Automated Amortised Resource Analysis for Term Rewrite Systems*

Georg Moser¹ and Manuel Schneckenreither²

¹ Department of Computer Science, University of Innsbruck, Austria, email: georg.moser@uibk.ac.at
² Department of Information Systems, Production and Logistics Management, University of Innsbruck, Austria, email: manuel.schneckenreither@uibk.ac.at

Abstract. Based on earlier work on amortised resource analysis, we establish a novel automated amortised resource analysis for term rewrite systems. The method is presented in an inference system akin to a type system and gives rise to polynomial bounds on the innermost runtime complexity of the analysed term rewrite system. Our analysis does not restrict the input rewrite system in any way. This facilitates integration in a general framework for resource analysis of programs. In particular, we have implemented the method and integrated it into our tool TCT.

Keywords: analysis of algorithms, amortised complexity, term rewriting, types, automation

The final authenticated version is available online at https://doi.org/10.1007/978-3-319-90686-7_14.

1 Introduction

Amortised resource analysis [1,2] is a powerful method to assess the overall complexity of a sequence of operations precisely. It has been established by Sleator and Tarjan in the context of self-balancing data structures, which sometimes require costly operations that however balance out in the long run.

For automated resource analysis, amortised cost analysis has been in particular pioneered by Hoffmann et al., whose RaML prototype has grown into a highly sophisticated analysis tool for (higher-order) functional programs, cf. [3]. In a similar spirit, resource analysis tools for imperative programs like COSTA [4], CoFloCo [5] and LOOPUS [6] have integrated amortised reasoning. In this paper, we establish a novel automated amortised resource analysis for term rewrite systems (TRSs for short).

Consider the rewrite system $R_1$ in Figure 1 encoding a variant of an example by Okasaki [7, Section 5.2] (see also [8, Example 1]); $R_1$ encodes an efficient

* This research is partly supported by DARPA/AFRL contract number FA8750-17-C-088.
implementation of a queue in functional programming. A queue is represented as a pair of two lists \( \text{que}(f, r) \), 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 (\text{err\_head} or \text{err\_tail}) is raised. To exemplify the physicist’s method of amortised analysis [2] we assign to every queue \( \text{que}(f, r) \) 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. [7]. Thus, based on an amortised analysis, we may deduce the optimal linear runtime complexity for \( R \).

Taking inspirations from [8,9], the amortised analysis is based on the potential method, as exemplified above. It employs the standard (small-step) semantics of innermost rewriting and exploits a footprint relation in order to facilitate the extension to TRSs. For the latter, we suit a corresponding notion of Avanzini et al. [10] to our context. Due to the small-step semantics we immediately obtain an analysis which does not presuppose termination. The incorporation of the footprint relations allows the immediate adaption of the proposed method to general rule-based languages. The most significant extension, however, 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 TCT [11]. 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 paper 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. \cite{8,12}. 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 TCT. We have assessed the viability of the method in context of the TPDB as well as on an independent benchmark.

This paper is structured as follows. In the next section we cover basics. In Section 3 we introduce the inference system and prove soundness of the method. In Section 4 we detail the implementation of the method and remark on challenges posed by automation. Section 5 provides the experimental assessment of the method. Finally we conclude in Section 6, where we also sketch future work. The formal proofs and the full definitions of additional examples have been omitted due to space restrictions. Full details can be found in the second author’s master thesis, cf. \cite{13}.

2 Preliminaries

We assume familiarity with term rewriting \cite{14,15} 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 $\text{Var}(t)$ to denote the set of variables occurring in term $t$. The size $|t|$ of a term is defined as the number of symbols in $t$.

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 $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. We write $\text{dom}(\sigma)$ ($\text{rg}(\sigma)$) to denote the domain (range) of $\sigma$.

Let $\rightarrow \subseteq S \times S$ be a binary relation. We denote by $\rightarrow^+$ the transitive and by $\rightarrow^*$ the transitive and reflexive closure of $\rightarrow$. By $\rightarrow^n$ we denote the $n$-fold application of $\rightarrow$. If $t$ is in normal form with respect to $\rightarrow$, we write $s \rightarrow^1 t$. We say that $\rightarrow$ is well-founded or terminating if there is no infinite sequence $s_0 \rightarrow s_1 \rightarrow \ldots$. It is finitely branching if the set $\{t \mid s \rightarrow t\}$ is finite for each
s ∈ S. For two binary relations →_A and →_B, the relation of →_A relative to →_B is defined by →_A/→_B := →_B^∗ · →_A · →_B^∗.

