

# Multivariate Amortised Resource Analysis for Term Rewrite Systems\*

Martin Hofmann<sup>1</sup> and Georg Moser<sup>2</sup>

1 Institute of Computer Science  
LMU Munich, Germany  
hofmann@ifi.lmu.de

2 Institute of Computer Science  
University of Innsbruck, Austria  
georg.moser@uibk.ac.at

---

## Abstract

We study amortised resource analysis in the context of term rewrite systems. We introduce a novel amortised analysis based on the potential method. The method is represented in an inference system akin to a type system and gives rise to polynomial bounds on the innermost runtime complexity of the analysed rewrite system. The crucial feature of the inference system is the admittance of multivariate bounds in the context of arbitrary data structures in a completely uniform way. This extends our earlier univariate resource analysis of typed term rewrite systems and continues our program of applying automated amortised resource analysis to rewriting.

**1998 ACM Subject Classification** F.3.2 Program Analysis

**Keywords and phrases** program analysis, amortised analysis, term rewriting, multivariate bounds

**Digital Object Identifier** 10.4230/LIPIcs.TLCA.2015.241

## 1 Introduction

Amortised resource analysis was pioneered by Sleator and Tarjan who used it to analyse the performance of new data structures that sometimes need to perform costly operations that pay off later on, e.g. rebalancing operations on a search tree.

Briefly, one assigns to the participating datastructures a nonnegative real-value, the *potential* in an a priori arbitrary fashion. One then defines the amortised cost of an operation as its actual cost, e.g. runtime plus the difference in potential of all datastructures before and after the operations. In this way, the amortised cost of a costly operation may be small if it results in a big decrease of potential. On the other hand some cheap operations that increase the potential will be overcharged. In this way, one can “save money” now to pay for costly operations later. By a simple telescoping argument the sum of all amortised costs in a sequence of operations plus the potential of the initial input data structure is also an upper bound on the *actual* cost of that sequence. In this way, amortised analysis yields rigorous bounds on actual resource usage and not just approximate or average bounds. If the potential functions are chosen well then the amortised costs of operations are either constant or exhibit merely a very simple dependency on the maximum size of all intermediate results which considerably facilitates a compositional analysis: the costs of running composite expressions

---

\* This research is partially supported by FWF (Austrian Science Fund) project P25781.



can be calculated as the sum of the individual costs. If costs are highly input-dependent, on the other hand, one must get bounds on the sizes or shapes of intermediate results which can be very difficult. This compositional aspect of amortised analysis makes it attractive for syntax-directed automation.

Of course, the crux of the matter is the choice of the correct potential functions. A simple concrete example is the implementation of a queue by two stacks, an in-tray and an out-tray. Incoming elements are added to the in-tray, outgoing elements are taken from the top of the out-tray. Only if the out-tray becomes empty the entire in-tray is reverse-copied into the out-tray. In this case, the length of the in-tray is clearly a suitable potential function. The costly operation of copying can entirely be paid from the big decrease in potential it causes.

In a nutshell automated amortised analysis works as follows. One selects a collection of basic potential functions, called basic resource functions, and assumes that all potential functions are linear combinations of these basic resource functions. One then performs a symbolic amortised cost analysis where the coefficients of these linear combinations as well as the amortised costs of operations (assumed constant) are unknowns. This yields a system of linear constraints for these unknowns whose solution provides the desired amortised analysis from which actual cost bounds in the form of functions of the size of the initial input can be easily read off.

Most automated amortised analyses are univariate in the sense that the joint potential of several arguments to an operation (a “context”) is calculated as the sum of the individual potentials. This, however, proved unsatisfactory in the analysis of nested data structures such as lists of lists or trees and led to the development of multivariate analysis where also products and sums of products of the individual potentials may be used [9].

While automated amortised analysis has hitherto mostly been applied to functional and to a lesser extent to imperative programs; we are here interested in its application to term rewriting understood as a generalisation of functional programming to arbitrary constructor-defined datatypes. After a first step in this direction [12] which was based on univariate analysis, we now generalise to multivariate analysis and indeed subsume and further extend the entire system from [9].

On the one hand this gives a more general treatment of algebraic datatypes which are now untyped and merely defined by their constructor symbols. On the other hand, this necessitates a more general approach to basic resource functions which also streamlines the existing format in [9] or for that matter [11]. In [9] a basic resource function for a list type  $L(A)$  is given by a finite list  $[p_1, \dots, p_k]$  of basic resource functions for the underlying type of entries  $A$ . The interpretation of such a list as a nonnegative  $\mathbb{R}$ -valued potential function was then given by the formula

$$[p_1, \dots, p_k]([v_1, \dots, v_n]) := \sum_{1 \leq i_1 < \dots < i_k \leq n} p_1(v_{i_1}) \cdots p_k(v_{i_k}).$$

By treating tree types as a list of entries in depth-first order this same format could then be applied to trees as well. While these formats provided a smooth interaction with the typing rules and allow a very precise analysis of many examples they still look somewhat arbitrary and unjustified.

In the present paper, these formats are subsumed under a very general pattern that is on the one hand simpler and on the other hand permits an even smoother interaction with the typing rules for constructors and matching.

Namely, for us, a basic resource function is defined simply by a bottom-up tree automaton

$\mathcal{A}$  which acts on values by

$$p_{\mathcal{A}}(v) := \text{number of accepting runs of } \mathcal{A} \text{ on } v .$$

If  $c$  is a binary constructor, we have

$$p_{\mathcal{A}}(c(v_1, v_2)) = \sum_{c(\beta_1, \beta_2) \rightarrow \alpha \in \mathcal{A}} p_{(\mathcal{A}, \beta_1)}(v_1) \cdot p_{(\mathcal{A}, \beta_2)}(v_2) .$$

where  $\alpha$  is the final state of  $\mathcal{A}$  and  $c(\beta_1, \beta_2) \rightarrow \alpha$  represents a transition in  $\mathcal{A}$  and  $(\mathcal{A}, \beta_i)$  denotes  $\mathcal{A}$  with the final state set to  $\beta_i$ . Using this formula, the above formats for potentials on lists and trees are readily derived and obviously much more general potential functions can be defined which for example perform a rudimentary kind of type checking by simply ignoring certain constructors or, more interestingly, insisting on certain local patterns.

We note that the expression  $p_{\mathcal{A}}(v)$  is known as the *ambiguity* of  $\mathcal{A}$  [13] and has been extensively studied. In particular, the above recursive expansion of  $p_{\mathcal{A}}(v)$  is known and attributed to Kuich, cf. [13]. However, these previous studies focused mainly on bounding  $\max_v p_{\mathcal{A}}(v)$  as a function of the number of states (in the case where this quantity is at all finite) and has to our knowledge never been applied to complexity analysis of tree-like data structures.

Let us now look at a concrete example. Consider the following TRS  $\mathcal{R}_{\text{dyade}}$ , encoding vector multiplication. The example forms a direct translation of the `dyade.raml` program discussed in [9].

$$\begin{array}{ll} 0 + y \rightarrow y & \mathfrak{s}(x) + y \rightarrow \mathfrak{s}(x + y) \\ 0 \times y \rightarrow 0 & \mathfrak{s}(x) \times y \rightarrow y + (x \times y) \\ \text{mult}(n, []) \rightarrow [] & \text{mult}(n, x :: xs) \rightarrow (n \times x) :: \text{mult}(n, xs) \\ \text{dyade}([], ls) \rightarrow [] & \text{dyade}(x :: xs, ls) \rightarrow \text{mult}(x, ls) :: \text{dyade}(xs, ls) . \end{array}$$

Consider a call to `dyade`( $ls_1, ls_2$ ), where  $ls_1$  and  $ls_2$  are lists. It is easy to see that the runtime complexity of  $\mathcal{R}_{\text{dyade}}$  crucially depends on the sum of the entries of  $ls_1$  times the sum of the entries of  $ls_2$ , that is an optimal (automated) analysis should provide us with the certificate  $O(|ls_1| \cdot |ls_2|)$ . However, state-of-the-art complexity tools, like AProVE [7], or  $\mathsf{TCT}$  [2] overestimate the actual resource usage. For example  $\mathsf{TCT}$  will provide a polynomial interpretation of degree 2, which is quadratic in  $|ls_1|$ , even if the monotonicity conditions are weakened suitably, cf. [8]. Also our earlier amortised resource analysis of typed TRS [12] can only provide the non-optimal bound  $O(|ls_1|^2 + |ls_2|^2)$ . On the other hand the automated analysis of the RaML prototype is more to the point; the analysis with respect to `dyade.raml`, just overestimates the optimal bound by a linear factor and provides unnecessary big constants. The multivariate amortised analysis provided in this paper allows to lift this analysis to the above example and provides essentially optimal bounds (see the example on page 254).

