Multivariate Amortized Resource Analysis

Jan Hoffmann Klaus Aehlig Martin Hofmann

Ludwig-Maximilians-Universität München

{jan.hoffmann,aehlig,martin.hofmann}@ifi.lmu.de

Abstract

We study the problem of automatically analyzing the worst-case resource usage of procedures with several arguments. Existing automatic analyses based on amortization, or sized types bound the resource usage or result size of such a procedure by a sum of unary functions of the sizes of the arguments.

In this paper we generalize this to arbitrary multivariate polynomial functions thus allowing bounds of the form mn which had to be grossly overestimated by $m^2 + n^2$ before. Our framework even encompasses bounds like $\sum_{i,j \leq n} m_i m_j$ where the m_i are the sizes of the entries of a list of length n.

This allows us for the first time to derive useful resource bounds for operations on matrices that are represented as lists of lists and to considerably improve bounds on other super-linear operations on lists such as longest common subsequence and removal of duplicates from lists of lists. Furthermore, resource bounds are now closed under composition which improves accuracy of the analysis of composed programs when some or all of the components exhibit super-linear resource or size behavior.

The analysis is based on a novel multivariate amortized resource analysis. We present it in form of a type system for a simple first-order functional language with lists and trees, prove soundness, and describe automatic type inference based on linear programming.

We have experimentally validated the automatic analysis on a wide range of examples from functional programming with lists and trees. The obtained bounds were compared with actual resource consumption. All bounds were asymptotically tight, and the constants were close or even identical to the optimal ones.

Categories and Subject Descriptors F.3.2 [*Logics And Meanings Of Programs*]: Semantics of Programming Languages—Program Analysis; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Performance, Languages, Theory, Reliability

Keywords Functional Programming, Static Analysis, Amortized Analysis, Resource Consumption, Quantitative Analysis

1. Introduction

A primary feature of a computer program is its quantitative performance characteristics: the amount of resources like time, memory and power the program needs to perform its task.

Ideally, it should be possible for an experienced programmer to extrapolate from the source code of a well-written program to its *asymptotic* worst-case behavior. But it is often insufficient to determine the asymptotic behavior of program only. A conservative estimation of the resource consumption for a specific input or a comparison of two programs with the same asymptotic behavior require instead *concrete upper bounds for specific hardware*. That is to say, closed functions in the sizes of the program's inputs that bound the number of clock cycles or memory cells used by the program for inputs of these sizes on a given system.

Concrete worst-case bounds are particularly useful in the development of embedded systems and hard real-time systems. In the former, one wants to use hardware that is *just good enough* to accomplish a task in order to produce a large number of units at lowest possible cost. In the latter, one needs to guarantee specific worst-case running times to ensure the safety of the system.

The manual determination of such bounds is very cumbersome. Cf., e.g., the careful analyses carried out by Knuth in *The Art of Computer Programming* where he pays close attention to the concrete and best possible values of constants for the MIX architecture. Not everyone commands the mathematical ease of Knuth and even he would run out of steam if he had to do these calculations over and over again while going through the debugging loops of program development. In short, derivation of precise bounds by hand appears to be unfeasible in practice in all but the simplest cases.

As a result, *automatic methods for static resource analysis* are highly desirable and have been the subject of extensive research. On the one hand there is the large field of WCET (worst-case execution time) analysis [27] that is focused on (yet not limited to) the run-time analysis of sequential code without loops taking into account low-level features like hardware caches and instruction pipelines. On the other hand there is an active research community that employs type systems and abstract interpretation to deal with the analysis of loops, recursion and data structures [15, 3, 24].[1]

In this paper we continue our work [16] on the resource analysis of programs with recursion and inductive data structures. Our approach is as follows. 1. We consider *Resource Aware ML (RAML)*, a first-order fragment of OCAML that features integers, lists, binary trees, and recursion. 2. We define a big-step operational semantics that *formalizes the actual resource consumptions* of programs. It is parametrized with a resource metric that can be directly related to the compiled assembly code for a specific system architecture [23].[2] 3. We describe an elaborated *resource-parametric type system* whose type judgments establish concrete worst-case bounds in terms of closed, easily understood formulas. The type system allows for an efficient and completely automatic inference algorithm that is based on linear programming. 4. We *prove* the non-trivial soundness of the derived resource bounds with respect to the big-

[1] See §8 for a detailed overview of the state of the art.

[2] To obtain clock-cycle bounds for atomic steps one has to employ WCET tools [23].

step operational semantics. 5. We verify the practicability of our approach with a publically available implementation and a reproducible *experimental evaluation*.

As pioneered by Hofmann and Jost [18] to analyze the heap-space consumption of first-order functional programs, our type system relies on the potential-method of amortized analysis to take into account the interactions between different parts of a computation. This technique has been successfully applied to object-oriented programs [19, 20], to generic resource metrics [23, 7], to polymorphic and higher-order programs [24], and to Java-like bytecode by means of separation logic [4]. The main limitation shared by these analysis systems is their *restriction to linear resource bounds* which can be efficiently reduced to solving linear constraints.

A recently discovered technique [16, 17] yields an automatic amortized analysis for polynomial bounds while still relying on linear constraint solving only. The resulting extension of the linear system [18, 23] efficiently computes resource bounds for first-order functional programs that are sums $\sum p_i(n_i)$ of univariate polynomials p_i. For instance, it automatically infers evaluation-step bounds for the sorting algorithms quick sort and insertion sort that exactly match the measured worst-case behavior of the functions [17]. The computation of these bounds takes less then a second.

This analysis system for polynomial bounds has, however, two drawbacks that hamper the automatic computation of bounds for larger programs. First, many functions with multiple arguments that appear in practice have *multivariate* cost characteristics like $m \cdot n$. Secondly, if data from different sources is interlinked in a program then multivariate bounds like $(m+n)^2$ arise even if all functions have a univariate resource behavior. In these cases the analysis fails, or the bounds are hugely over-approximated by $3m^2 + 3n^2$.

To overcome these drawbacks, this paper presents an *automatic type-based amortized analysis for multivariate polynomial resource bounds*. We faced three main challenges in the development of the analysis.

1. The identification of multivariate polynomials that accurately describe the resource cost of typical examples. It is necessary that they are closed under natural operations to be suitable for local typing rules. Moreover, they must handle an unbounded number of arguments to tightly cope with nested data structures.
2. The automatic relation of sizes of data structures in function arguments and results, even if data that is scattered over different locations (like $n_1 + n_2 \leq n$ in the partitioning of quick sort).
3. The smooth integration of the inference of size relations and resource bounds to deal with the interactions of different functions while keeping the analysis technically feasible in practice.

To address challenge one we define *multivariate resource polynomials* that are a generalization of the resource polynomials that we used earlier [16]. To address challenges two and three we introduce a multivariate potential-based amortized analysis (§5 and §6). The local type rules emit only simple linear constraints and are remarkably modest considering the variety of relations between different parts of the data that are taken into account.

Our experiments with a prototype implementation[3] (see §7) show that our system automatically infers tight multivariate bounds for complex programs that involve nested data structures such as trees of lists. Additionally, it can deal with the same wide range of linear and univariate programs as the previous systems.

As representative examples we present in §7 the analyses of the dynamic programming algorithm for the length of the longest common subsequence of two lists and an implementation of insertion sort that lexicographically sorts a list of lists. Note that the latter example exhibits a worst-case running time of the form $O(n^2 m)$ where n is the length of the outer list and m is the maximal length of the inner lists. The reason is that each of the $O(n^2)$ comparisons performed by insertion sort needs time linear in m.

We also implemented a more involved case study on matrix operations were matrices are lists of lists of integers. It demonstrates interesting capabilities like the precise automatic tracking of data sizes when transposing matrices or the automatic analyses of complex functions like the multiplication of lists of matrices of different (fitting) dimensions. Details are available on the web.

The main contributions we make in this paper are as follows.

1. The definition of multivariate resource polynomials that generalize univariate resource polynomials [16]. (in §4)
2. The introduction of type annotation that correspond to *global polynomial potential functions* for amortized analysis which depend on the sizes of several parts of the input. (in §5)
3. The presentation of *local type rules* that modify type annotations for global potential functions. (in §6)
4. The implementation of an efficient type inference algorithm that relies on *linear constraint solving* only.

2. Background and Informal Presentation

Amortized Analysis *Amortized analysis* with the *potential method* has been introduced [26] to manually analyze the efficiency of data structures. The key idea is to incorporate a non-negative potential into the analysis that can be used to pay (costly) operations.

To apply the potential method to statically analyze a program, one has to determine a mapping from machine states to potentials for every program point. Then one has to show that for every possible evaluation, the potential at a program point suffices to cover the cost of the next transition and the potential at the succeeding program point. The initial potential is then an upper bound on the resource consumption of the program.