A rewrite rule is a pair l → r of terms, such that (i) the root symbol of l is defined, and (ii) Var(l) ⊇ Var(r). A term rewrite system (TRS) over F is a finite set of rewrite rules. 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 R is denoted as NF(R), or NF for short. We call a substitution σ normalised with respect to R if all terms in the range of σ are ground normal forms of R. Typically 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.

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. Note that an orthogonal TRS is confluent. A TRS is constructor if all arguments of left-hand sides are basic.

The innermost rewrite relation →_R of a TRS R is defined on terms as follows: s →_R t if there exists a rewrite rule l → r ∈ R, a context C, and a substitution σ such that s = C[lσ], t = C[rσ], and all proper subterms of lσ are normal forms of R. In order to generalise the innermost rewriting relation to relative rewriting, we introduce the slightly technical construction of the restricted rewrite relation [16].

The restricted rewrite relation Q →_R is the restriction of →_R where all arguments of the redex are in normal form with respect to the TRS Q. We define the innermost rewrite relation, dubbed →_{R/S}, of a relative TRS R/S as follows.

→_{R/S} := R∪S →_R^∗ · R∪S →_R · R∪S →_R^∗.

Observe that →_{R/S} = →_{R/∅} holds.

Let s and t be terms, such that t is in normal-form. Then a derivation D: s →_{R/S} t with respect to a TRS R is a finite sequence of rewrite steps. The derivation height of a term s with respect to a well-founded, finitely branching relation → is defined as dh(s, →) = max{n | ∃t s →^n t}.

Definition 1. We define the innermost runtime complexity (with respect to R/S): \( \text{rc}_{R/S}(n) := \max\{\text{dh}(t, \rightarrow_{R/S}) \mid t \text{ is basic and } |t| \leq n\}. \)

Intuitively the innermost runtime complexity wrt. R/S counts the maximal number of eager evaluation steps in R in a derivation over R ∪ S. In the definition, we tacitly assume that →_{R/S} is terminating and finitely branching.

For the rest of the paper the relative TRS R/S and its signature F are fixed. In the sequel of the paper, substitutions are assumed to be normalised with respect to R ∪ S.

4
3 Resource Annotations

In this section, we establish a novel amortised resource analysis for TRSs. This analysis is based on the potential method and isochronous in an inference system. Firstly, we annotate the (untyped) signature by the prospective resource usage (Definition 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 10) for which we deduce polynomial bounds on the innermost runtime complexity for a suitably chosen class of annotations, cf. Theorem 16.

A resource annotation \( p \) is a vector \( p = (p_1, \ldots, p_k) \) over non-negative rational numbers. The vector \( p \) is also simply called annotation. Resource annotations are denoted by \( p, q, u, v, \ldots \), possibly extended by subscripts and we write \( A \) for the set of such annotations. For resource annotations \( (p) \) of length 1 we write \( p \).

We will see that 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 \( \varepsilon \) if needed. So, for example \( (1) \) and \( (1, 0) \).

**Definition 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 \( [p_1 \times \cdots \times p_n] \overset{k}{\rightarrow} q \). The set of annotations is denoted \( F_{\text{pol}} \).

We lift signatures \( F \) to annotated signatures \( \tilde{F} : C \cup D \rightarrow (P(F_{\text{pol}}) \setminus \emptyset) \) 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 of constructor-like symbol \( f \) results in \( q \), there must only be exactly one declaration of the form \( [p_1 \times \cdots \times p_n] \overset{k}{\rightarrow} q \) in \( F(f) \), 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 \( [p_1 \times \cdots \times p_n] \overset{k_s}{\rightarrow} q \) and \( [p'_1 \times \cdots \times p'_n] \overset{k'}{\rightarrow} q' \) then it also has the annotations \( [\lambda p_1 \times \cdots \times \lambda p_n] \overset{k}{\rightarrow} \lambda q \) \((\lambda \in \mathbb{Q}^+, \lambda \geq 0)\) and \( [p_1 + p'_1 \times \cdots \times p_n + p'_n] \overset{k+k'}{\rightarrow} q + q' \).