This paper is structured as follows. In the next section we cover basics. In Section 3 we introduce resource functions as generalisations of resource polynomials to arbitrary constructor-defined datatypes. In Section 4 we present our type system and establish its soundness. Finally, we conclude in Section 5, where we also present related work.

## 2 Term Rewrite Systems and Tree Automata

We assume familiarity with term rewriting [4, 16] and tree automata [6] but briefly review basic concepts and notations.

$$\begin{array}{c}
\frac{x\sigma = v}{\sigma \stackrel{0}{|} x \Rightarrow v} \qquad \qquad \qquad \frac{c \in \mathcal{C} \quad x_1\sigma = v_1 \quad \cdots \quad x_n\sigma = v_n}{\sigma \stackrel{0}{|} c(x_1, \dots, x_n) \Rightarrow c(v_1, \dots, v_n)} \\
\\
\frac{f(l_1, \dots, l_n) \rightarrow r \in \mathcal{R} \quad \exists \tau \forall i: x_i\sigma = l_i\tau \quad \sigma \uplus \tau \stackrel{m}{|} r \Rightarrow v}{\sigma \stackrel{m+1}{|} f(x_1, \dots, x_n) \Rightarrow v} \\
\\
\text{all } x_i \text{ are fresh} \\
\frac{\sigma \uplus \rho \stackrel{m_0}{|} f(x_1, \dots, x_n) \Rightarrow v \quad \sigma \stackrel{m_1}{|} t_1 \Rightarrow v_1 \quad \cdots \quad \sigma \stackrel{m_n}{|} t_n \Rightarrow v_n \quad m = \sum_{i=0}^n m_i}{\sigma \stackrel{m}{|} f(t_1, \dots, t_n) \Rightarrow v} \\
\\
\text{Here } \rho := \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}.
\end{array}$$

■ **Figure 1** Operational Big-Step Semantics.

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  $\mathcal{V}\text{ar}(t)$  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. The set of ground constructor terms is denoted as  $\mathcal{T}(\mathcal{C})$ , ground constructor terms are also called *values*. A *substitution*  $\sigma$  is a mapping from variables to terms. Substitutions are conceived as sets of assignments:  $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ . We write  $\text{dom}(\sigma)$  ( $\text{rg}(\sigma)$ ) to denote the domain (range) of  $\sigma$ . Let  $\sigma, \tau$  be substitutions such that  $\text{dom}(\sigma) \cap \text{dom}(\tau) = \emptyset$ . Then we denote the (disjoint) union of  $\sigma$  and  $\tau$  as  $\sigma \uplus \tau$ . We call a substitution  $\sigma$  *normalised* if all terms in the range of  $\sigma$  are values.

A *rewrite rule* is a pair  $l \rightarrow r$  of terms, such that (i) the root symbol of  $l$  is defined, and (ii)  $\mathcal{V}\text{ar}(l) \supseteq \mathcal{V}\text{ar}(r)$ . A rule  $l \rightarrow r$  is called *left-linear*, if  $l$  is linear. A *term rewrite system* (TRS for short) over  $\mathcal{F}$  is a finite set of rewrite rules. In the sequel,  $\mathcal{R}$  always denotes a TRS. The rewrite relation is denoted as  $\rightarrow_{\mathcal{R}}$  and we use the standard notations for its transitive and reflexive closure. We simply write  $\rightarrow$  for  $\rightarrow_{\mathcal{R}}$  if  $\mathcal{R}$  is clear from context. Let  $s$  and  $t$  be terms. If exactly  $n$  steps are performed to rewrite  $s$  to  $t$ , we write  $s \rightarrow^n t$ . In the sequel we are concerned with *innermost* rewriting, that is, an eager evaluation strategy. The *innermost rewrite relation*  $\overset{\cdot}{\rightarrow}_{\mathcal{R}}$  of a TRS  $\mathcal{R}$  is defined on terms as follows:  $s \overset{\cdot}{\rightarrow}_{\mathcal{R}} t$  if there exists a rewrite rule  $l \rightarrow r \in \mathcal{R}$ , a context  $C$ , and a substitution  $\sigma$  such that  $s = C[l\sigma]$ ,  $t = C[r\sigma]$ , and all proper subterms of  $l\sigma$  are normal forms of  $\mathcal{R}$ .

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. Let  $s$  and  $t$  be terms, such that  $t$  is in normal-form. Then a *derivation*  $D: s \rightarrow_{\mathcal{R}}^* t$  with respect to a TRS  $\mathcal{R}$  is a finite sequence of rewrite steps. The *derivation height* of a term  $s$  with respect to a well-founded, finitely branching relation  $\rightarrow$  is defined as:  $\text{dh}(s, \rightarrow) = \max\{n \mid \exists t \ s \rightarrow^n t\}$ . A term  $t = f(t_1, \dots, t_k)$  is called *basic* if  $f$  is defined, and all  $t_i \in \mathcal{T}(\mathcal{C})$ . We define the (*innermost*) *runtime complexity* (with respect to  $\mathcal{R}$ ):  $\text{rc}_{\mathcal{R}}(n) := \max\{\text{dh}(t, \overset{\cdot}{\rightarrow}_{\mathcal{R}}) \mid t \text{ is basic and } |t| \leq n\}$ .

We study *constructor* TRSs  $\mathcal{R}$ , that is, for each rule  $f(l_1, \dots, l_n) \rightarrow r$  we have that the arguments  $l_i$  are constructor terms. Furthermore, we restrict to *completely defined* and *orthogonal* systems. These restrictions are natural in the context of functional programming as orthogonal TRSs correspond to first-order function programs with pattern matching. Let  $\mathcal{F}$  denote the signature underlying  $\mathcal{R}$ .

As  $\mathcal{R}$  is completely defined, any derivation ends in a value. In connection with innermost rewriting this yields a *call-by-value* strategy. Furthermore, as  $\mathcal{R}$  is non-overlapping any innermost derivation is determined modulo the order in which parallel redexes are contracted. This allows us to recast innermost rewriting into an operational big-step semantics instrumented with resource counters, cf. Figure 1. The semantics resembles similar definitions given in the literature on amortised resource analysis.

► **Proposition 1.** *Let  $f$  be a defined function symbol of arity  $n$  and  $\sigma$  a normalised substitution. Then  $\sigma \Vdash^m f(x_1, \dots, x_n) \Rightarrow v$  holds iff  $\text{dh}(f(x_1\sigma, \dots, x_n\sigma), \rightarrow_{\mathcal{R}}) = m$ .*

We suit the standard definition of bottom-up tree automata to our context. A *tree automaton* is a quadruple  $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \alpha, \Delta)$ , consisting of a finite signature  $\mathcal{F}$ , a finite non-empty set of states  $\mathcal{Q}$  (disjoint from  $\mathcal{F}$ ), a unique final state  $\alpha$ , and a set of non-empty transitions  $\Delta$ . Every rule in  $\Delta$  has the following form  $f(\alpha_1, \dots, \alpha_n) \rightarrow \beta$  with  $f \in \mathcal{F}$ ,  $\alpha_1, \dots, \alpha_n, \beta \in \mathcal{Q}$ .

Note that we only consider tree automata that consist of at least one state and feature a non-empty transition relation. As we will only be concerned with tree automata, we drop the qualifier “tree” and simply speak of an automaton. Observe that an automaton  $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \alpha, \Delta)$  is conceivable as a finite ground TRS  $\Delta$  over the signature  $\mathcal{F} \cup \mathcal{Q}$ , where the shape of the rewrite rules is restricted. The induced rewrite relation on  $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$  is denoted as  $\rightarrow_{\mathcal{A}}$ . A ground term  $t$  is *accepted* by  $\mathcal{A}$  if  $t \rightarrow_{\mathcal{A}}^* \alpha$ ; we set  $L(\mathcal{A}) := \{t \mid t \rightarrow_{\mathcal{A}}^* \alpha\}$ . Two automata  $\mathcal{A}$  and  $\mathcal{B}$  are *equivalent*, if  $L(\mathcal{A}) = L(\mathcal{B})$ . We use the notation  $(\mathcal{A}, \beta)$  to refer to the automaton  $(\mathcal{F}, \mathcal{Q}, \beta, \Delta)$  where  $\beta \in \mathcal{Q}$ . Note that  $(\mathcal{A}, \alpha) = \mathcal{A}$  and we sometimes use the succinct notation instead of the expanded one.

In the sequel,  $\mathcal{A}$  will always denote an automaton. Henceforth,  $\mathcal{R}$  and  $\mathcal{F}$ , as well as the defined symbols  $\mathcal{D}$  and constructors  $\mathcal{C}$  are kept fixed. Furthermore, all considered substitutions are normalised.