Linear Potential One way to achieve such an analysis is to use linear potential functions [18]. Inductive data structures are statically annotated with a positive rational numbers q to define non-negative potentials $\Phi(n) = q \cdot n$ as a function of the size n of the data. Then a sound albeit incomplete type-based analysis of the program text statically verifies that the potential is sufficient to pay for all operations that are performed on this data structure during any possible evaluation of the program.

The analysis is best explained by example. Consider the function *filter*:$(int, L(int)) \to L(int)$ that removes the multiples of a given integer from a list of integers.

```
filter(p,l) = match l with | nil -> nil
        | (x::xs) -> let xs' = filter(p,xs) in
                     if x mod p == 0 then xs' else x::xs'
```

Assume that we need two memory cells to create a new list cell. Then the heap-space usage of an evaluation of *filter(p,ℓ)* is at most $2|\ell|$. To infer an upper bound on the heap-space usage we enrich the type of *filter* with a priori unknown potential annotations[4] $q_{(0,i)}, p_i \in \mathbb{Q}_0^+$.

$$filter{:}((int, L(int)), (q_{(0,0)}, q_{(0,1)})) \to (L(int), (p_0, p_1))$$

The intuitive meaning of the resulting type is as follows: to evaluate *filter(p,ℓ)* one needs $q_{(0,1)}$ memory cells per element in the list ℓ and $q_{(0,0)}$ additional memory cells. After the evaluation there are p_0 memory cells and p_1 cells per element of the returned list left. We say that the pair *(p,ℓ)* has potential

[3] See `http://raml.tcs.ifi.lmu.de` for a web interface, example programs, and the source code.

[4] We use the naming scheme of the unknowns that arises from the more general method introduced in this paper.

$\Phi((p,\ell),(q_{(0,0)},q_{(0,1)})) = q_{(0,0)} + q_{(0,1)} \cdot |\ell|$ and that $\ell' = \mathit{filter}(p,\ell)$ has potential $\Phi(\ell',(p_0,p_1)) = p_0 + p_1 \cdot |\ell'|$. A valid potential annotation would be for instance $q_{(0,0)} = p_0 = p_1 = 0$ and $q_{(0,1)} = 2$. Another valid annotation would be $q_{(0,0)} = p_0 = 0$, $p_1 = 2$, and $q_{(0,1)} = 4$. It can be used to type the inner call of *filter* in an expression like *filter(a,filter(b,ℓ))*.

To infer the potential annotations one can use a standard type inference in which simple linear constraints are collected as each type rule is applied. For the heap-space consumption of *filter* the constraints would state that $q_{(0,0)} \geq p_0$ and $q_{(0,1)} \geq 2 + p_1$.

Univariate Polynomials An automatic amortized analysis can be also used to derive potential functions of the form $\sum_{i=0,\dots,k} q_i \binom{n}{i}$ with $q_i \geq 0$ while still relying on solving linear inequalities only [16]. These potential functions are attached to inductive data structures via type annotations of the form $\vec{q} = (q_0,\dots,q_k)$ with $q_i \in \mathbb{Q}_0^+$. For instance, the typing $\ell{:}(L(\mathit{int}),(4,3,2,1))$, defines the potential $\Phi(\ell,(4,3,2,1)) = 4 + 3|\ell| + 2\binom{|\ell|}{2} + 1\binom{|\ell|}{3}$.

The use of the binomial coefficients rather than powers of variables has several advantages. In particular, the identity $\sum_{i=0,\dots,k} q_i \binom{n+1}{i} = \sum_{i=0,\dots,k-1} q_{i+1}\binom{n}{i} + \sum_{i=0,\dots,k} q_i \binom{n}{i}$ gives rise to a local typing rule for *list match* which allows to type naturally both, recursive calls and other calls to subordinate functions in branches of a pattern match.

This identity forms the mathematical basis of the *additive shift* $\lhd$ of a type annotation which is defined by $\lhd(q_0,\dots,q_k) = (q_0+q_1,\dots,q_{k-1}+q_k,q_k)$. For example, it appears in the typing $\mathit{tail}{:}(L(\mathit{int}),\vec{q}) \to (L(\mathit{int}),\lhd(\vec{q}))$ of the function *tail* that removes the first element from a list. The potential resulting from the contraction $\mathit{xs}{:}(L(\mathit{int}),\lhd(\vec{q}))$ of a list $\mathit{(x{::}xs)}{:}(L(\mathit{int}),\vec{q})$, usually in a pattern match, suffices to pay for three common purposes: (i) to pay the constant costs q_1 after and before the recursive calls, (ii) to fund, by $(q_2,\dots,q_n)$, calls to auxiliary functions, and (iii) to pay, by $(q_0,\dots,q_n)$, for the recursive calls.

To see how the polynomial potential annotations are used, consider the function $\mathit{eratos}{:}L(\mathit{int}){\to}L(\mathit{int})$ that implements the sieve of Eratosthenes. It successively calls the function *filter* to delete multiples of the first element from the input list. If *eratos* is called with a list of the form $[2,3,\dots,n]$ then it computes the list of primes p with $2 \leq p \leq n$.

```
eratos l = match l with | nil -> nil
           | (x::xs) -> x::eratos(filter(x,xs))
```

Note that it is possible in our system to implement the function *filter* with a destructive pattern match (just replace *match* with *matchD*). That would result in a filter function that does not consume heap-cells and in a linear heap-space consumption of *eratos*. But to illustrate the use of quadratic potential we use the *filter* function with linear heap-space consumption from the first example.[5] In an evaluation of *eratos(ℓ)* the function *filter* is called once for every sublist of the input list ℓ in the worst case. Then the calls of *filter* cause a worst-case heap-space consumption of $2\binom{|\ell|}{2}$. This is for example the case if ℓ is a list of pairwise distinct primes. Additionally, there is the creation of a new list element for every recursive call of *eratos*. Thus, the total worst-case heap-space consumption of the function is $2n + 2\binom{n}{2}$ if n is the size of the input list.

To bound the heap-space consumption of *eratos*, our analysis system automatically computes the following type.

$$\mathit{eratos}{:}(L(\mathit{int}),(0,2,2)) \to (L(\mathit{int}),(0,0,0))$$

Since the typing assigns the initial potential $2n+2\binom{n}{2}$ to a function argument of size n, the analysis computes a tight heap-space bound for *eratos*. In the pattern match, the additive shift assigns the type $(L(\mathit{int}),(2,4,2))$ to the variable *xs*. The constant potential 2 is then used to pay for the cons operation (i). The non-constant potential $\mathit{xs}{:}(L(\mathit{int}),(0,4,2))$ is shared between the two occurrences of *xs* in the following expression by using $\mathit{xs}{:}(L(\mathit{int}),(0,2,0))$ to pay the cost of *filter(xs)* (ii) and by using $\mathit{xs}{:}(L(\mathit{int})(0,2,2)$ to pay for the recursive call of *eratos* (iii).

To infer the typing, we start with an unknown potential annotation as in the linear case.

$$\mathit{eratos}{:}(L(\mathit{int}),(q_0,q_1,q_2)) \to (L(\mathit{int}),(p_0,p_1,p_2))$$

The syntax-directed type analysis then computes linear inequalities which state that $q_0 \geq p_0$, $q_1 \geq 2 + p_1$, and $q_2 \geq 2 + p_2$.

This analysis method works for many functions that admit a worst-case resource consumption that can be expressed by sums of univariate polynomials like $n^2 + m^2$. However, it often fails to compute types for functions whose resource consumption is bounded by a mixed term like $n^2{\cdot}m$. The reason is that the potential is attached to a single data structure and does not take into account relations between different data structures.

Multivariate Bounds This paper extends type-based amortized analysis to compute mixed resource bounds like $2n \cdot \binom{m}{2}$. To this end, we introduce a global polynomial potential annotation that can express a variety of relations between different parts of the input. To give a flavor of the basic ideas we informally introduce this global potential in this section for pairs of integer lists.

The potential of a single integer list can be expressed as a vector $(q_0,q_1,\dots,q_k)$ that defines a potential-function of the form $\sum_{i=0}^{k} q_i\binom{n}{i}$. To represent mixed terms of degree $\leq k$ for a pair of integer lists we use a triangular matrix $Q = (q_{(i,j)})_{0\leq i+j\leq k}$. Then Q defines a potential-function of the form $\sum_{0\leq i+j\leq k} q_{(i,j)}\binom{n}{i}\binom{m}{j}$ where m and n are the lengths of the two lists.