**Example 3.** Consider the sets \( D = \{\text{enq}, \text{rev}, \text{rev'}, snoc, \text{chk}, \text{hd}, \text{tl}\} \) and \( C = \{\text{nil}, \sharp, \text{que}, 0, \$\} \), which together make up the signature \( F \) of the motivating example \( R_1 \) in Figure 1. Annotations of the constructors \( \text{nil} \) and \( \sharp \) would for example be as follows. \( F(\text{nil}) = \{[[\overset{0}{\rightarrow} k] \mid k \geq 0]\} \) and \( F(\sharp) = \{[[0 \times k] \overset{k}{\rightarrow} k] \mid k \geq 0\} \). 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 \((0, \ldots, 0, 1)\) (remember the implicit filling up with 0s). An annotated signature \(F\) is simply called signature, where we sometimes write \(f: [p_1 \times \cdots \times p_n] \xrightarrow{k} q \) instead of \([p_1 \times \cdots \times p_n] \xrightarrow{k} q \in F(f)\).

The next definition introduces the notion of the potential of a normal form. For rules \(f(l_1, \ldots, l_n) \rightarrow r\) in non-constructor TRSs the left-hand side \(f(l_1, \ldots, l_n)\) need not necessarily be basic terms. However, the arguments \(l_i\) are deconstructed in the rule \((app)\) that we will see in Figure 2. 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 7).

**Definition 4.** Let \(v = f(v_1, \ldots, v_n)\) be a normal form and let \(q\) be a resource annotation. We define the potential of \(v\) with respect to \(q\), written \(\Phi(v: q)\) by cases. First suppose \(v\) contains only constructors or constructor-like symbols. Then the potential is defined recursively.

\[
\Phi(v: q) := k + \Phi(v_1: p_1) + \cdots + \Phi(v_n: p_n),
\]

where \([p_1 \times \cdots \times p_n] \xrightarrow{k} q \in F(f)\). Otherwise, we set \(\Phi(v: q) := 0\).

The sharing relation \(\gamma(p|p_1, p_2)\) holds if \(p_1 + p_2 = p\).

**Lemma 5.** Let \(v\) be a a normal form. If \(\gamma(p|p_1, p_2)\) then \(\Phi(v: p) = \Phi(v: p_1) + \Phi(v: p_2)\). Furthermore, if \(p \leq q\) then \(\Phi(v: p) \leq \Phi(v: q)\).

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

**Definition 6.** Our potential based amortised analysis is coached in an inference system whose rules are given in Figure 2. Let \(t\) be a term and \(q\) a resource annotation. The inference system derives judgements of the form \(\Gamma \vdash t: q\), where \(\Gamma\) 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 \(\Gamma \vdash_{fp} t: q\). 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 \(\Gamma \vdash t: q\) and \(\Gamma \vdash_{fp} t: q\).

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 \(\Gamma\) if for all \(x \in \text{dom}(\sigma)\) if
Example 8. Consider the rule

\[
\begin{align*}
\forall x_i & \text{ are fresh} \\
\left[ x_1 : p_1, \ldots, x_n : p_n \right] f(x_1, \ldots, x_n) & \rightarrow q \\
\Gamma, x_1 : p_1, \ldots, x_n : p_n & \vdash f(x_1, \ldots, x_n) : q \\
\end{align*}
\]

Lemma 9. Let \( f(l_1, \ldots, l_n) \rightarrow r, n \geq 1 \) be a rule in the TRS \( R/S \). Further suppose \( f : [p_1 \times \cdots \times p_n] \rightarrow q \) 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 \( y_1 : r_1, \ldots, y_m : r_m \) and an amortised cost \( \ell \), defined as follows:

- The sequence \( l'_1, \ldots, l'_n \) is a linearisation of \( l_1, \ldots, l_n \). Set \( Z := \bigcup_{i=1}^n \text{Var}(l'_i) \) and let \( Z = \{ z_1, \ldots, z_{m'} \} \), where \( m' \geq m \).
- There exist annotations \( s_1, \ldots, s_{m'} \) such that for all \( i \) there exist costs \( \ell_i \) such that \( z_1 : s_1, \ldots, z_{m'} : s_{m'} \vdash l'_i : p_i \).
- Let \( y_j \in V \) and let \( \{ z_{j_1}, \ldots, z_{j_n} \} \subseteq Z \) be all renamings of \( y_j \). Define annotations \( r_j := s_{j_1} + \cdots + s_{j_n} \).
- Finally, \( \ell := \sum_{i=1}^n \ell_i \).