### 3 Resource Functions

We define a set  $\mathcal{BF}$  of *basic functions*, that map terms to natural numbers. Basic functions are indexed by a pair consisting of an automaton  $\mathcal{A}$  and a state  $\alpha$ . Resource functions will then be defined as nonnegative rational linear combinations of basic functions.

► **Definition 2.** Let  $\mathcal{A} = (\mathcal{C}, \mathcal{Q}, \alpha, \Delta)$ . For  $v \in \mathcal{T}(\mathcal{C})$  we define the *basic function*  $p_{\mathcal{A}}$ , whose value  $p_{\mathcal{A}}(v)$  is the number of accepting runs of  $\mathcal{A}$  on  $v$ . The set of basic functions is denoted as  $\mathcal{BF}$ .

For any set of constructors  $\mathcal{C}$ , there exists an automaton  $\mathcal{A}$  with  $p_{\mathcal{A}}(v) = 1$  for all values  $v$ . Moreover  $\mathcal{A}$  is unique upto renaming of the single state  $\alpha$ . We call  $\mathcal{A}$  the *canonical* automaton, denoted by  $\emptyset$ , whose unique state is denoted by  $\emptyset$ . As mentioned in the introduction  $p_{\mathcal{A}}(v)$  is called *ambiguity* in the literature (see for example [13, 14]). In particular it is known that the finiteness of the *degree of ambiguity*  $\sup_v p_{\mathcal{A}}(v)$  is polytime decidable [13]. The following is direct from the definition.

► **Proposition 3.** *Let  $v \in \mathcal{T}(\mathcal{C})$  be a value such that  $v = c(v_1, \dots, v_n)$  and let  $\alpha \in \mathcal{Q}$ . We then have:*

$$p_{(\mathcal{A}, \alpha)}(c(v_1, \dots, v_n)) := \sum_{c(\alpha_1, \dots, \alpha_n) \rightarrow \alpha \in \Delta} p_{(\mathcal{A}, \alpha_1)}(v_1) \cdots p_{(\mathcal{A}, \alpha_n)}(v_n).$$

Note that  $p_{(\mathcal{A}, \alpha)}(c) = \sum_{c \rightarrow \alpha \in \Delta} 1$ , as the empty product equals 1.

We could alternatively have used the latter as a recursive definition of  $p_{(\mathcal{A},\alpha)}(v)$ . The advantage of the definition based on runs is that we can immediately read off an exponential upper bound:

► **Proposition 4.** *Let  $q$  be the number of states of  $\mathcal{A}$  and  $n$  the size of  $v \in \mathcal{T}(\mathcal{C})$ . Then  $p_{\mathcal{A}}(v) \leq q^n$ .*

It is also easy to see that this bound is actually taken on so that unlike the basic functions in [9] ours are not in general polynomials. If desired, it is however easy to impose syntactic restrictions that ensure polynomial growth. For example, we can use a ranking function on states and require that for each transition  $c(\alpha_1, \dots, \alpha_n) \rightarrow \alpha$  the ranks of the  $\alpha_i$  are not bigger than that of  $\alpha$  and that they strictly decrease for all but one transition with symbol  $c$  and result state  $\alpha$ . All the concrete basic functions we use in examples satisfy this restriction and thus exhibit polynomial growth. We believe, however, that in some cases also exponential bounding functions may prove useful.

Let  $\mathcal{C} = \{0, s, [], ::\}$  and consider the following automaton  $\mathcal{A}$  with final state  $\beta_k$ .

$$\begin{array}{llllll} 0 \rightarrow \alpha_0 & s(\alpha_0) \rightarrow \alpha_1 & \cdots & s(\alpha_{\ell-1}) \rightarrow \alpha_\ell & & \\ & s(\alpha_0) \rightarrow \alpha_0 & \cdots & s(\alpha_{\ell-1}) \rightarrow \alpha_{\ell-1} & s(\alpha_\ell) \rightarrow \alpha_\ell & \\ [] \rightarrow \beta_0 & \alpha_{i_k} :: \beta_0 \rightarrow \beta_1 & \cdots & \alpha_{i_1} :: \beta_{k-1} \rightarrow \beta_k & & \\ & \alpha_0 :: \beta_0 \rightarrow \beta_0 & \cdots & \alpha_0 :: \beta_{k-1} \rightarrow \beta_{k-1} & \alpha_0 :: \beta_k \rightarrow \beta_k & . \end{array}$$

First, by a simple inductive argument we see that  $p_{(\mathcal{A},\alpha_i)}(s^n(0)) = \binom{n}{i}$  for  $n \geq 0$  and  $i = 0, \dots, \ell$ . Based on this, we conclude by induction on  $m$ :

$$p_{(\mathcal{A},\beta_k)}([\mathbf{n}_1, \dots, \mathbf{n}_m]) = \sum_{1 \leq j_1 < \dots < j_k \leq m} \binom{n_{j_1}}{i_1} \cdots \binom{n_{j_k}}{i_k},$$

where we denote the numeral  $s^n(0)$  by  $\mathbf{n}$ ,  $[\mathbf{n}_1, \dots, \mathbf{n}_m]$  abbreviates the corresponding cons-list, and  $1 \leq i_j \leq \ell$  for all  $j = 1, \dots, k$ .

Consider the set, denoted as  $\mathfrak{A}$ , of all non-equivalent (and non-empty) automata over  $\mathcal{C}$ . In the following we will frequently appeal to an enumeration of  $\mathfrak{A}$ , referring to a suitable chosen, but inessential encoding of automata. Note that in effect we will only work with a small, in particular finite subset of  $\mathfrak{A}$ .

► **Definition 5.** A *resource function*  $r: \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Q}^+$  is a non-negative rational linear combination of basic functions, that is,

$$r(t) := \sum_{\mathcal{A} \in \mathfrak{A}} q_{\mathcal{A}} \cdot p_{\mathcal{A}}(t),$$

where  $p_{\mathcal{A}} \in \mathcal{BF}$ . The set of resource functions is denoted as  $\mathcal{RF}$ .

The example above hints at the fact that the expressivity of our basic functions exceeds the expressivity of the *base polynomials* considered by Hoffmann et al. [10, 9]. The next proposition makes this fact precise.

► **Lemma 6.** *All the resource polynomials from [9] are also resource functions in the present automata-based setting.*

**Proof.** This is proved by induction on the definition of resource polynomials [9]. We do not need to recall their definition here; the inductive cases we establish reveal enough detail.

If  $p, q$  are basic resource polynomials for types  $A, B$  respectively then  $\lambda ab(p(a) \cdot q(b))$  is a base resource polynomial for the product type  $A \times B$ . In rewriting, we can simulate product types by introducing a binary constructor  $\text{pair}(x, y)$ . Now, if, inductively  $p = p_{(\mathcal{A}, \alpha)}$  and  $q = p_{(\mathcal{B}, \beta)}$ , and w.l.o.g. the two automata have disjoint state sets, then we can build an automaton  $\mathcal{C}$  whose states are the union of the states of  $\mathcal{A}$  and  $\mathcal{B}$  together with a new state  $\gamma$  and a transition  $\text{pair}(\alpha, \beta) \rightarrow \gamma$  in addition to the transitions from  $\mathcal{A}$  and  $\mathcal{B}$ . We then have  $p_{(\mathcal{C}, \gamma)}(a, b) = p_{(\mathcal{A}, \alpha)}(a) \cdot p_{(\mathcal{B}, \beta)}(b) = p(a) \cdot q(b)$  as required. If  $p_1, \dots, p_k$  are base polynomials for type  $A$  then the base polynomial  $[p_1, \dots, p_k]$  given by

$$[p_1, \dots, p_k]([a_1, \dots, a_m]) := \sum_{1 \leq i_1 < i_2 < \dots < i_k \leq m} p_1(a_{i_1}) \cdots p_k(a_{i_k}),$$

is the generic base polynomial for lists over type  $A$ . Assuming that lists are constructed with the symbols  $[]$  and  $::$  and that, inductively,  $p_i = p_{(\mathcal{A}_i, \alpha_i)}$ , we can build an automaton  $\mathcal{B}$  as the disjoint union of the  $\mathcal{A}_i$  together with  $k + 1$  states  $\beta_0, \beta_1, \dots, \beta_k$ , and transitions:

$$\emptyset :: \beta_i \rightarrow \beta_i \quad \alpha_{i+1} :: \beta_{i+1} \rightarrow \beta_i \quad [] \rightarrow \emptyset,$$