This definition has the same advantages as the univariate version of the system. Particularly, we can still use the additive shift to assign potential to sublists. To generalize the additive shift of the univariate system, we use the identity $\sum_{0\leq i+j\leq k} q_{(i,j)}\binom{n+1}{i}\binom{m}{j} = \sum_{0\leq i+j\leq k-1} q_{(i+1,j)}\binom{n}{i}\binom{m}{j} + \sum_{0\leq i+j\leq k} q_{(i,j)}\binom{n}{i}\binom{m}{j}$. It is reflected by two additive shifts $\lhd_1(Q) = (q_{(i,j)} + q_{(i+1,j)})_{0\leq i+j\leq k}$ and $\lhd_2(Q) = (q_{(i,j)} + q_{(i,j+1)})_{0\leq i+j\leq k}$ where $q_{(i,j)} := 0$ if $i + j > k$. The shift operations can be used like in the univariate case. For example, we derive the typing $\mathit{tail1}{:}\ ((L(\mathit{int}),L(\mathit{int})),Q) \to ((L(\mathit{int}),L(\mathit{int})),\lhd_1(Q))$ for the function *tail1(xs,ys)=(tail xs,ys)*.

To see how the mixed potential is used, consider the function *dyade* that computes the dyadic product of two lists.

```
mult(x,l) = match l with | nil -> nil
                         | (y::ys) -> x*y::mult(x,ys)

dyade(l,ys) = match l with | nil -> nil
              | (x::xs) -> (mult(x,ys))::dyade(xs,ys)
```

Similar to previous examples, *mult* consumes $2n$ heap cells if n is the length of input. This exact bound is represented by the typing

$$\mathit{mult}{:}\ ((\mathit{int},L(\mathit{int})),(0,2,0)) \to (L(\mathit{int}),(0,0,0))$$

that states that the potential is $0 + 2n + 0\binom{n}{2}$ before and 0 after the evaluation of *mult(x,ℓ)* if ℓ is a list of length n.

The function *dyade* consumes $2n + 2nm$ heap cells if n is the length of first argument and m is the length of the second argument. This is why the following typing represents a tight heap-space bound for the function.

$$\mathit{dyade}{:}\ ((L(\mathit{int}),L(\mathit{int})),\begin{pmatrix} 0 & 0 & 0 \\ 2 & 2 & \\ 0 & & \end{pmatrix}) \to (L(\mathit{int},\mathit{int}),0)$$

[5] It is just more convenient to argue about heap space than to argue about evaluation steps.

To verify this typing of *dyade*, the additive shift $\lhd_1$ is used in the pattern matching. This results in the potential

$$\textit{(xs,ys)}\!: ((L(\mathit{int}), L(\mathit{int})), \begin{pmatrix} 2 & 2 & 0 \\ 2 & 2 & \\ 0 & & \end{pmatrix})$$

that is used as in the function *eratos*: the constant potential 2 is used to pay for the *cons* operation (i), the linear potential $ys{:}(L(\mathit{int}), (0, 2, 0))$ is used to pay the cost of *mult(ys)* (ii), the rest of the potential is used to pay for the recursive call (iii).

Multivariate potential is also needed to assign a super-linear potential to the result of a function like *append*. This is, for example, needed to type an expression like *eratos(append(ℓ_1,ℓ_2))*. Here, *append* would have the type

$$\textit{append:}\ ((L(\mathit{int}), L(\mathit{int})), \begin{pmatrix} 0 & 2 & 2 \\ 4 & 2 & \\ 2 & & \end{pmatrix}) \to (L(\mathit{int}), (0, 2, 2)).$$

The correctness of the bound follows from the convolution formula $\binom{n+m}{2} = \binom{n}{2} + \binom{m}{2} + nm$ and from the fact that *append* consumes $2n$ resources if n is the length of the first argument. The respective initial potential $4n + 2m + 2(\binom{n}{2} + \binom{m}{2} + mn)$ furnishes a tight bound on the worst-case heap-space consumption of the evaluation of *eratos(append(ℓ_1,ℓ_2))*, where $|\ell_1| = n$, $|\ell_2| = m$.

3. Resource Aware ML

RAML (Resource Aware ML) is a first-order functional language with ML-style syntax, booleans, integers, pairs, lists, binary trees, recursion and pattern match. In the implementation of RAML we already included a destructive pattern match that we could handle using the methods described here.

Syntax To simplify typing rules and semantics, we define the following *expressions of RAML* to be in *let normal form*. In the implementation we transform unrestricted expressions into a let normal form with explicit sharing before the type analysis.

$$\begin{aligned} e ::= {} & () \mid \mathit{True} \mid \mathit{False} \mid n \mid x \mid x_1\ \mathit{binop}\ x_2 \mid f(x_1, \ldots, x_n) \\ & \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \\ & \mid (x_1, x_2) \mid \mathit{nil} \mid \mathit{cons}(x_h, x_t) \mid \mathit{leaf} \mid \mathit{node}(x_0, x_1, x_2) \\ & \mid \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e \\ & \mid \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \\ & \mid \mathit{match}\ x\ \mathit{with}\ \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 \\ \mathit{binop} ::= {} & + \mid - \mid * \mid \mathit{mod} \mid \mathit{div} \mid \mathit{and} \mid \mathit{or} \end{aligned}$$

We skip the standard definitions of integer constants $n \in \mathbb{Z}$ and variable identifiers $x \in$ VID. For the resource analysis it is unimportant which ground operations are used in the definition of *binop*. In fact, one can use here every function that has a constant worst-case resource consumption.

Simple Types We define the well-typed expressions of RAML by assigning a *simple type*, a usual ML type without resource annotations, to well-typed expressions. Simple types are data types and first-order types as given by the grammars below.

$$A ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid L(A) \mid T(A) \mid (A, A) \qquad F ::= A \to A$$

To each simple type A we assign a set of semantic values $[\![A]\!]$ in the obvious way. For example $[\![T(\mathit{int}, \mathit{int})]\!]$ is the set of finite binary trees whose nodes are labeled with pairs of integers. It is convenient to identify tuples like (A_1, A_2, A_3, A_4) with the pair type $(A_1, (A_2, (A_3, A_4)))$.

A *typing context* Γ is a partial, finite mapping from variable identifiers to data types. A *signature* Σ is a finite, partial mapping of function identifiers to first-order types. The typing judgment $\Gamma \vdash_\Sigma e : A$ states that the expression e has type A under the signature Σ in the context Γ. The typing rules that define the typing judgment are standard and a subset of the resource-annotated typing rules from §5 if the resource annotations are omitted.

Programs Each *RAML program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions with a distinguished variable identifier such that $y_f{:}A \vdash_\Sigma e_f{:}B$ if $\Sigma(f) = A \to B$.

We write $f(y_1, \ldots, y_k) = e'_f$ as an abbreviation to indicate that $\Sigma(f) = (A_1, (A_2, (\ldots, A_k) \cdots) \to B$ and $y_1{:}A_1, \ldots, y_k{:}A_k \vdash_\Sigma e'_f{:}B$. In this case, f is defined by $e_f = \text{match } y_f \text{ with } (y_1, y'_f) \to \text{match } y'_f \text{ with } (y_2, y''_f) \ldots e'_f$. Of course, one can use such function definitions also in the implementation.

Operational Semantics To prove the correctness of our analysis, we define a big-step operational semantics that measures the quantitative resource consumption of programs. It is parametric in the resource of interest and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. The actual constants for a step on a specific system architecture can be derived by analyzing the translation of the step in the compiler implementation for that architecture [23].

The semantics is formulated with respect to a stack and a heap as usual: A value $v \in$ Val is either a location $\ell \in$ Loc, a boolean constant b, an integer n, a null value NULL or a pair of values (v_1, v_2). A *heap* is a finite partial mapping $\mathcal{H} : \text{Loc} \to \text{Val}$ from locations to values. A *stack* is a finite partial mapping $\mathcal{V} : \text{VID} \to \text{Val}$ from variables to values.

The operational evaluation rules define an evaluation judgment of the form $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ expressing the following. If the stack $\mathcal{V}$ and the initial heap $\mathcal{H}$ are given then the expression e evaluates to the value v and the new heap $\mathcal{H}'$. To evaluate e one needs at least $q \in \mathbb{Q}^+$ resource units and after the evaluation there are $q' \in \mathbb{Q}^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available during the execution of e.

Fig. 1 shows the evaluation rules of the big-step semantics. There is at most one pair (q, q') such that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ for a given expression e, a heap $\mathcal{H}$ and a stack $\mathcal{V}$. The non-negative number q is the (high) watermark of resources that are used simultaneously during the evaluation.

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid $\mathcal{Q} = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, \cdot)$. The neutral element is $(0, 0)$ which means that resources are neither used nor restituted. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q',\ p') & \text{if } q' \leq p \\ (q,\ p' + q' - p) & \text{if } q' > p \end{cases}$$

If resources are never restituted (as with time) then we can restrict to elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$.

We identify a rational number q with an element of $\mathcal{Q}$ as follows: $q \geq 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$. This notation avoids case distinctions in the evaluation rules since the constants K that appear in the rules might be negative.