Example 8. Consider the rule \( \text{enq}(s(n)) \rightarrow \text{snoc}(\text{enq}(n), n) \) in the running example, together with the annotated signature \( \text{enq} : [15] \rightarrow 7 \). The left-hand side contains the subterm \( s(n) \). Using the generic annotation \( s|k \rightarrow k \), the footprint \( n : k \rightarrow p \vdash s(n) : k \) is derivable for any \( k \geq 0 \). Thus, in particular the rule frees the context \( n : 15 \) and cost 15.

Lemma 9. Let \( f(l_1, \ldots, l_n) \rightarrow r, n \geq 1 \) be a rule in the TRS \( R/S \) and let \( c : [p_1 \times \cdots \times p_n] \rightarrow q \) denote a fresh, cost-free constructor. Let \( y_1 : r_1, \ldots, y_m : r_m \) and \( \ell \) be free by the rule. We obtain:

\[
\begin{align*}
\forall y_i & \text{ are fresh} \\
y_1 : r_1, \ldots, y_m : r_m & \vdash c(l_1, \ldots, l_n) \rightarrow q \\
\end{align*}
\]
in [8]. 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 10.** Let \( R/S \) be a relative TRS, let \( F \) be a signature and let \( f \in F \). An annotation \([p_1 \times \cdots \times p_n] \xrightarrow{k} q \in F(f)\) is called resource bounded if for any rule \( f(l_1, \ldots, l_n) \rightarrow r \in R \cup S\), we have

\[
y_1 : r_1, \ldots, y_l : r_l \xrightarrow{k + \ell - K_{\text{rule}}} r : q,
\]

where \( y_1 : r_1, \ldots, y_l : r_l \) and \( \ell \) are freed by the rule if \( n \geq 1 \) and \( \ell = 0 \) otherwise. Here, the cost \( K_{\text{rule}} \) for the application of the rule is defined as follows: (i) \( K_{\text{rule}} := 1 \) iff \( f(l_1, \ldots, l_n) \rightarrow r \in R \) and (ii) \( K_{\text{rule}} := 0 \) iff \( f(l_1, \ldots, l_n) \rightarrow r \in S \).

We call an annotation cost-free resource bounded if the cost \( K_{\text{rule}} \) is always set to zero.

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

In a nutshell, the method works as follows: Suppose the judgement \( \Gamma \mid t : q \) is derivable and suppose \( \sigma \) is consistent with \( \Gamma \). 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 \( O(n^k) \), where \( n \) is the size of the arguments and \( k \) the maximal length of the resource annotations needed, then the innermost runtime complexity of \( R/S \) lies in \( O(n^k) \).

More precisely consider the \texttt{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 \( k'_i \) \((i \geq 1)\) 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 11** (continued from Example 3). \texttt{TCT} derives the following annotations for the operators in the running example.

\[
\begin{array}{llll}
\text{enq: } & [15] & \xrightarrow{12} 7 & \text{rev: } [1] \xrightarrow{4} 0 \\
\text{snoc: } & [7 \times 0] & \xrightarrow{14} 7 & \text{hd: } [11] \xrightarrow{9} 0 \\
\text{rev': } & [1 \times 0] & \xrightarrow{2} 0 & \text{tl: } [11] \xrightarrow{1} 1 \\
\end{array}
\]

\( \square \)
We consider resource boundedness of $R_1$ with respect to the given (monomorphic) annotated signatures of Example 11. For simplicity we restrict to boundedness of $\text{enq}$. We leave it to the reader to check the other cases. In addition to the annotations for constructor symbols (cf. Example 3) we can always assume the presence of zero-cost annotations, e.g. $\sharp: [0 \times 0] \rightarrow 0$. Observe that Rule 6 frees the context $n: 15$ and cost 15. Thus, we obtain the following derivation.

\[
\begin{align*}
\text{snoc}: [7 \times 0] & \quad \xrightarrow{14} \quad 7 & \text{(app)} \\
q: 7, m: 0 \quad & \quad \xrightarrow{14} \quad \text{snoc}(q, m): 7 \\
n_1: 15, n_2: 0 \quad & \quad \xrightarrow{26} \quad \text{snoc}(\text{enq}(n_1), n_2): 7 & \text{(comp)} \\
n: 15 \quad & \quad \xrightarrow{26} \quad \text{snoc}(\text{enq}(n), n): 7 & \text{(share)}
\end{align*}
\]

In comparison to [8, Example 13], where the annotations were found manually, we note that the use of the interleaving operation [8] 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 12.** Let $t$ be a normal form w.r.t. $R$, where $t$ consists of constructor or constructor-like symbols only. If $\Gamma \vdash^k t: q$, then $\Phi(t\sigma: q) = \Phi(\sigma: \Gamma) + k$.

