition systems. In *International Symposium on Frontiers of Combining Systems*, pages 132–150. Springer, 2017.

[33] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *Proc. of the 38th SP*, pages 710–728. IEEE Computer Society, 2017.

[34] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2005.

[35] T. Nipkow. Amortized complexity verified. In *Proc. 6th ITP*, volume 9236 of *LNCS*, pages 310–324, 2015.

[36] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[37] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, 2000.

[38] H. Simões, P. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ACM SIGPLAN Notices*, volume 47, pages 165–176. ACM, 2012.

[39] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. Software Engineering*, volume 252 of *LNI*, pages 101–102, 2016.

[40] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proc. of the 15th STOC*, pages 235–245. ACM, 1983.

[41] R. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth*, 6(2):306–318, 1985.

[42] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracks in Theoretical Computer Science*. Cambridge University Press, 2003.

[43] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, University of Aachen, Department of Computer Science, 2007.

[44] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[45] B. Wegbreit. Verifying program performance. *J. ACM*, 23(4):691–699, 1976.