A notorious dissatisfying feature of classical big-step semantics is that it does not provide evaluation judgments for non-terminating evaluations. In a companion paper [17] we describe a big-step operational semantics for partial evaluations that agrees with the usual big-step semantics on terminating computations. It inductively defines statements of the form $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid q$ for a stack $\mathcal{V}$, a heap $\mathcal{H}$, $q \in \mathbb{Q}_0^+$ and an expression e. The meaning is that there is a partial evaluation of e with the stack $\mathcal{V}$ and the heap $\mathcal{H}$ that consumes q resources. This allows for a smooth extension of the soundness theorem (Theorem 1) to non-terminating evaluations (see [17]).

$$\frac{x \in \mathrm{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash x \rightsquigarrow \mathcal{V}(x),\mathcal{H} \mid K^{\mathrm{var}}}\ (\text{E:}\textsc{Var}) \qquad \frac{}{\mathcal{V},\mathcal{H} \vdash () \rightsquigarrow \textsc{Null},\mathcal{H} \mid K^{\mathrm{unit}}}\ (\text{E:}\textsc{ConstU}) \qquad \frac{n \in \mathbb{Z}}{\mathcal{V},\mathcal{H} \vdash n \rightsquigarrow n,\mathcal{H} \mid K^{\mathrm{int}}}\ (\text{E:}\textsc{ConstI})$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\}}{\mathcal{V},\mathcal{H} \vdash b \rightsquigarrow b,\mathcal{H} \mid K^{\mathrm{bool}}}\ (\text{E:}\textsc{ConstB}) \qquad \frac{\mathcal{V}(x)=v \qquad [y_f \mapsto v],\mathcal{H} \vdash e_f \rightsquigarrow v',\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash f(x) \rightsquigarrow v',\mathcal{H}' \mid K_1^{\mathrm{app}} \cdot (q,q') \cdot K_2^{\mathrm{app}}}\ (\text{E:}\textsc{App})$$

$$\frac{x_1,x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = op(\mathcal{V}(x_1),\mathcal{V}(x_2))}{\mathcal{V},\mathcal{H} \vdash x_1\ op\ x_2 \rightsquigarrow v,\mathcal{H} \mid K^{op}}\ (\text{E:}\textsc{BinOp}) \qquad \frac{\mathcal{V}(x)=\mathit{True} \qquad \mathcal{V},\mathcal{H} \vdash e_t \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\,x\ \mathit{then}\,e_t\ \mathit{else}\,e_f \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{conT}}\cdot(q,q')\cdot K_2^{\mathrm{conT}}}\ (\text{E:}\textsc{CondT})$$

$$\frac{\mathcal{V}(x)=\mathit{False} \qquad \mathcal{V},\mathcal{H} \vdash e_f \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\,x\ \mathit{then}\,e_t\ \mathit{else}\,e_f \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{conF}}\cdot(q,q')\cdot K_2^{\mathrm{conF}}}\ (\text{E:}\textsc{CondF})$$

$$\frac{\mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v_1,\mathcal{H}_1 \mid (q,q') \qquad \mathcal{V}[x \mapsto v_1],\mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2,\mathcal{H}_2 \mid (p,p')}{\mathcal{V},\mathcal{H} \vdash \mathit{let}\,x = e_1\ \mathit{in}\,e_2 \rightsquigarrow v_2,\mathcal{H}_2 \mid K_1^{\mathrm{let}} \cdot (q,q') \cdot K_2^{\mathrm{let}} \cdot (p,p') \cdot K_3^{\mathrm{let}}}\ (\text{E:}\textsc{Let})$$

$$\frac{x_1,x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_1),\mathcal{V}(x_2))}{\mathcal{V},\mathcal{H} \vdash (x_1,x_2) \rightsquigarrow v,\mathcal{H} \mid K^{\mathrm{pair}}}\ (\text{E:}\textsc{Pair}) \qquad \frac{\mathcal{V}(x)=(v_1,v_2) \qquad \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2],\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\,x\ \mathit{with}\,(x_1,x_2) \to e \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{matP}} \cdot (q,q') \cdot K_2^{\mathrm{matP}}}\ (\text{E:}\textsc{MatP})$$

$$\frac{}{\mathcal{V},\mathcal{H} \vdash \mathit{nil} \rightsquigarrow \textsc{Null},\mathcal{H} \mid K^{\mathrm{nil}}}\ (\text{E:}\textsc{Nil}) \qquad \frac{x_h,x_t \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_h),\mathcal{V}(x_t)) \qquad l \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V},\mathcal{H} \vdash \mathit{cons}(x_h,x_t) \rightsquigarrow l,\mathcal{H}[l \mapsto v] \mid K^{\mathrm{cons}}}\ (\text{E:}\textsc{Cons})$$

$$\frac{\mathcal{V}(x)=\textsc{Null} \qquad \mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\,x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{matN}} \cdot (q,q') \cdot K_2^{\mathrm{matN}}}\ (\text{E:}\textsc{MatNil})$$

$$\frac{\mathcal{V}(x)=l \qquad \mathcal{H}(l)=(v_h,v_t) \qquad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t],\mathcal{H} \vdash e_2 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\,x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{matC}} \cdot (q,q') \cdot K_2^{\mathrm{matC}}}\ (\text{E:}\textsc{MatCons})$$

$$\frac{}{\mathcal{V},\mathcal{H} \vdash \mathit{leaf} \rightsquigarrow \textsc{Null},\mathcal{H} \mid K^{\mathrm{leaf}}}\ (\text{E:}\textsc{Leaf}) \qquad \frac{x_0,x_1,x_2 \in \mathrm{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_0),\mathcal{V}(x_1),\mathcal{V}(x_2)) \qquad l \notin \mathrm{dom}(\mathcal{H})}{\mathcal{V},\mathcal{H} \vdash \mathit{node}(x_0,x_1,x_2) \rightsquigarrow l,\mathcal{H}[l \mapsto v] \mid K^{\mathrm{node}}}\ (\text{E:}\textsc{Node})$$

$$\frac{\mathcal{V}(x)=\textsc{Null} \qquad \mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\,x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0,x_1,x_2) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{matTL}} \cdot (q,q') \cdot K_2^{\mathrm{matTL}}}\ (\text{E:}\textsc{MatLeaf})$$

$$\frac{\mathcal{V}(x)=l \qquad \mathcal{H}(l)=(v_0,v_1,v_2) \qquad \mathcal{V}[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2],\mathcal{H} \vdash e_2 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\,x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0,x_1,x_2) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\mathrm{matTN}} \cdot (q,q') \cdot K_2^{\mathrm{matTN}}}\ (\text{E:}\textsc{MatNode})$$

Figure 1. Evaluation rules of the big-step operational semantics.

The Cost-Free Resource Metric The type rules in §6 make use of the *cost-free* resource metric. This is the metric in which all constants K that appear in the rules are instantiated to zero. It follows that if $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (q,q')$ then $q = q' = 0$. We will use the cost-free metric in §6 to pass on potential in the typing rule for let expressions.

Well-Formed Environments If $\mathcal{H}$ is a heap, v is a value, A is a type, and $a \in [\![A]\!]$ then we write $\mathcal{H} \vDash v{\mapsto}a : A$ to mean that v defines the semantic value $a \in [\![A]\!]$ when pointers are followed in $\mathcal{H}$ in the obvious way. We elide a formal definition of this judgment.

Note that if $\mathcal{H} \vDash v{\mapsto}a : A$ then v may well point to a data structure with some aliasing, but no circularity is allowed since this would require infinitary values a. We do not include them because in our functional language there is no way of generating such values; in principle our method can encompass circular data [19].

We also write $\mathcal{H} \vDash v : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $\mathcal{H} \vDash v{\mapsto}a : A$. A stack $\mathcal{V}$ and a heap $\mathcal{H}$ are *well-formed* with respect to a context Γ if $\mathcal{H} \vDash \mathcal{V}(x) : \Gamma(x)$ holds for every $x \in \mathrm{dom}(\Gamma)$. We then write $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$. Formal definitions can be found in the literature [24].

4. Resource Polynomials

A resource polynomial maps a value of some data type to a nonnegative rational number. Potential functions are always given by such resource polynomials.

In the case of an inductive tree-like data type, a resource polynomial will only depend on the list of entries of the data structure in pre-order. Thus, if $D(A)$ is such a data type with entries of type A, e.g., A-labelled binary trees, and v is a value of type $D(A)$ then we write $\mathrm{elems}(v) = [a_1,\ldots,a_n]$ for this list of entries.

An analysis of typical polynomial computations operating on a data structure v with $\mathrm{elems}(v) = [a_1,\ldots,a_n]$ shows that it consists of operations that are executed for every k-tuple $(a_{i_1},\ldots,a_{i_k})$ with $1 \le i_1 < \cdots < i_k \le n$. The simplest examples are linear map operations that perform some operation for every a_i. Another example are common sorting algorithms that perform comparisons for every pair (a_i,a_j) with $1 \le i < j \le n$ in the worst case.