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

**Lemma 13.** Let $\Gamma$ be a context and let $\sigma$ be a substitution consistent with $\Gamma$. Then $\Gamma \vdash t: q$ implies $\vdash t\sigma: q$.

We establish soundness with respect to relative innermost rewriting.

**Theorem 14.** Let $R/S$ be a resource bounded TRS and let $\sigma$ be a normalised substitution such that $\sigma$ is consistent with the context $\Gamma$. Suppose $\Gamma \vdash^k t: p$ and $t\sigma \rightarrow^k_{R/S} \tau \cdot v$. Then there exists a normalising substitution $\tau$ such that $\Delta \vdash^k t: q$ is derivable and $\Phi(\sigma: \Gamma) + k - \Phi(\tau: \Delta) \geq K$.

**Proof.** Let $\Pi$ denote the derivation of the judgement $\Gamma \vdash^k t: q$. The proof proceeds by case distinction on derivation $D: t\sigma \rightarrow^k_{R/S} \tau \cdot v$ and side-induction on $\Pi$. The proof proceeds by case distinction on $D$ and induction on the length of $\Pi$.

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

**Corollary 15.** Let $R/S$ be a bounded TRS and let $\sigma$ be a normalising substitution consistent with the context $\Gamma$. Suppose $\Gamma \vdash^k t: q$ and $D: \tau \cdot v \rightarrow^1_{R/S} v \in \text{NF}$. Then (i) $\vdash v: q$ and (ii) $\Phi(\sigma: \Gamma) - \Phi(v: q) + k \geq |D|$ hold.
The next theorem defines suitable constraints on the resource annotations to
deduce polynomial innermost runtime from Theorem 14. Its proof follows the
pattern of the proof of Theorem 14 in [8].

**Theorem 16.** Suppose that for each constructor $c$ with $[p_1 \times \cdots \times p_n] \xrightarrow{k'} q \in F(c)$, there exists $r_i \in A$ such that $p_i \leq q + r_i$, where $\max r_i \leq \max q =: r$ and $p \leq r$ with $|r_i| < |q| =: k$. Then $\Phi(v; q) \leq r|v|^k$, and thus the innermost runtime complexity of the TRS under investigation is in $\mathcal{O}(n^k)$.

**Proof.** The theorem follows the pattern of the proof of Theorem 14 in [8]. □

We note that our running example satisfies the premise of Theorem 16. Thus the linear bound on the innermost runtime complexity of the running example $R_1$ follows. The next example clarifies that without further assumptions potentials are not restricted to polynomials.

**Example 17.** Consider that we annotate the constructors for natural numbers as $0: [] \xrightarrow{0} p$ and $s: [2p] \xrightarrow{p} p$, where $p = (p_1, \ldots, p_k)$. We then have, for example, $\Phi(t; 1) = 2^{v} - 1$, 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 given TRS. While on the other hand the integration into $\mathsf{TCT}$ [11] uses relative rewriting. Clearly, as an integration into $\mathsf{TCT}$ was planned from the beginning, the language used for the implementation of the amortised resource analysis module is Haskell. The modular design of $\mathsf{TCT}$ 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 16).

**Inference Tree Derivation and Resource Boundedness.** To be able to apply the inference rules the expected root judgement of each rule is generated (as in Example 11) by the program and the inference rules of Figure 2 are applied. To gain determinism the inference rules are ordered in the following way. The share-rule has highest priority, followed by $\mathsf{app}$, $\mathsf{var}$, $\mathsf{comp}$ 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

\footnote{See http://haskell.org/}
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 \( T\subseteq \) 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.

**Superposition of Constructors.** Recall that constructor and constructor-like symbols \( f \) must satisfy the superposition principle. Therefore, for each annotation \([p_1 \times \cdots \times p_n] \xrightarrow{\lambda k} q\) of \( f \) it must be ensured that there is no annotation \([\lambda \cdot p_1 \times \cdots \times \lambda \cdot p_n] \xrightarrow{\lambda k} q'\) 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 \( q' \geq q \) 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 \([\lambda \cdot p_1 \times \cdots \times \lambda \cdot p_n] \xrightarrow{\lambda k} \lambda \cdot q\).