where  $i = 0, \dots, k - 1$ . (Recall that  $\emptyset$  denotes the final state of the canonical automaton  $\emptyset$ , whose inclusion we here tacitly assume.) We obtain  $[p_1, \dots, p_k]([a_1, \dots, a_n]) = p_{(\mathcal{B}, \beta_k)}(a_1 :: a_2 \cdots a_n :: [])$ . Finally, the generic resource polynomial for  $A$ -labelled trees takes the form

$$[p_1, \dots, p_k](t) := [p_1, \dots, p_k](l_t),$$

where  $l_t$  is the list of entries of  $t$  (in the leaves) in depth-first order. Note that we have

$$ls(t) := \begin{cases} \sum_{i=0}^k ([p_1, \dots, p_i](t_1) \cdot & \text{if } ls = [p_1, \dots, p_k], k > 1, \text{ and } t = \text{node}(t_1, t_2) \\ \quad \cdot [p_{i+1}, \dots, p_k](t_2)) & \\ p(a) & \text{if } ls = [p] \text{ and } t = \text{leaf}(a) \\ 1 & \text{if } ls = [] \text{ and } t = \text{leaf}(a) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, assuming the automata  $\mathcal{A}_i$  ( $i = 1, \dots, k$ ) as above, we can construct a new automaton  $\mathcal{B}$  for  $[p_1, \dots, p_k]$  as the disjoint union of the  $\mathcal{A}_i$  together with new states  $\beta_{i,j}$  for  $1 \leq i \leq j \leq k$  and the following transitions:

$$\text{leaf}(\alpha_i) \rightarrow \beta_{i,i+1} \quad \text{leaf}(\emptyset) \rightarrow \beta_{i,i} \quad \text{node}(\beta_{i,t}, \beta_{t,j}) \rightarrow \beta_{i,j},$$

where  $t = i, \dots, j$ . As above, we obtain  $[p_1, \dots, p_k](t) = p_{(\mathcal{B}, \beta_{1,k})}(l_t)$ . Thus the lemma follows.  $\blacktriangleleft$

► **Lemma 7.** *If  $r$  and  $r'$  are resource functions, then  $r + r', r \cdot r' \in \mathcal{RF}$ .*

**Proof.** First, resource functions are obviously closed under addition. With respect to multiplication we employ the linearity of resource functions to see that it suffices to prove the claim for basic functions. Here we argue similar to the proof of Lemma 6 by using a product automata construction.  $\blacktriangleleft$

► **Definition 8.** We define the set of *multivariate basic functions*, denoted as  $\mathcal{BF}(n)$ :

$$\mathcal{BF}(n) := \{v_1, \dots, v_n \mapsto p_{\mathcal{A}_1}(v_1) \cdots p_{\mathcal{A}_n}(v_n) \mid \text{for all } i: v_i \in \mathcal{T}(\mathcal{C}) \text{ and } p_{\mathcal{A}_i} \in \mathcal{BF}\}.$$

We enumerate the set  $\mathcal{BF}(n)$  by sequences of automata, such that  $p_{(\mathcal{A}_1, \dots, \mathcal{A}_n)}(v_1, \dots, v_n) = p_{\mathcal{A}_1}(v_1) \cdots p_{\mathcal{A}_n}(v_n)$ .

In the following sequences of automata (like  $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ ) are sometimes abbreviated as  $\vec{\mathcal{A}}$ , in particular we set  $\vec{\emptyset} = (\emptyset, \dots, \emptyset)$ . Note that  $\mathcal{BF}(1) = \mathcal{BF}$ ; in the following we use  $\mathcal{BF}$  as unique denotation.

► **Lemma 9.** *For all  $p \in \mathcal{BF}(2)$ , there exists  $p' \in \mathcal{BF}$ , such that  $p(v, v) = p'(v)$  for all values  $v$ .*

**Proof.** By definition there exists automata  $\mathcal{A}, \mathcal{B}$  such that  $p(v, v) = p_{\mathcal{A}}(v) \cdot p_{\mathcal{B}}(v)$ . By the previous lemma there exists an automaton  $\mathcal{C}$  such that  $p_{\mathcal{C}}(v) = p_{\mathcal{A}}(v) \cdot p_{\mathcal{B}}(v)$ . Thus we set  $p' := p_{\mathcal{C}}$  to conclude the lemma. ◀

Let  $\mathcal{C}$  denote a set of constructor symbols. A *resource annotation over  $\mathcal{C}$* , or simply *annotation*, is a family  $Q = (q_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$  with  $q_{\mathcal{A}} \in \mathbb{Q}^+$  with all but finitely many of the coefficients  $q_{\mathcal{A}}$  equal to 0. It represents a (finite) linear combination of basic resource functions. We generalise annotation for sequences of terms. An annotation for a sequence of length  $n$  is a family  $Q = (q_{(\mathcal{A}_1, \dots, \mathcal{A}_n)})_{\mathcal{A}_i \in \mathfrak{A}}$  again vanishing almost everywhere. We denote annotations with upper-case letters from the end of the alphabet and use the convention that the corresponding lower-case letter denotes the elements of the family.

► **Definition 10.** The *potential* of a value  $v$  with respect to an annotation  $Q$  (of length 1), that is,  $Q = (q_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$ , is defined as:

$$\Phi(v; Q) := \sum_{\mathcal{A} \in \mathfrak{A}} q_{\mathcal{A}} \cdot p_{\mathcal{A}}(v),$$

where  $p_{\mathcal{A}} \in \mathcal{BF}$ . We generalise this to the potential of a term sequence with respect to an annotation  $\bar{Q} = (q_{(\mathcal{A}_1, \dots, \mathcal{A}_n)})_{\mathcal{A}_i \in \mathfrak{A}}$ :  $\Phi(v_1, \dots, v_n; \bar{Q}) := \sum_{\mathcal{A}_1, \dots, \mathcal{A}_n \in \mathfrak{A}} q_{(\mathcal{A}_1, \dots, \mathcal{A}_n)} \cdot p_{(\mathcal{A}_1, \dots, \mathcal{A}_n)}(v_1, \dots, v_n)$ , where  $p_{(\mathcal{A}_1, \dots, \mathcal{A}_n)} \in \mathcal{BF}(n)$ .

We are ready to generalise the notion of additive shift studied in [10, 9, 12].

► **Definition 11.** Suppose  $Q = (q_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B})})_{\mathcal{A}_i, \mathcal{B} \in \mathfrak{A}}$  denotes a resource annotation of length  $m + 1$ . Let  $c \in \mathcal{C}$  be a constructor symbol of arity  $n$ . The *additive shift for  $c$*  of  $Q$  is an annotation  $\triangleleft_c(Q) = (q'_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B}_1, \dots, \mathcal{B}_n)})_{\mathcal{A}_i, \mathcal{B}_j \in \mathfrak{A}}$  for a sequence of length  $n + m$ , where  $q'_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B}_1, \dots, \mathcal{B}_n)}$  is defined as follows:

$$q'_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B}_1, \dots, \mathcal{B}_n)} := \sum_{\substack{\mathcal{B} = (\mathcal{C}, \mathcal{Q}, \beta, \Delta) \in \mathfrak{A} \\ \text{with } c(\beta_1, \dots, \beta_n) \rightarrow \beta \in \Delta}} q_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B})}, \quad (1)$$

Here  $\mathcal{B}_i = (\mathcal{C}, \mathcal{Q}_{\mathcal{B}_i}, \beta_i, \Delta_{\mathcal{B}_i})$  for each  $i = 1, \dots, n$ .

The correctness of the additive shift operation follows from the next lemma.

► **Lemma 12.** *Let  $v = c(v_1, \dots, v_n)$  be a value and let  $Q = (q_{(\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B})})_{\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{B} \in \mathfrak{A}}$  be an annotation of length  $m + 1$ . Then  $\Phi(w_1, \dots, w_m, v; Q) = \Phi(w_1, \dots, w_m, v_1, \dots, v_n; \triangleleft_c(Q))$ .*

**Proof.** In proof, we restrict to the case  $m = 0$ . Consider the value  $v = c(v_1, \dots, v_n)$ :

$$\begin{aligned}
\Phi(v : Q) &= \sum_{\mathcal{B} \in \mathfrak{A}} q_{\mathcal{B}} \cdot p_{\mathcal{B}}(c(v_1, \dots, v_n)) \\
&= \sum_{\mathcal{B} \in \mathfrak{A}} q_{\mathcal{B}} \cdot \left( \sum_{c(\beta_1, \dots, \beta_n) \rightarrow \beta \in \Delta} p_{(\mathcal{B}, \beta_1)}(v_1) \cdots p_{(\mathcal{B}, \beta_n)}(v_n) \right) \\
&= \sum_{\mathcal{B}_1, \dots, \mathcal{B}_n \in \mathfrak{A}} q'_{(\mathcal{B}_1, \dots, \mathcal{B}_n)} \cdot (p_{\mathcal{B}_1}(v_1) \cdots p_{\mathcal{B}_n}(v_n)) \\
&= \sum_{\mathcal{B}_1, \dots, \mathcal{B}_n \in \mathfrak{A}} q'_{(\mathcal{B}_1, \dots, \mathcal{B}_n)} \cdot p_{(\mathcal{B}_1, \dots, \mathcal{B}_n)}(v_1, \dots, v_n) = \Phi(v_1, \dots, v_n : \triangleleft_c(Q)).
\end{aligned}$$