Base Polynomials For each data type A we now define a set $\mathcal{P}(A)$ of functions $p : [\![A]\!] \to \mathbb{N}$ that map values of type A to natural numbers. The resource polynomials for type A are then given as nonnegative rational linear combinations of these *base polynomials*. We define $\mathcal{P}(A)$ as follows.

$$\mathcal{P}(A) = \{a \mapsto 1\} \text{ if } A \text{ is an atomic type}$$
$$\mathcal{P}(A_1, A_2) = \{(a_1, a_2) \mapsto p_1(a_1) \cdot p_2(a_2) \mid p_i \in \mathcal{P}(A_i)\}$$
$$\mathcal{P}(D(A)) = \{v \mapsto \sum_{1 \le j_1 < \cdots < j_k \le n} \prod_{i=1}^{k} p_i(a_{j_i}) \mid k \in \mathbb{N}, p_i \in \mathcal{P}(A)\}$$

In the last clause $[a_1, \ldots, a_n] = \text{elems}(v)$. Every set $\mathcal{P}(A)$ contains the constant function $v \mapsto 1$. In the case of $D(A)$ this arises for $k = 0$ (one element sum, empty product).

For example, the function $\ell \mapsto \binom{|\ell|}{k}$ is in $\mathcal{P}(L(A))$ for every $k \in \mathbb{N}$; simply take $p_1 = \ldots = p_k = 1$ in the definition of $\mathcal{P}(D(A))$. The function $(\ell_1, \ell_2) \mapsto \binom{|\ell_1|}{k_1} \cdot \binom{|\ell_2|}{k_2}$ is in $\mathcal{P}(L(A), L(B))$ for every $k_1, k_2 \in \mathbb{N}$ and $[\ell_1, \ldots, \ell_n] \mapsto \sum_{1 \le i < j \le n} \binom{|\ell_i|}{k_1} \cdot \binom{|\ell_j|}{k_2} \in \mathcal{P}(L(L(A)))$ for every $k_1, k_2 \in \mathbb{N}$.

Resource Polynomials A *resource polynomial* $p : [\![A]\!] \to \mathbb{Q}_0^+$ for a data type A is a non-negative linear combination of base polynomials, i.e.,

$$p = \sum_{i=1,\ldots,m} q_i \cdot p_i$$

for $q_i \in \mathbb{Q}_0^+$ and $p_i \in \mathcal{P}(A)$. We write $\mathcal{R}(A)$ for the set of resource polynomials for A.

An instructive, but not exhaustive, example is given by $\mathcal{R}_n = \mathcal{R}(L(\mathit{int}), \ldots, L(\mathit{int}))$. The set $\mathcal{R}_n$ is the set of linear combinations of products of binomial coefficients over variables $x_1, \ldots, x_n$, that is, $\mathcal{R}_n = \{\sum_{i=1}^{m} q_i \prod_{j=1}^{n} \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$. These expressions naturally generalize the polynomials used in our univariate analysis [16] and meet two conditions that are important to efficiently manipulate polynomials during the analysis. First, the polynomials are non-negative, and secondly, they are closed under the discrete difference operators Δ_i for every i. The discrete derivative $\Delta_i\, p$ is defined through $\Delta_i\, p(x_1, \ldots, x_n) = p(x_1, \ldots, x_i + 1, \ldots, x_n) - p(x_1, \ldots, x_n)$.

As in [16] it can be shown that $\mathcal{R}_n$ is the largest set of polynomials enjoying these closure properties. It would be interesting to have a similar characterisation of $\mathcal{R}(A)$ for arbitrary A. So far, we know that $\mathcal{R}(A)$ is closed under sum and product (see Lemma 1) and are compatible with the construction of elements of data structures in a very natural way (see Lemmas 2 and 3). This provides some justification for their choice and canonicity. An abstract characterization would have to take into account the fact that our resource polynomials depend on an unbounded number of variables, e.g., sizes of inner data structures, and are not invariant under permutation of these variables. It seems that some generalization of infinite symmetric polynomials to subgroups of the symmetric group could be useful, but this would not serve our immediate goal of accurate multivariate resource analysis.

5. Annotated Types

The resource polynomials described in §4 are non-negative linear combinations of base polynomials. The rational coefficients of the linear combination are present as type annotations in our type system. To relate type annotations to resource polynomials we systematically describe base polynomials and resource polynomials for data of a given type.

If one considers only univariate polynomials then their description is straightforward. Every inductive data of size n admits a potential of the form $\sum_{1 \le i \le k} q_i \binom{n}{i}$. So we can describe the potential function with a vector $\vec{q} = (q_1, \ldots, q_k)$ in the corresponding recursive type. For instance can we write $L^{\vec{q}}(A)$ for annotated list types. Since each annotation refers to the size of one input part only, univariatly annotated types can be directly composed. For example, an annotated type for a pair of lists has the form $(L^{\vec{q}}(A), L^{\vec{p}}(A))$. See [16] for details.

Here, we work with multivariate potential functions, i.e., functions that depend on the sizes of different parts of the input. For a pair of lists of lengths n and m we have, for instance, a potential function of the form $\sum_{0 \le i+j \le k} q_{ij} \binom{n}{i} \binom{m}{j}$ which can be described by the coefficients q_{ij}. But we also want to describe potential functions that refer to the sizes of different lists inside a list of lists, etc. That is why we need to describe a set of indexes $I(A)$ that enumerate the basic resource polynomials p_i and the corresponding coefficients q_i for a data type A. These type annotations can be, in a straight forward way, automatically transformed into usual easily understood polynomials. This is done in our prototype to present the bounds to the user at the end of the analysis.

Names For Base Polynomials To assign a unique name to each base polynomial we define the *index set* $I(A)$ to denote resource polynomials for a given data type A. Interestingly, but as we find coincidentally, $I(A)$ is essentially the meaning of A with every atomic type replaced by *unit*.

$$I(A) = \{*\} \text{ if } A \in \{\mathit{int}, \mathit{bool}, \mathit{unit}\}$$
$$I(A_1, A_2) = \{(i_1, i_2) \mid i_1 \in I(A_1) \text{ and } i_2 \in I(A_2)\}$$
$$I(L(B)) = I(T(B)) = \{[i_1, \ldots, i_k] \mid k \ge 0, i_j \in I(B)\}$$

The *degree* $\deg(i)$ of an index $i \in I(A)$ is defined as follows.

$$\deg(*) = 0$$
$$\deg(i_1, i_2) = \deg(i_1) + \deg(i_2)$$
$$\deg([i_1, \ldots, i_k]) = k + \deg(i_1) + \cdots + \deg(i_k)$$

Define $I_k(A) = \{i \in I(A) \mid \deg(i) \le k\}$. The indexes $i \in I_k(A)$ are an enumeration of the base polyonomials $p_i \in \mathcal{P}(A)$ of degree at most k. For each $i \in I(A)$, we define a base polynomial $p_i \in \mathcal{P}(A)$ as follows: If $A \in \{\mathit{int}, \mathit{bool}, \mathit{unit}\}$ then

$$p_*(v) = 1.$$

If $A = (A_1, A_2)$ is a pair type and $v = (v_1, v_2)$ then

$$p_{(i_1,i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2)$$

If $A = D(B)$ (in our type system D is either lists or binary node-labelled trees) is a data structure and $\text{elems}(v) = [v_1, \ldots, v_n]$ then

$$p_{[i_1,\ldots,i_m]}(v) = \sum_{1 \le j_1 < \cdots < j_m \le n} p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m})$$

We use the notation 0_A (or just 0) for the index in $I(A)$ such that $p_{0_A}(a) = 1$ for all a. We have $0_{\mathit{int}} = *$ and $0_{(A_1,A_2)} = (0_{A_1}, 0_{A_2})$ and $0_{D(B)} = []$. If $A = D(B)$ for B a data type then the index $[0, \ldots, 0] \in I(A)$ of length n is denoted by just n. We identify the index (i_1, i_2, i_3, i_4) with the index $(i_1, (i_2, (i_3, i_4)))$.

For a list $i = [i_1, \ldots, i_k]$ we write $i_0{::}i$ to denote the list $[i_0, i_1, \ldots, i_k]$. Furthermore, we write ii' for the concatenation of two lists i and i'.

Lemma 1 *If* $p, p' \in \mathcal{R}(A)$ *then* $p+p', p \cdot p' \in \mathcal{R}(A)$, *and* $\deg(p+p') = \max\{\deg(p), \deg(p')\}$ *and* $\deg(p \cdot p') = \deg(p) + \deg(p')$.

By linearity it suffices to show this lemma for base polynomials. This is done by induction on A.

Corollary 1 *For every* $p \in \mathcal{R}(A, A)$ *there exists* $p' \in \mathcal{R}(A)$ *with* $\deg(p') = \deg(p)$ *and* $p'(a) = p(a, a)$ *for all* $a \in [\![A]\!]$.

This follows directly from Lemma 1 noticing that base polynomials $p \in \mathcal{P}(A, A)$ take the form $p_i \cdot p_{i'}$.