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 \[17\]. Early experiments revealed their bad performance. Overcoming this issue using Farkas’ Lemma \[18\] is not possible here. Thus, we developed the 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\[4\]. Thus, in contrast to the techniques presented in \[3,19,20\], 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 18.** Consider the base constructors \( \sharp_1 : [(0, 0) \times (1, 0)] \xrightarrow{1} (1, 0) \) and \( \sharp_2 : [(0, 0) \times (2, 1)] \xrightarrow{1} (0, 1) \) 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. \( \square \)

**Cost-Free Function Symbols.** Inspired by Hoffmann \[19\] p.93ff we additionally implemented a cost-free inference tree derivation when searching for non-linear

---

\[4\] We use the SMT Solvers z3 [https://github.com/Z3Prover/z3/wiki] and MiniSmt [http://cl-informatik.uibk.ac.at/software/minismt/].
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 \texttt{app}-rule, which separates the function signature into two parts, cf. Figure 3. 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 $f(l_1, \ldots, l_n) \rightarrow r$ and freed context $y_1: \mathbf{r}_1, \ldots, y_l: \mathbf{r}_l$ and cost $\ell$. Thus, the new set of annotations for a defined function symbols $f$ is given by the following set, cf. [19, p. 93].

$$\{[p_1 + \lambda \cdot p_1^{cf} \times \cdots \times p_n + \lambda \cdot p_n^{cf}] \xrightarrow{k + \lambda \cdot k^{cf}} q + \lambda \cdot q^{cf} | \lambda \in \mathbb{Q}^+, \lambda \geq 0\}.$$ 

The decision of which \texttt{app} rule is applied utilises the strongly connected component (SCC) of the call graph analysis as done in [19, p.93ff].

Alternative Implementation of the Superposition Principle. Similar to [8,19] we integrated the additive shift $\ll(p)$ and interleaving $p \parallel q$ for constructors when type information is given. Here $\ll(p_1, \ldots, p_k) := \langle p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k \rangle$ and $p \parallel q := (p_1, q_1, p_2, q_2, \ldots, p_k, q_k)$, 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 whenever these heuristics are used which tremendously reduces the execution times.

However, according to the experiments (see the detail results online) 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 Experimental Evaluation

In this section we will have a look at how the amortised analysis deals with some selected examples including the paper’s running example queue. All experiments 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 as well as a collection consisting of 140 TRSs representing first-order functional programs, transformations from higher-order programs, or RaML programs 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 Figure 4 shows the results of the experiments conducted for the TCT with ARA. In a companion paper, we have studied best case complexity and suitable adapted amortised resource analysis to obtain lower bounds on the best case complexity. Therefore, the standalone tool is also able to infer best case complexity bounds for TRSs.

---

| Example | O(1) | O(n^1) | O(n^2) | O(n^3) | O(n^2^) | Fail |
|---------|------|--------|--------|--------|---------|------|
| #Systems | 2    | 59     | 17     | 21     | 8       | 33   |
| Time (in s) | 0.05 | 0.54   | 2.86   | 5.06   | 10.14   | 58.30 |

Fig. 4: Experimental evaluation of TCT with ARA.

---

#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 the Appendix for the TRS. The optimal runtime complexity of \( R_1 \) is linear in \( n \). For this, first observe that the evaluation of \( \text{half}(s^m(0)) \) and \( \text{lastbit}(s^m(0)) \) 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 \( k := \lfloor (n)_2 \rfloor \). 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 \( O(n^2) \) we find this optimal linear bound of \( O(n) \) when ARA is enabled. Furthermore, the best case analysis of ARA shows that this bound is tight by returning \( \Omega(n) \). Similarly AProVE yields the tight bound employing a size abstraction to integer transition systems (ITSs for short), cf. [24]. The resulting ITSs are then solved with CoFloCo, which also embodies an amortisation analysis.