Here is straightforward to check that  $q'_{(\mathcal{B}_1, \dots, \mathcal{B}_n)}$  is as in (1). Thus the last equation (and the lemma) follows.  $\blacktriangleleft$

Let  $Q = (q_{(\mathcal{A}_1, \dots, \mathcal{A}_n)})_{\mathcal{A}_i \in \mathfrak{A}}$  denote a resource annotation of length  $n$ . Let  $\vec{\mathcal{B}} = (\mathcal{B}_1, \dots, \mathcal{B}_m)$ ; we define the *projection* of  $Q$  with respect to  $\vec{\mathcal{B}}$  to an annotation of length  $\ell < n$ . The projection is denoted as  $\pi_{\vec{\mathcal{B}}}^{\ell}(Q)$ . We set

$$\pi_{\vec{\mathcal{B}}}^{\ell}(Q) := (q'_{(\mathcal{A}_1, \dots, \mathcal{A}_{\ell})})_{\mathcal{A}_i \in \mathfrak{A}},$$

where  $q'_{(\mathcal{A}_1, \dots, \mathcal{A}_{\ell})} = q_{(\mathcal{A}_1, \dots, \mathcal{A}_{\ell}, \mathcal{B}_1, \dots, \mathcal{B}_m)}$  and  $n = \ell + m$ . Suppose  $\Gamma, v_1, v_2 : Q$  denotes an annotated sequence of length  $m + 2$ . Suppose  $v_1 = v_2$  and we want to *share* the values. Then we make use of the operator  $\Upsilon(Q)$  that adapts the potential suitably. The operator is also called *sharing operator*.

► **Lemma 13.** *Let  $\Gamma, v_1, v_2 : Q$  denote an annotated sequence of length  $m + 2$ . Then there exists a resource annotation  $\Upsilon(Q)$  such that  $\Phi(\Gamma, v_1, v_2 : Q) = \Phi(\Gamma, v : \Upsilon(Q))$ , if  $v_1 = v_2 = v$ .*

**Proof.** This follows from Lemma 9.  $\blacktriangleleft$

Let  $Q$  be an annotation over the set of constructor symbols  $\mathcal{C}$  and let  $K \in \mathbb{Q}^+$ . Then we define  $Q' := Q + K$  as follows:  $Q' = (q'_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$ , where  $q'_{\emptyset} := q_{\emptyset} + K$  and for all  $\mathcal{A} \neq \emptyset$ ,  $q'_{\mathcal{A}} := q_{\mathcal{A}}$ . We define the comparison  $\leq$  of two annotations  $Q, Q'$  over  $\mathcal{C}$  pointwise:  $Q \leq Q'$  if for all  $\mathcal{A} \in \mathfrak{A}$ :  $q_{\mathcal{A}} \leq q'_{\mathcal{A}}$ .

## 4 Amortised Cost Analysis

The rest of the paper is essentially an adaptation of the type systems given in [10, 9, 12]. There are no essential surprises but care must be taken with the rule for the evaluation of non-constructor terms which is essentially a combination of the let rule and the function application rule from [9].

Let  $\Gamma$  denote a sequence of variables  $x_1, \dots, x_n$ ;  $\Gamma$  is called a *variable context* or simply a *context*. The *length*  $n$  of  $\Gamma$  is denoted as  $|\Gamma|$ . A *resource annotation for  $\Gamma$*  or simply *annotation* is an annotation of length  $|\Gamma|$ . Now let  $Q$  be a resource annotation for  $\Gamma$ , let  $Q'$  be a resource annotation, let  $t$  be a term and let  $v$  denote its normalform. Then the typing judgement  $\Gamma : Q \vdash t : Q'$  expresses that for *bounded TRS* (see Definition 15) the potential of  $\Gamma$  with respect to  $Q$  is sufficient to pay for the total cost  $m$  of the evaluation  $\sigma \stackrel{m}{\mapsto} t \Rightarrow v$  ( $\sigma$  a substitution), plus the potential of the value  $v$  with respect to  $Q'$ .

We comment on the inference rules in Figure 2. For the majority of the typing rules their definition is straightforward. Consider exemplary the typing rule for constructors

$c \in \mathcal{C}$ . Due to Lemma 6 the potential of a value  $c(x_1\sigma, \dots, x_n\sigma)$  equals the potential of the variable context modulo an additive shift with respect to  $c$ . This is precisely expressed in the corresponding type rule. On the other hand the composition rule is more involved. Consider the judgement  $\Gamma_1, \dots, \Gamma_n : Q \vdash f(t_1, \dots, t_n) : Q'$ . Observe that on the basis of the sharing rule we can assume that  $t = f(t_1, \dots, t_n)$  is linear and thus the variable context splits into its parts  $\Gamma_i$  ( $i = 1, \dots, n$ ). The intuition of the composition is that the potential of  $\Gamma_1, \dots, \Gamma_n$  (represented through the annotation  $Q$ ) should be distributed over the evaluations of all arguments  $t_i$  and the evaluation of the function  $f$ , in a way such that the interdependency of the bounds is preserved. This is achieved by type checking each of the arguments and the context individually and verifying that all issuing annotations are consistent with each other.

In order to see how this works, we detail some of the constraints. First, consider  $\pi_{\Gamma_1}^{\vec{j}_1}(Q) = P_1(\vec{j}_1)$ . Here  $\vec{j}_1$  denotes an index for  $\Gamma_2, \dots, \Gamma_n$ . The projection asserts that the resources annotated in  $Q$  are projected to  $\Gamma_1$ , so that the typing  $\Gamma_1 : P_1(\vec{j}_1) \vdash^{(\text{cf})} t_1 : R_1(\vec{j}_1)$  is realisable for every index  $\vec{j}_1$ . The constraint  $p_\ell^{i+1, \vec{j}_{i+1}} = r_k^{i, \vec{j}_i}$ , where  $1 \leq i < n$ ,  $\ell$  an index for  $\Gamma_i$  and  $k \in I$ , guarantees that the annotations for the remaining contexts  $\Gamma_i$ ,  $i > 1$  are consistent with each other. Finally, constraint  $\pi_{x_n}^{\vec{j}_n}(S) = R_n(\vec{j}_n)$  links the potential after the evaluation of the last argument  $t_n$  consistently with the annotation  $S$  for the variable context  $x_1, \dots, x_n$ .

The type system given requires the use of *cost-free* judgements. Here the rules of the given TRS are considered as weak rules that are not taken into account in the complexity evaluation (see Definition 15).

► **Definition 14.** An *annotated signature*  $\overline{\mathcal{F}}$  is a mapping from  $\mathcal{D}$  to sets of pairs of resource annotations:

$$\overline{\mathcal{F}}(f) := \left\{ Q \rightarrow Q' \mid \begin{array}{l} \text{if the arity of } f \text{ is } n, Q \text{ is an annotation of length } \\ n \text{ and } Q' \text{ a resource annotation} \end{array} \right\}.$$

Usually, we confuse the signature and the annotated signature and denote the latter simply as  $\mathcal{F}$ .

The set of indices of an annotation for a context  $\Gamma$  is defined as follows:

$$I(\Gamma) := \{(\mathcal{A}_1, \dots, \mathcal{A}_n) \mid \text{if } \mathcal{A} \in \mathfrak{A} \text{ and } |\Gamma| = n\}.$$

We also set  $I := \mathfrak{A}$ . Let  $Q$  be a resource annotation of length  $n$ , let  $\Gamma = x_1, \dots, x_\ell$  be a variable context, and let  $\vec{i} = \mathcal{B}_1, \dots, \mathcal{B}_m$  be an index of length  $m$ . We define the *projection with respect to*  $\Gamma$ :  $\pi_{\Gamma}^{\vec{i}}(Q) := \pi_{|\Gamma|}^{\vec{i}}(Q)$ .

Recall that any rewrite rule  $l \rightarrow r \in \mathcal{R}$  can be written as  $f(l_1, \dots, l_n) \rightarrow r$  with  $l_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$  and that no variable occurs twice in  $l$ . Similar to the notion of well-typed TRSs in [12], we introduce *bounded* TRSs.