Lemma 2 *Let $a \in [\![A]\!]$ and $\ell \in [\![L(A)]\!]$. Let $i_0, \ldots, i_k \in I(A)$ and $k \geq 0$. Then $p_{[i_0,i_1,\ldots,i_k]}([]) = 0$ and $p_{[i_0,i_1,\ldots,i_k]}(a{::}\ell) = p_{i_0}(a) \cdot p_{[i_1,\ldots,i_k]}(\ell) + p_0(a) \cdot p_{[i_0,i_1,\ldots,i_k]}(\ell)$.*

To prove this, one decomposes the sum in the definition of $p_{[i_0,i_1,\ldots,i_k]}(a{::}\ell)$ into two summands, one corresponding to the case where the first position j_1 equals one, thus hits a and where it is greater than one, thus a is not considered. Note that $p_0(a) = 1$; this factor is there to achieve the format of the resource polynomials for types like $(A, L(A))$.

Lemma 3 characterizes concatenations of lists (written as juxtaposition) as they will occur in the construction of tree-like data. Note that, e.g., $\text{elems}(\text{node}(a, t_1, t_2)) = a{::}\text{elems}(t_1)\, \text{elems}(t_2)$.

Lemma 3 *Let $\ell_1, \ell_2 \in [\![L(A)]\!]$. Then $\ell_1\ell_2 \in [\![L(A)]\!]$ and $p_{[i_1,\ldots,i_k]}(\ell_1\ell_2) = \sum_{t=0}^{k} p_{[i_1,\ldots,i_t]}(\ell_1) \cdot p_{[i_{t+1},\ldots,i_k]}(\ell_2)$.*

This can be proved by induction on the length of ℓ_1 using Lemma 2 or else by a decomposition of the defining sum according to which indices hit the first list and which ones hit the second.

Annotated Types and Potential Functions We use the indexes and base polynomials to define type annotations and resource polynomials. We then give examples to illustrate the definitions.

A *type annotation* for a data type A is defined to be a family

$$Q_A = (q_i)_{i \in I(A)} \text{ with } q_i \in \mathbb{Q}_0^+$$

We say Q_A is of *degree (at most) k* if $q_i = 0$ for every $i \in I(A)$ with $deg(i) > k$. An *annotated data type* is a pair (A, Q_A) of a data type A and a type annotation Q_A of some degree k.

Let $\mathcal{H}$ be a heap and let v be a value with $\mathcal{H} \vDash v{\mapsto}a : A$ for a data type A. Then the type annotation Q_A defines the *potential*

$$\Phi_{\mathcal{H}}(v{:}(A, Q_A)) = \sum_{i \in I(A)} q_i \cdot p_i(a)$$

Usually we define type annotations Q_A by only stating the values of the non-zero coefficients q_i. However, it is sometimes handy to write annotations $(q_0, \ldots, q_n)$ for a list of atomic types just as a vector. Similarly, we write annotations $(q_0, q_{(1,0)}, q_{(0,1)}, q_{(1,1)}, \ldots)$ for pairs of lists of atomic types sometimes as a triangular matrix.

If $a \in [\![A]\!]$ and Q is a type annotation for A then we also write $\Phi(a : (A, Q))$ for $\sum_i q_i p_i(a)$.

Examples The simplest annotated types are those for atomic data types like integers. The indexes for *int* are $I(int) = \{*\}$ and thus each type annotation has the form (int, q_0) for a $q_0 \in \mathbb{Q}_0^+$. It defines the constant potential function $\Phi_{\mathcal{H}}(v{:}(int, q_0)) = q_0$. Similarly, tuples of atomic types feature a single index of the form $(*, \ldots, *)$ and a constant potential function defined by some $q_{(*,\ldots,*)} \in \mathbb{Q}_0^+$.

More interesting examples are lists of atomic types like, e.g., $L(int)$. The set of indexes of degree k is then $I_k(L(int)) = \{[], [*], [*, *], \ldots, [*, \ldots, *]\}$ where the last list contains k unit elements. Since we identify a list of i unit elements with the integer i we have $I_k(L(int)) = \{0, 1, \ldots, k\}$. Consequently, annotated types have the form $(L(int), (q_0, \ldots, q_k))$ for $q_i \in \mathbb{Q}_0^+$. The defined potential function is $\Phi([a_1, \ldots, a_n]{:}(L(int), (q_0, \ldots, q_n)) = \sum_{0 \leq i \leq k} q_i \binom{n}{i}$.

The next example is the type $(L(int), L(int))$ of pairs of integer lists. The set of indexes of degree k is $I_k(L(int), L(int)) = \{(i, j) \mid i + j \leq k\}$ if we identify lists of units with their lengths as usual. Annotated types are then of the from $((L(int), L(int)), Q)$ for a triangular $k \times k$ matrix Q with non-negative rational entries. If $\ell_1 = [a_1, \ldots, a_n]$, $\ell_2 = [b_1, \ldots, b_m]$ are two lists then the potential function is $\Phi((\ell_1, \ell_2), ((L(int), L(int)), (q_{(i,j)}))) = \sum_{0 \leq i+j \leq k} q_{(i,j)} \binom{n}{i}\binom{m}{j}$.

Finally, consider the type $A = L(L(int))$ of lists of lists of integers. The set of indexes of degree k is then $I_k(L(L(int))) = \{[i_1, \ldots, i_m] \mid m \leq k, i_j \in \mathbb{N}, \sum_{j=1,\ldots,m} i_j \leq k - m\} = \{0, \ldots, k\} \cup \{[1], \ldots, [k-1]\} \cup \{[0, 1], \ldots\} \cup \cdots$. Let $\ell = [[a_{11}, \ldots, a_{1m_1}], \ldots, [a_{n1}, \ldots, a_{nm_n}]]$ be a list of lists and $Q = (q_i)_{i \in I_k(L(L(int)))}$ be a corresponding type annotation. The defined potential function is then $\Phi(\ell, (L(L(int)), Q)) =$

$$\sum_{[i_1,\ldots,i_l] \in I_k(A)} \sum_{1 \leq j_1 < \cdots < j_l \leq n} q_{[i_1,\ldots,i_l]} \binom{m_{j_1}}{i_1} \cdots \binom{m_{j_l}}{i_l}$$

In practice the potential functions are usually not very complex since most of the q_i are zero. Note that the resource polynomials for binary trees are identical to those for lists.

The Potential of a Context For use in the type system we need to extend the definition of resource polynomials to typing contexts. We treat a context like a tuple type.

Let $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ be a typing context and let $k \in \mathbb{N}$. The index set $I_k(\Gamma)$ is defined through

$$I_k(\Gamma) = \{(i_1, \ldots, i_n) \mid i_j \in I_{m_j}(A_j), \sum_{j=1,\ldots,n} m_j \leq k\}.$$

A *type annotation* Q of degree k for Γ is a family

$$Q = (q_i)_{i \in I_k(\Gamma)} \text{ with } q_i \in \mathbb{Q}_0^+.$$

We denote a *resource-annotated context* with $\Gamma; Q$. Let $\mathcal{H}$ be a heap and $\mathcal{V}$ be a stack with $\mathcal{H} \vDash \mathcal{V} : \Gamma$ where $\mathcal{H} \vDash \mathcal{V}(x_j){\mapsto}a_{x_j} : \Gamma(x_j)$. The potential of $\Gamma; Q$ with respect to $\mathcal{H}$ and $\mathcal{V}$ is

$$\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) = \sum_{(i_1,\ldots,i_n) \in I_k(\Gamma)} q_{\vec{\imath}} \prod_{j=1}^{n} p_{i_j}(a_{x_j})$$

In particular, if $\Gamma = \emptyset$ then $I_k(\Gamma) = \{()\}$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma; q_{()}) = q_{()}$. We sometimes also write q_0 for $q_{()}$.

6. Type Rules

If $f : [\![A]\!] \to [\![B]\!]$ is a function computed by some program and $K(a)$ is the cost of the evaluation of $f(a)$ then our type system will essentially try to identify resource polynomials $p \in \mathcal{R}(A)$ and $\bar{p} \in \mathcal{R}(B)$ such that $p(a) \geq \bar{p}(f(a)) + K(a)$. The key aspect of such amortized cost accounting is that it interacts well with composition.

Proposition 1 *Let $p \in \mathcal{R}(A)$, $\check{p} \in \mathcal{R}(B)$, $\bar{p} \in \mathcal{R}(C)$, $f : [\![A]\!] \to [\![B]\!]$, $g : [\![B]\!] \to [\![C]\!]$, $K_1 : [\![A]\!] \to \mathbb{Q}$, and $K_2 : [\![B]\!] \to \mathbb{Q}$. If $p(a) \geq \check{p}(f(a)) + K_1(a)$ and $\check{p}(b) \geq \bar{p}(g(b)) + K_2(b)$ for all a, b, c then $p(a) \geq \bar{p}(g(f(a))) + K_1(a) + K_2(f(a))$ for all a.*

Notice that if we merely had $p(a) \geq K_1(a)$ and $\check{p}(b) \geq K_2(b)$ then no bound could be directly obtained for the composition.