---

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

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

7 See [https://aprove-developers.github.io/trs_complexity_via_its/](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.
**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 Appendix, and can be found in the TPDB. Note that the TRS uses strict rules for the equality check which 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. TT 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 TT 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 TT is triggered to find the best bound within the timeout, it will infer $O(n^3)$ as AProVE does. This bound is tight [19] 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 $O(n^2)$ [19] 158ff. Although far from being optimal, TT 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 TT with ARA is in comparison to an older version able to solve the problem. The inferred quadratic worst case bound 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 the Appendix 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 TT 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, TT returns the asymptotically optimal upper bound defined by the list matrix multiplication of $O(n^4)$. Neither the older version of TT 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 TT was adapted to focus on TRSs which were translated from functional programs. Thus, the examples which can be solved are distinct from the TT competition strategy of 2016 to a great extent. Due to ARA the latest competition strategy
of $\mathcal{TCT}$ 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 $\mathcal{TCT}$ with ARA can find better bounds for 22 examples in contrast to $\mathcal{TCT}$ without ARA and additionally $\text{bfs.raml}$ can be solved. For further experiments see the detailed results.

6 Conclusion

In this paper we have established a novel automated amortised cost analysis for term rewriting. In doing so we have not only implemented the methods detailed in earlier work [8], but also generalised the theoretical basis considerably. We have provided a prototype implementation and integrated into $\mathcal{TCT}$.

More precisely, we have extended the method of 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 $\mathcal{TCT}$. The provided integration of amortised resource analysis into $\mathcal{TCT}$ 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.

In future work 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. [9]. However efficient automation of the method proposed in [9] 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 [1,2], compare also [27].

References

1. Sleator, D., Tarjan, R.: Self-adjusting binary trees. In: Proc. of the 15th STOC, ACM (1983) 235–245
2. Tarjan, R.: Amortized computational complexity. SIAM J. Alg. Disc. Meth 6(2) (1985) 306–318
3. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for ocaml. In: Proc. 44th POPL, ACM (2017) 359–373
4. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. JAR 46(2) (2011)
5. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Proc. 21st FM. Volume 9995 of LNCS. (2016)
6. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Proc. Software Engineering. Volume 252 of LNI. (2016) 101–102
7. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
8. Hofmann, M., Moser, G.: Amortised resource analysis and typed polynomial interpretations. In: Proc. of Joint 25th RTA and 12th TLCA. Volume 8560 of LNCS. (2014) 272–286
9. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: Proc. 13th TLCA. Volume 38 of LIPIcs. (2015) 241–256
10. Avanzini, M., Lago, U.D.: Automating sized-type inference for complexity analysis. PACMPL 1(ICFP) (2017) 43:1–43:29
11. Avanzini, M., Moser, G., Schaper, M.: TcT: Tyrolean Complexity Tool. In: Proc. 22nd TACAS. LNCS (2016) 407–423
12. Moser, G., Schneckenreither, M.: Amortised analysis for bestcase lowerbounds. submitted (2018)
13. Schneckenreither, M.: Amortized resource analysis for term rewrite systems. Master’s thesis, University of Innsbruck (2018) [https://www.uibk.ac.at/wipl/team/team/docs/masterthesis_schneckenreither.pdf]
14. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
15. TeReSe: Term Rewriting Systems. Volume 55 of Cambridge Tracks in Theoretical Computer Science. Cambridge University Press (2003)
16. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, University of Aachen, Department of Computer Science (2007)
17. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017) Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
18. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik 124 (1902) 1–27
19. Hoffmann, J.: Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis. PhD thesis, Ludwig-Maximilians-Universität München (2011)
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. TOPLAS 34(3) (2012) 14
21. Glenstrup, A.: Terminator ii: Stopping partial evaluation of fully recursive programs. Master’s thesis, DIKU (1999) Technical Report DIKU-TR-99/8.
22. Frederiksen, C.: Automatic runtime analysis for first order functional programs. Master’s thesis, DIKU (2002) DIKU TOPPS D-470.
23. Avanzini, M., Lago, U.D., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: Proc. 20th ICFP, ACM (2015) 152–164
24. Naaf, M., Frohn, F., Brockschmidt, M., Fuhs, C., Giesl, J.: Complexity analysis for term rewriting by integer transition systems. In: International Symposium on Frontiers of Combining Systems, Springer (2017) 132–150
25. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plicker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with apropre. JAR 58(1) (2017) 3–31
26. Flores-Montoya, A.: Cofloco: System description. In: 15th International Workshop on Termination. (2016) 20
27. Nipkow, T.: Amortized complexity verified. In: Proc. 6th ITP. Volume 9236 of LNCS. (2015) 310–324