► **Definition 15.** Let  $f(l_1, \dots, l_n) \rightarrow r$  be a rewrite rule in  $\mathcal{R}$  and let  $\text{Var}(f(l_1, \dots, l_n)) = \{x_1, \dots, x_\ell\}$ . Suppose  $f$  is defined and let  $Q \rightarrow Q' \in \mathcal{F}(f)$ . Suppose further, for any such annotation  $Q \rightarrow Q'$  we can derive

$$x_1, \dots, x_\ell : P \vdash r : Q', \tag{2}$$

where  $P + 1 = R$  and  $R$  is obtained by iteratively applying shift operations with respect to the constructors occurring in  $\bigcup_{i=1}^n l_i$  to  $Q$ . Then we say  $f$  is *bounded* wrt.  $\mathcal{F}$ . On the other hand  $f$  is *weakly bounded* if the rewrite step is not counted, that is, the judgement  $x_1, \dots, x_\ell : R \vdash r : Q'$  is asserted instead of (2). A TRS  $\mathcal{R}$  over  $\mathcal{F}$  is *bounded* (*weakly bounded*) if any defined  $f$  is bounded (weakly bounded).

$$\begin{array}{c}
\frac{}{x:Q \vdash x:Q} \qquad \frac{\Gamma: \pi_{\Gamma}^{\emptyset}(Q) \vdash t:Q'}{\Gamma, x:Q \vdash t:Q'} \\
\frac{f \in \mathcal{D} \quad Q \rightarrow Q' \in \mathcal{F}(f) \quad f \text{ is } n\text{-ary}}{x_1, \dots, x_n:Q \vdash f(x_1, \dots, x_n):Q'} \qquad \frac{c \in \mathcal{C} \quad c \text{ is } n\text{-ary}}{x_1, \dots, x_n:\triangleleft_c(Q) \vdash c(x_1, \dots, x_n):Q} \\
\forall \vec{j}_1, \dots, \vec{j}_n \quad \vec{j}_i \in I^{i-1} \times I(\Gamma_{i+1}) \times \dots \times I(\Gamma_n) \\
P_i(\vec{j}_i) = (p^{\ell, \vec{j}_i})_{\ell \in I(\Gamma_i)} \quad R_i(\vec{j}_i) = (r^{k, \vec{j}_i})_{k \in I} \\
p_{\ell}^{i+1, \vec{j}_{i+1}} = r_{k}^{i, \vec{j}_i} \text{ if } \vec{j}_i = a @ \ell @ b \text{ and } \vec{j}_{i+1} = a @ k @ b \\
\frac{\pi_{\Gamma_1}^{\vec{j}_1}(Q) = P_1(\vec{j}_1) \quad \pi_{x_n}^{\vec{j}_n}(S) = R_n(\vec{j}_n) \quad x_1, \dots, x_n:S \vdash f(x_1, \dots, x_n):Q' \quad \Gamma_1: P_1(\vec{j}_1) \vdash^{(\text{cf})} t_1: R_1(\vec{j}_1) \quad \dots \quad \Gamma_n: P_n(\vec{j}_n) \vdash^{(\text{cf})} t_n: R_n(\vec{j}_n)}{\Gamma_1, \dots, \Gamma_n: Q \vdash f(t_1, \dots, t_n):Q'} \\
\frac{\Gamma, x, y:Q \vdash t[x, y]:Q' \quad x, y \text{ are fresh}}{\Gamma, z:\Upsilon(Q) \vdash t[z, z]:Q'} \qquad \frac{\Gamma, x:P \vdash t:P' \quad P \leq Q \quad P' \geq Q'}{\Gamma, x:Q \vdash t:Q'}
\end{array}$$

In the composition rule  $a$  and  $b$  denote suitable chosen indices, where  $@$  denotes concatenation of indices. Further, the judgement  $\Gamma_i: P_i(\vec{j}_i) \vdash^{(\text{cf})} t_i: R_i(\vec{j}_i)$  abbreviates that  $\Gamma_i: P_i(\vec{\emptyset}) \vdash t_i: R_i(\vec{\emptyset})$  and  $\Gamma_i: P_i(\vec{j}_i) \vdash^{(\text{cf})} t_i: R_i(\vec{j}_i)$  for all  $\vec{j}_i \neq \vec{\emptyset}$  and  $i = 1, \dots, n$ .

■ **Figure 2** Multivariate Analysis of Term Rewrite Systems.

$$\frac{\frac{y_1: P_1(j) \vdash^{(\text{cf})} y_1: R_1(j) \quad \frac{\times: P_2(i) \rightarrow R_2(i)}{x, y_2: P_2(i) \vdash^{(\text{cf})} x \times y_2: R_2(i)} \quad \frac{+: S \rightarrow S'}{u, v: S \vdash u + v: S'}}{y_1, x, y_2: T \vdash y_1 + (x \times y_2): M'} \quad \frac{}{x, y: \overline{M}_2 \vdash y + (x \times y): M'}}{}$$

Here  $i \in I$ , and  $j \in I(x, y_2)$ .

■ **Figure 3** Derivation of  $x, y: \overline{M}_2 \vdash y + (x \times y): M'$ .

Let  $\sigma$  denote a substitution,  $\Gamma = x_1, \dots, x_n$  a context and  $Q$  a resource annotation. Then we define the *potential* of  $\Gamma: Q$  with respect to  $\sigma$ :

$$\Phi(\sigma, \Gamma: Q) := \Phi(x_1\sigma, \dots, x_n\sigma: Q).$$

Note that the above definition employs the shift operator in a similar way as in [9], where this is part of the type system in the case for pattern matching.

Before we state our main result, we exemplify the use of the type system on a simple (but clarifying) example. Consider the following TRS  $\mathcal{R}_{\times}$ , restricting our motivating example (see page 243).

$$\begin{array}{ll}
0 + y \rightarrow y & \mathfrak{s}(x) + y \rightarrow \mathfrak{s}(x + y) \\
0 \times y \rightarrow 0 & \mathfrak{s}(x) \times y \rightarrow y + (x \times y).
\end{array}$$

We consider the canonical automaton  $\emptyset$  for the constructor symbols  $\{0, \mathfrak{s}\}$  together with the automaton  $\mathcal{A}$  defined as follows:

$$\mathcal{A}: \quad 0 \rightarrow \emptyset \qquad \mathfrak{s}(\emptyset) \rightarrow \alpha \qquad \mathfrak{s}(\emptyset) \rightarrow \emptyset \qquad \mathfrak{s}(\alpha) \rightarrow \alpha$$

We show that  $\mathcal{R}_\times$  is bounded with respect to the following annotated signatures:

$$\begin{aligned} +: & \{ (p_{(\emptyset,\emptyset)}, p_{(\mathcal{A},\emptyset)}) \rightarrow (p'_{\emptyset}) \mid p_{(\emptyset,\emptyset)} \geq 1, p_{(\mathcal{A},\emptyset)} \geq 1, p_{(\emptyset,\emptyset)} - 1 \geq p'_{\emptyset} \} \\ \times: & \left\{ \begin{pmatrix} m_{(\emptyset,\emptyset)} & m_{(\mathcal{A},\emptyset)} \\ 0 & m_{(\mathcal{A},\mathcal{A})} \end{pmatrix} \rightarrow (m'_{\emptyset}, m'_{\mathcal{A}}) \mid \begin{array}{l} m_{(\emptyset,\emptyset)} \geq 1, m_{(\mathcal{A},\emptyset)} \geq 2, \\ m_{(\mathcal{A},\mathcal{A})} \geq 1, m_{(\emptyset,\emptyset)} - 1 \geq m'_{\emptyset} \end{array} \right\}. \end{aligned}$$

Each of the four rules induced one of the following demands, cf. (2). (Recall that we denote annotations with upper-case letters and the elements of the families with the corresponding lower-case letters.)

$$y: \bar{P}_1 \vdash y: P' \qquad \bar{P}_1 + 1 = \triangleleft_0(P) \qquad (3)$$

$$x, y: \bar{P}_2 \vdash s(x + y): P' \qquad \bar{P}_2 + 1 = \triangleleft_s(P) \qquad (4)$$

$$y: \bar{M}_1 \vdash 0: M' \qquad \bar{M}_1 + 1 = \triangleleft_0(M) \qquad (5)$$

$$x, y: \bar{M}_2 \vdash y + (x \times y): M' \qquad \bar{M}_2 + 1 = \triangleleft_s(M) \qquad (6)$$

It is not difficult to see that (3) and (4) induce the constraints  $p_{(\emptyset,\emptyset)} \geq 1$ ,  $p_{(\mathcal{A},\emptyset)} \geq 1$ , and  $p_{(\emptyset,\emptyset)} - 1 \geq p'_{\emptyset}$ , and to ease the presentation we set  $p'_{\mathcal{A}} = 0$ . Furthermore, the typing judgement (5) yields the constraint  $m_{(\emptyset,\emptyset)} - 1 \geq m'_{\emptyset}$ .