Interaction with parallel composition, i.e., $(a, c) \mapsto (f(a), c)$, is more complex due to the presence of mixed multiplicative terms in the resource polynomials.

Proposition 2 *Let $p \in \mathcal{R}(A, C)$, $\bar{p} \in \mathcal{R}(B, C)$, $f : [\![A]\!] \to [\![B]\!]$, and $K : [\![A]\!] \to \mathbb{Q}$. For each $j \in I(C)$ let $p^{(j)} \in \mathcal{R}(A)$ and $\bar{p}^{(j)} \in \mathcal{R}(B)$ be such that $p(a, c) = \sum_j p^{(j)}(a) p_j(c)$ and $\bar{p}(b, c) = \sum_j \bar{p}^{(j)}(b) p_j(c)$.*

If $p^{(0)}(a) \geq \bar{p}^{(0)}(f(a)) + K(a)$ and $p^{(j)}(a) \geq \bar{p}^{(j)}(f(a))$ holds for all a and $j \neq 0$ then $p(a, c) \geq \bar{p}(f(a), c) + K(a)$.

In fact, the situation is more complicated due to our accounting for high watermarks as opposed to merely additive cost, and also due to the fact that functions are recursively defined and may be partial. Furthermore, we have to deal with contexts and not merely types. To gain an intuition for the development to come, the above simplified view should, however, prove helpful.

Type Judgments The declarative type rules for RAML expressions (see Fig. 2) define a *resource-annotated typing judgment* of the form $\Sigma; \Gamma; Q \vdash e : (A, Q')$ where e is a RAML expression, Σ is a resource-annotated signature (see below), $\Gamma; Q$ is a resource-annotated context and (A, Q') is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $\Phi(\Gamma; Q)$ resource units available then this is sufficient to evaluate e. In addition, there are more than $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

Programs with Annotated Types *Resource-annotated first-order types* have the form $(A, Q) \to (B, Q')$ for annotated data types (A, Q) and (B, Q'). A *resource-annotated signature* Σ is a finite, partial mapping of function identifiers to *sets of* resource-annotated first-order types.

A RAML program with resource-annotated types consists of a resource-annotated signature Σ and a family of expressions with variables identifiers $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ such that $\Sigma; y_f{:}A; Q \vdash e_f : (B, Q')$ for every function type $(A, Q) \to (B, Q') \in \Sigma(f)$.

Notations Families that describe type and context annotations are denoted with upper case letters $Q, P, R, \ldots$ with optional superscripts. We use the convention that the elements of the families are the corresponding lower case letters with corresponding superscripts, i.e., $Q = (q_i)_{i \in I}$, $Q' = (q'_i)_{i \in I}$, and $Q^x = (q^x_i)_{i \in I}$.

Let Q, Q' be two annotations with the same index set I. We write $Q \le Q'$ if $q_i \le q'_i$ for every $i \in I$. For $K \in \mathbb{Q}$ we write $Q = Q' + K$ to state that $q_{\vec{0}} = q'_{\vec{0}} + K \ge 0$ and $q_i = q'_i$ for $i \ne \vec{0} \in I$. Let $\Gamma = \Gamma_1, \Gamma_2$ be a context, let $i = (i_1, \ldots, i_k) \in I(\Gamma_1)$ and $j = (j_1, \ldots, j_l) \in I(\Gamma_2)$. We write (i, j) to denote the index $(i_1, \ldots, i_k, j_1, \ldots, j_l) \in I(\Gamma)$.

We write $\Sigma; \Gamma; Q \overset{\mathrm{cf}}{\vdash} e : (A, Q')$ to refer to cost-free type judgments where all constants K in the rules from Fig. 2 are zero. We use it to assign potential to an extended context in the let rule. More explanations will follow later.

Let Q be an annotation for a context Γ_1, Γ_2. For $j \in I(\Gamma_2)$ we define the *projection* $\pi^{\Gamma_1}_j(Q)$ of Q to Γ_1 to be the annotation Q' with $q'_i = q_{(i,j)}$. The essential properties of the projections are stated by Propositions 2 and 3; they show how the analysis of juxtaposed functions can be broken down to individual components.

Proposition 3 *Let $\Gamma, x{:}A; Q$ be an annot. context, $\mathcal{H} \vDash \mathcal{V} : \Gamma, x{:}A$, and $\mathcal{H} \vDash \mathcal{V}(x) \mapsto a : A$. Then it is true that $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, x{:}A; Q) = \sum_{j \in I(A)} \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; \pi^{\Gamma}_j(Q)) \cdot p_j(a)$.*

Additive Shift A key notion in the type system is the *additive shift* that is used to assign potential to typing contexts that result from a pattern match or from the application of a constructor of an inductive data type. We first define the additive shift, then illustrate the definition with examples and finally state the soundness of the operation.

Let $\Gamma, y{:}L(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, y:L(A))}$ be a context annotation of degree k. The *additive shift for lists* $\lhd_L(Q)$ of Q is an annotation $\lhd_L(Q) = (q'_i)_{i \in I(\Gamma, x:A, xs:L(A))}$ of degree k for a context $\Gamma, x{:}A, xs{:}L(A)$ that is defined through

$$q'_{(i,j,\ell)} = \begin{cases} q_{(i,j::\ell)} + q_{(i,\ell)} & j = 0 \\ q_{(i,j::\ell)} & j \ne 0 \end{cases}$$

Let $\Gamma, t{:}T(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, t:T(A))}$ be a context annotation of degree k. The *additive shift for binary trees* $\lhd_T(Q)$ of Q is an annotation $\lhd_T(Q) = (q'_i)_{i \in I(\Gamma')}$ of degree k for a context $\Gamma' = \Gamma, x{:}A, xs_1{:}T(A), xs_2{:}T(A)$ that is defined by

$$q'_{(i,j,\ell_1,\ell_2)} = \begin{cases} q_{(i,j::\ell_1\ell_2)} + q_{(i,\ell_1\ell_2)} & j = 0 \\ q_{(i,j::\ell_1\ell_2)} & j \ne 0 \end{cases}$$

The definition of the additive shift is short but substantial. We begin by illustrating its effect in some example cases. To start with, consider a context $\ell{:}L(int)$ with a single integer list that features an annotation $(q_0, \ldots, q_k) = (q_{[]}, \ldots, q_{[0,\ldots,0]})$. The shift operation $\lhd_L$ for lists produces an annotation for a context of the form $x{:}int, xs{:}L(int)$, namely $\lhd_L(q_0, \ldots, q_k) = (q_{(0,0)}, \ldots, q_{(0,k)})$ such that $q_{(0,i)} = q_i + q_{i+1}$ for all $i < k$ and $q_{(0,k)} = q_k$. This is exactly the additive shift that we introduced in our previous work for the univariate system [16]. We use it in a context where ℓ points to a list of length $n+1$ and xs is the tail of ℓ. It reflects the fact that $\sum_{i=0,\ldots,k} q_i \binom{n+1}{i} = \sum_{i=0,\ldots,k-1} q_{i+1} \binom{n}{i} + \sum_{i=0,\ldots,k} q_i \binom{n}{i}$.

Now consider the annotated context $t{:}T(int); (q_0, \ldots, q_k)$ with a single variable t that points to a tree with $n+1$ nodes. The additive shift $\lhd_T$ produces an annotation for a context of the form $x{:}int, t_1{:}T(int), t_2{:}T(int)$. We have $\lhd_T(q_0, \ldots, q_k) = (q_{(0,i,j)})_{i+j \le k}$ where $q_{(0,i,j)} = q_{i+j} + q_{i+j+1}$ if $i + j < k$ and $q_{(0,i,j)} = q_{i+j}$ if $i + j = k$. The intention is that t_1 and t_2 are the subtrees of t which have n_1 and n_2 nodes, respectively ($n_1 + n_2 = n$). The definition of the additive shift for trees incorporates the convolution $\binom{n+m}{k} = \sum_{i+j=k} \binom{n}{i}\binom{m}{j}$ for binomials. It is true that $\sum_{i=0,\ldots,k} q_i \binom{n+1}{i} = \sum_{i=0,\ldots,k-1} (q_i + q_{i+1}) \binom{n}{i} + q_k \binom{n}{k} = \sum_{i=0}^{k-1} \sum_{j_1+j_2=i} (q_i + q_{i+1}) \binom{n_1}{j_1}\binom{n_2}{j_2} + \sum_{j_1+j_2=k} q_i \binom{n_1}{j_1}\binom{n_2}{j_2}$.