Of more interest is the precise derivation of (6). In this derivation we make use of the weakly boundedness of  $\mathcal{R}_\times$  with respect to the cost-free signature  $+: (p_{(\emptyset,\emptyset)}) \rightarrow p'_{\emptyset}$  and  $\times: (m_{(\emptyset,\emptyset)}) \rightarrow m'_{\emptyset}$ , where it suffices to demand that  $p_{(\emptyset,\emptyset)} \geq p'_{\emptyset}$  and  $m_{(\emptyset,\emptyset)} \geq m'_{\emptyset}$ . We obtain the following derivation in Figure 3. This derivation induces the constraint  $\Upsilon(T) = \bar{M}_2$ , as first (reading bottom-up) we employ a sharing rule. The composition rule yields the constraints  $\pi_{y_1}^j(T) = P_1(j)$  ( $j \in I(x, y_2)$ ),  $\pi_v^i(S) = R_2(i)$  ( $i \in I$ ), and  $r_i^{1,j} = p_j^{2,i}$ , where  $R_1(j) = (r^{1,j})_{i \in I}$  and  $P_2(i) = (p^{2,i})_{j \in I(x, y_2)}$ . Finally, the axioms yield the constraints  $P_1(j) \geq R_i(j)$  for  $j \in I(x, y_2)$  as well as the indicated conditions on the signature. It is tedious, but straightforward to check that these constraints can be met for the given annotations. Thus,  $\times$  can be typed with the annotation  $m_{(\emptyset,\emptyset)} = 1$ ,  $m_{(\mathcal{A},\emptyset)} = 2$ , and  $m_{(\mathcal{A},\mathcal{A})} = 1$  which yields the bound  $\text{dh}(\mathbf{m} \times \mathbf{n}, \rightarrow_{\mathcal{R}_\times}) \leq m \cdot n + 2m + 1$ .

It is instructive to depict the annotation  $\bar{M}_2$  in matrix format, which allows a simple expression of the  $\triangleleft_s$ -operator.

$$\bar{M}_2 = \begin{pmatrix} \bar{m}_{\emptyset,\emptyset} & \bar{m}_{\mathcal{A},\emptyset} \\ \bar{m}_{\emptyset,\mathcal{A}} & \bar{m}_{\mathcal{A},\mathcal{A}} \end{pmatrix} = \begin{pmatrix} m_{\emptyset,\emptyset} + m_{\mathcal{A},\emptyset} - 1 & m_{\mathcal{A},\emptyset} \\ m_{\mathcal{A},\mathcal{A}} & m_{\mathcal{A},\mathcal{A}} \end{pmatrix} = \triangleleft_s(M).$$

Then it becomes apparent that the annotation  $\bar{M}_2$  is the result of adding the auxiliary annotation

$$\begin{pmatrix} m_{\mathcal{A},\emptyset} - 1 & 0 \\ m_{\mathcal{A},\mathcal{A}} & 0 \end{pmatrix}, \qquad (7)$$

to the annotation for multiplication. Intuitively the type system asserts that we can split the annotation  $\bar{M}_2$  into an annotation  $M$  that pays for the recursive call and the annotation (7) that pays for the call to addition.

► **Theorem 16.** *Let  $\mathcal{R}$  be bounded. Suppose  $\Gamma: Q \vdash t: Q'$  and  $\sigma \stackrel{m}{\vdash} t \Rightarrow v$ . Then  $\Phi(\sigma, \Gamma: Q) - \Phi(v: Q') \geq m$ .*

**Proof.** Let  $\Pi$  be the proof deriving  $\sigma \stackrel{m}{\vdash} t \Rightarrow v$  and let  $\Xi$  be the proof of  $\Gamma: Q \vdash t: Q'$ . The proof of the theorem proceeds by main-induction on the length of  $\Pi$  and by side-induction

on the length of  $\Xi$ . We consider the case for composition. Suppose the last rule in  $\Pi$  has the form

$$\frac{\sigma \uplus \rho \mid^{m_0} f(x_1, \dots, x_n) \Rightarrow v \quad \sigma \mid^{m_i} t_i \Rightarrow v_i \quad i = 1, \dots, n \quad m = \sum_{i=0}^n m_i}{\sigma \mid^m t \Rightarrow v} .$$

We can assume that  $t = f(x_1, \dots, x_n)$  is linear, due the presence of the share operator. In proof we restrict to the case where  $n = 2$ . Hence the last rule in the type inference  $\Xi$  is of the following form.

$$\frac{\Gamma_1 : P(j) \vdash^{(\text{cf})} t_1 : \bar{P}(j) \quad \Gamma_2 : R(i) \vdash^{(\text{cf})} t_2 : \bar{R}(i) \quad x, y : S \vdash f(x, y) : Q'}{\Gamma_1, \Gamma_2 : Q \vdash f(t_1, t_2) : Q'}$$

The following conditions hold, where we use the notations  $P(j) = (p_i^j)_{i \in I(\Gamma_1)}$ ,  $\bar{P}(j) = (\bar{p}_i^j)_{i \in I(\Gamma_1)}$ ,  $R(i) = (r_j^i)_{j \in I(\Gamma_2)}$ , and  $\bar{R}(i) = (\bar{r}_j^i)_{j \in I(\Gamma_2)}$

$$\forall j \in I(\Gamma_2) \quad \pi_{\Gamma_1}^j(Q) = P(j) \quad \forall i \in I \quad \pi_y^i(S) = \bar{R}(i) \quad \forall i, j \quad \bar{p}_i^j = r_j^i . \quad (8)$$

By induction hypothesis, we have (i)  $\Phi(\sigma \uplus \rho, x, y : R) - \Phi(v : Q') \geq m_0$ , (ii) for all  $j \in I(\Gamma_2)$ :  $\Phi(\sigma, \Gamma_1 : P(j)) - \Phi(v_1 : \bar{P}(j)) \geq m_1$ , and (iii) for all  $i \in I$ :  $\Phi(\sigma, \Gamma_2 : R(i)) - \Phi(v_2 : \bar{R}(i)) \geq m_2$ . Let  $\vec{x} := x_1, \dots, x_n$ , where  $\text{Var}(t_1) = \{x_1, \dots, x_n\}$  and let  $\vec{y} := y_1, \dots, y_n$  with  $\text{Var}(t_2) = \{y_1, \dots, y_n\}$ . The theorem follows by a straightforward calculation:

$$\begin{aligned} \Phi(\sigma, \Gamma_1, \Gamma_2 : Q) &= \sum_{i \in I(\Gamma_1), j \in I(\Gamma_2)} q_{(i,j)} \cdot p_i(\vec{x}\sigma) \cdot p_j(\vec{y}\sigma) \\ &= \sum_{j \in I(\Gamma_2)} p_j(\vec{y}\sigma) \cdot \left( \sum_{i \in I(\Gamma_1)} p_i^j \cdot p_i(\vec{x}\sigma) \right) \\ &\geq \sum_{j \in I(\Gamma_2)} p_j(\vec{y}\sigma) \cdot \left( \sum_{i \in I} \bar{p}_i^j \cdot p_i(v_1) \right) + m_1 \\ &= \sum_{i \in I} p_i(v_1) \cdot \left( \sum_{j \in I(\Gamma_2)} r_j^i \cdot p_j(\vec{y}\sigma) \right) + m_1 \\ &\geq \sum_{i \in I} p_i(v_1) \cdot \left( \sum_{j \in I} \bar{r}_j^i \cdot p_j(v_2) \right) + m_1 + m_2 \\ &\geq \sum_{i \in I} q'_i \cdot p_i(v) + \sum_{i=0}^2 m_i = \Phi(v : Q') + \sum_{i=0}^2 m_i . \end{aligned}$$

Here we tacitly employ the conditions (8) together with the induction hypothesis which is employed in line 3, 5, and 6.  $\blacktriangleleft$

The following corollary to the theorem is immediate.

► **Corollary 17.** *Assume the conditions of the theorem. If additionally for all values  $v$  and annotations  $Q$ ,  $\Phi(v : Q) \in \mathcal{O}(n^k)$ , where  $n = |v|$ , then  $\text{rc}_{\mathcal{R}}(n) \in \mathcal{O}(n^k)$ .*

► **Remark.** Recall that we restrict to completely defined, orthogonal constructor TRSs. It is not difficult to see that Theorem 16 (and its corollary) generalise to the case where completely definedness is dropped. While completely definedness is essential for the correctness of the

big-step semantics presented in Figure 1, the proof of Theorem 16 extends with relative ease. The induction on the length of  $\Pi: \sigma \stackrel{m}{\vdash} t \Rightarrow v$  is replaced by an induction on the length of an innermost derivation  $D: t\sigma \rightarrow_{\mathcal{R}}^+ v$ .

Finally, we consider the motivating TRS  $\mathcal{R}_{\text{dyade}}$ . Based on the above example (page 251) it remains to consider the remaining four rules of  $\mathcal{R}_{\text{dyade}}$ :

$$\begin{aligned} \text{mult}(n, []) &\rightarrow [] & \text{mult}(n, x :: xs) &\rightarrow (n \times x) :: \text{mult}(n, xs) \\ \text{dyade}([], ls) &\rightarrow [] & \text{dyade}(x :: xs, ls) &\rightarrow \text{mult}(x, ls) :: \text{dyade}(xs, ls). \end{aligned}$$

We consider the following automata  $\emptyset$ ,  $\mathcal{A}$ ,  $\mathcal{B}$  and  $\mathcal{C}$ , where  $\emptyset$  denotes the canonical automata for  $\{0, s, [], ::\}$  and  $\mathcal{A}$  is defined as on page 251. The definition of  $\mathcal{B}$  and  $\mathcal{C}$  is given below.

$$\begin{aligned} \mathcal{B}: \quad 0 &\rightarrow \emptyset & s(\emptyset) &\rightarrow \emptyset & [] &\rightarrow \emptyset & \emptyset :: \emptyset &\rightarrow \beta & \emptyset :: \beta &\rightarrow \beta \\ \mathcal{C}: \quad 0 &\rightarrow \emptyset & s(\emptyset) &\rightarrow \emptyset & s(\emptyset) &\rightarrow \alpha & s(\alpha) &\rightarrow \alpha & [] &\rightarrow \emptyset \\ & & \alpha :: \emptyset &\rightarrow \gamma & \emptyset &:: \gamma &\rightarrow \gamma. \end{aligned}$$

Note that  $p_{\emptyset}(v) = 1$ ,  $p_{\mathcal{A}}(\mathbf{n}) = n$ ,  $p_{\mathcal{B}}(l) = |l|$ , and  $p_{\mathcal{C}}(l) = \sum_{i=1}^m n_i$ , where  $l = [\mathbf{n}_1, \dots, \mathbf{n}_m]$ .

We make use of a similar denotation of the annotations as in the example on page 251 and set  $\text{mult}: M \rightarrow (m'_{\emptyset, \emptyset})$ , where  $M = (m_{\mathcal{A}_1, \mathcal{A}_2})_{\mathcal{A}_1 \in \{\emptyset, \mathcal{A}\}, \mathcal{A}_2 \in \{\emptyset, \mathcal{B}, \mathcal{C}\}}$  and similarly for  $\text{dyade}: D \rightarrow (d'_{\emptyset, \emptyset})$ , where  $D = (d_{\mathcal{A}_1, \mathcal{A}_2})_{\mathcal{A}_1 \in \{\emptyset, \mathcal{B}, \mathcal{C}\}, \mathcal{A}_2 \in \{\emptyset, \mathcal{B}, \mathcal{C}\}}$ . Thus we can assert the signature of  $\text{mult}$  and  $\text{dyade}$  as follows:

$$\text{mult}: \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 1 \end{pmatrix} \rightarrow (0) \quad \text{dyade}: \begin{pmatrix} 1 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow (0).$$

Considering  $\text{dyade}(x :: xs, ls) \rightarrow \text{mult}(x, ls) :: \text{dyade}(xs, ls)$ , we study the effects of the additive shift on  $D$ . Let  $\bar{D} := \triangleleft_{::}(D) - 1$  such that  $\bar{D} = (\bar{d}_{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3})_{\mathcal{A}_1 \in \{\emptyset, \mathcal{A}\}, \mathcal{A}_2, \mathcal{A}_3 \in \{\emptyset, \mathcal{B}, \mathcal{C}\}}$ . For  $\mathcal{A}_1 = \mathcal{A}$ , then  $\bar{d}_{\mathcal{A}, \mathcal{A}_2, \mathcal{A}_3}$  vanishes almost everywhere, but  $\bar{d}_{\mathcal{A}, \emptyset, \mathcal{B}} = d_{\mathcal{C}, \mathcal{B}} = 1$  and  $\bar{d}_{\mathcal{A}, \emptyset, \mathcal{C}} = d_{\mathcal{C}, \mathcal{C}} = 1$ . To ease the presentation, we ignore these positive annotations and only consider the restricted annotation  $(\bar{d}_{\emptyset, \mathcal{A}_2, \mathcal{A}_3})_{\mathcal{A}_2, \mathcal{A}_3 \in \{\emptyset, \mathcal{B}, \mathcal{C}\}}$ . This annotation is again representable as a matrix and typability of the rule follows, as we can decompose the matrix suitably:

$$\begin{pmatrix} 2 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

In order to estimate  $\Phi(\sigma, xs, ys: D)$  for arbitrary  $\sigma$  we analyse the base functions of the considered automata. Thus the analysis yields the essentially optimal bound of the execution of  $\text{dyade}(ls_1, ls_2)$  as a multiplicative bound in the sum of the values of the first and second list  $ls_1 (= xs\sigma)$  and  $ls_2 (= ys\sigma)$ , respectively, together with a linear factor.

## 5 Conclusion

We have presented a novel amortised resource analysis in the context of term rewrite systems. The method is represented in an inference system akin to a type system and can give rise to polynomial bounds on the innermost runtime complexity of the analysed rewrite system. The crucial feature of the inference system is the admittance of multivariate bounds in the

context of arbitrary data structures in a completely uniform way. This extends our earlier univariate resource analysis of typed term rewrite systems and continues our program of applying automated amortised resource analysis to rewriting.

We already briefly commented on the differences of the here presented study to our earlier work [12] in the introduction. As far as we can tell this and the present result are currently the only attempts to lift amortised cost analysis to rewriting or provide such a study in the context of arbitrary constructor-defined datastructures. Hoffmann and Shao provide in [11] a multivariate amortised analysis of integers and arrays that extend upon [10]. These language extensions are also provided in RaML. However, the treatment still appears to be ad-hoc and does not provide a similar uniform framework than ours. Further we mention some general work on automated resource analysis. Albert et al. [1] underlies COSTA, an automated tool for the resource analysis of Java programs. Sinn et al. provide in [15] related approaches for the runtime complexity analysis of C programs, incorporated into LOOPUS. Very recently Brockschmidt et al. [5] have provided a runtime complexity analysis of integer programs, taking also size considerations into account. Basic steps for a modular complexity framework for rewrite systems have been established in [3]. Finally, the RaML prototype [10] provides an automated potential-based resource analysis for various resource bounds of functional programs. For term rewriting AProVE [7] and TCT [2] are the most powerful tools for complexity analysis of rewrite systems as witnessed during last year's termination competition.<sup>1</sup>

In future work we will clarify the automatability of the method. We expect that by restricting the number of states and the format of those tree automata  $\mathcal{A}$ , whose annotation do not vanish, we can reduce inference of annotations to linear constraint solving in much the same way as in [10]. More challenging would be a combination of linear programming and combinatorial constraint solving to infer the best possible structure of automata.

---

## References

- 1 E. Albert, P. Arenas, S. Genaim, G. Puebla, and G. Román-Díez. Conditional termination of loops over heap-allocated data. *Sci. Comput. Program.*, 92:2–24, 2014.
- 2 M. Avanzini and G. Moser. Tyrolean complexity tool: Features and usage. In *Proc. 24th RTA*, volume 21 of *LIPICs*, pages 71–80, 2013.
- 3 M. Avanzini and G. Moser. A combination framework for complexity. *IC*, 2015. To appear.
- 4 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- 5 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. 20th TACAS*, volume 8413 of *LNCS*, pages 140–155, 2014.
- 6 H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree automata techniques and applications*, 2007.
- 7 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.
- 8 N. Hirokawa and G. Moser. Automated complexity analysis based on context-sensitive rewriting. In *Proc. of Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 257–271, 2014.

---

<sup>1</sup> See <http://nfa.imn.htwk-leipzig.de/termcomp/competition/20>.

- 9 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14, 2012.
- 10 J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In *Proc. 24th CAV*, volume 7358 of *LNCS*, pages 781–786, 2012.
- 11 J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. In *Proc. 12th FLOPS*, volume 8475 of *LNCS*, pages 152–168, 2014.
- 12 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.
- 13 H. Seidl. On the finite degree of ambiguity of finite tree automata. *Acta Informatica*, 26(6):527–542, 1989.
- 14 H. Seidl. Ambiguity, Validity, and Costs. Technical report, TU München, 1992. Habilitation Thesis.
- 15 M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.
- 16 TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.