As a last example consider the context $l_1{:}L(int), l_2{:}L(int); Q$ where $Q = (q_{(i,j)})_{i+j \le k}$, l_1 is a list of length m, and l_2 is a list of length $n+1$. The additive shift results in an annotation for a context of the form $l_1{:}L(int), x{:}int, xs{:}L(int)$ and the intention is that xs is the tail of l_2, i.e., a list of length n. From the definition it follows that $\lhd_L(Q) = (q_{(i,0,j)})_{i+j \le k}$ where $q_{(i,0,j)} = q_{(i,j)} + q_{(i,j+1)}$ if $i + j < k$ and $q_{(i,0,j)} = q_{(i,j)}$ if $i + j = k$. The soundness follows from the fact that for every $i \le k$ it is true that $\sum_{j=1}^{k-i} q_{(i,j)} \binom{m}{i}\binom{n+1}{j} = \binom{m}{i} \big(\sum_{j=0}^{k-i-1} (q_{(i,j)} + q_{(i,j+1)}) \binom{n}{i} + q_{(i,k-i)} \binom{n}{k} \big)$.

Lemmas 4 and 5 state the soundness of the shift operations.

Lemma 4 *Let $\Gamma, \ell{:}L(A); Q$ be an annotated context, $\mathcal{H} \vDash \mathcal{V} : \Gamma, \ell{:}L(A)$, $\mathcal{H}(\ell) = (v_1, \ell')$ and let $\mathcal{V}' = \mathcal{V}[x_h \mapsto v_1, x_t \mapsto \ell']$. Then $\mathcal{H} \vDash \mathcal{V}' : \Gamma, x_h{:}A, x_t{:}L(A)$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, \ell{:}L(A); Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma, x_h{:}A, x_t{:}L(A); \lhd_L(Q))$.*

This is a consequence of Lemma 2. One takes the linear combination of instances of its second equation and regroups the right hand side according to the base polynomials for the resulting context.

Lemma 5 *Let $\Gamma, t{:}T(A); Q$ be an annotated context, $\mathcal{H} \vDash \mathcal{V} : \Gamma, t{:}T(A)$, $\mathcal{H}(t) = (v_1, t_1, t_2)$, and $\mathcal{V}' = \mathcal{V}[x_0 \mapsto v_1, x_1 \mapsto t_1, x_2 \mapsto t_2]$. If $\Gamma' = \Gamma, x{:}A, x_1{:}T(A), x_2{:}T(A)$ then $\mathcal{H} \vDash \mathcal{V}' : \Gamma'$ and $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, t{:}T(A); Q) = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma'; \lhd_T(Q))$.*

We remember that the potential of a tree only depends on the list of nodes in pre-order. So, we can think of the context splitting as done in two steps. First the head is separated, as in Lemma 4, and then the list of remaining elements into two lists. Lemma 5 is then proved like the previous one by regrouping terms using Lemma 2 for the first separation and Lemma 3 for the second one.

Sharing Let $\Gamma, x_1{:}A, x_2{:}A; Q$ be an annotated context. The *sharing operation* $\curlyvee(Q)$ defines an annotation for a context of the form $\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable. The following lemma shows that sharing is a linear operation that does not lead to any loss of potential.

Lemma 6 *Let A be a data type. Then there are non-negative rational numbers $c^{(i,j)}_k$ for $i, j, k \in I(A)$ and $\deg(k) \le \deg(i, j)$ such that the following holds. For every context $\Gamma, x_1{:}A, x_2{:}A; Q$ and every $\mathcal{H}, \mathcal{V}$ with $\mathcal{H} \vDash \mathcal{V} : \Gamma, x{:}A$ it holds that $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma, x{:}A; Q') = \Phi_{\mathcal{V}',\mathcal{H}}(\Gamma, x_1{:}A, x_2{:}A; Q)$ where $\mathcal{V}' = \mathcal{V}[x_1, x_2 \mapsto \mathcal{V}(x)]$ and $q'_{(\ell,k)} = \sum_{i,j \in I(A)} c^{(i,j)}_k q_{(\ell,i,j)}$.*

$$\frac{Q = Q' + K^{\text{var}}}{x{:}A; Q \vdash x : (A, Q')}\ (\text{T:Var}) \qquad \frac{q_0 = K^{\text{unit}} \quad q'_0 = 0}{\emptyset; Q \vdash () : (\mathit{unit}, Q')}\ (\text{T:ConstU}) \qquad \frac{n \in \mathbb{Z} \quad q_0 = K^{\text{int}} \quad q'_0 = 0}{\emptyset; Q \vdash n : (\mathit{int}, Q')}\ (\text{T:ConstI})$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\} \quad q_0 = K^{\text{bool}} \quad q'_0 = 0}{\emptyset; Q \vdash b : (\mathit{bool}, Q')}\ (\text{T:ConstB}) \qquad \frac{op \in \{+, -, *, \mathit{mod}, \mathit{div}\} \quad q_{(0,0)} = K^{op} \quad q'_0 = 0}{x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q \vdash x_1\ op\ x_2 : (\mathit{int}, Q')}\ (\text{T:OpInt})$$

$$\frac{P + K_1^{\text{app}} = Q \quad P' = Q' + K_2^{\text{app}} \quad (A, P) \to (A', P') \in \Sigma(f)}{x{:}A; Q \vdash f(x) : (A', Q')}\ (\text{T:App}) \qquad \frac{op \in \{\mathit{or}, \mathit{and}\} \quad q_{(0,0)} = K^{op} \quad q'_0 = 0}{x_1{:}\mathit{bool}, x_2{:}\mathit{bool}; Q \vdash x_1\ op\ x_2 : (\mathit{bool}, Q')}\ (\text{T:OpBool})$$

$$\frac{\begin{array}{c}\Gamma_1; P \vdash e_1 : (A, P') \quad \Gamma_2, x{:}A; R \vdash e_2 : (B, R') \quad P + K_1^{\text{let}} = \pi_{\vec{0}}^{\Gamma_1}(Q) \quad P' = \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}} \quad R' = Q' + K_3^{\text{let}} \\ \forall \vec{0} \neq j \in I(\Gamma_2)\colon\ \Gamma_1; P_j \overset{\text{cf}}{\vdash} e_1 : (A, P'_j) \quad P_j = \pi_j^{\Gamma_1}(Q) \quad P'_j = \pi_j^{x:A}(R)\end{array}}{\Gamma_1, \Gamma_2; Q \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : (B, Q')}\ (\text{T:Let})$$

$$\frac{\begin{array}{c}\Gamma; P \vdash e_t : (A, P') \quad P + K_1^{\text{conT}} = \pi_0^{\Gamma}(Q) \quad P' = Q' + K_2^{\text{conT}} \\ \Gamma; R \vdash e_f : (A, R') \quad R + K_1^{\text{conF}} = \pi_0^{\Gamma}(Q) \quad R' = Q' + K_2^{\text{conF}}\end{array}}{\Gamma, x{:}\mathit{bool}; Q \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f : (A, Q')}\ (\text{T:Cond}) \qquad \frac{\begin{array}{c}\Gamma, x_1{:}A_1, x_2{:}A_2; P \vdash e : (B, P') \\ P + K_1^{\text{matP}} = Q \quad P' = Q' + K_2^{\text{matP}}\end{array}}{\Gamma, x{:}A; Q \vdash \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e : (B, Q')}\ (\text{T:MatP})$$

$$\frac{Q = Q' + K^{\text{pair}}}{x_1{:}A_1, x_2{:}A_2; Q \vdash (x_1, x_2) : ((A_1, A_2), Q')}\ (\text{T:Pair}) \qquad \frac{q_0 = K^{\text{nil}} \quad q'_0 = 0}{\emptyset; Q \vdash \mathit{nil} : (L(A), Q')}\ (\text{T:Nil}) \qquad \frac{q_0 = K^{\text{leaf}} \quad q'_0 = 0}{\emptyset; Q \vdash \mathit{leaf} : (T(A), Q')}\ (\text{T:Leaf})$$

$$\frac{Q = \lhd_L(Q') + K^{\text{cons}}}{x_h{:}A, x_t{:}L(A); Q \vdash \mathit{cons}(x_h, x_t) : (L(A), Q')}\ (\text{T:Cons}) \qquad \frac{Q = \lhd_T(Q') + K^{\text{node}}}{x_0{:}A, x_1{:}T(A), x_2{:}T(A); Q \vdash \mathit{node}(x_0, x_1, x_2) : (T(A), Q')}\ (\text{T:Node})$$

$$\frac{\begin{array}{c}\Gamma; R \vdash e_1 : (B, R') \quad R + K_1^{\text{matN}} = \pi_0^{\Gamma}(Q) \\ R' = Q' + K_2^{\text{matN}} \quad \Gamma, x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P') \quad P + K_1^{\text{matC}} = \lhd_L(Q) \quad P' = Q' + K_2^{\text{matC}}\end{array}}{\Gamma, x{:}L(A); Q \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : (B, Q')}\ (\text{T:MatL})$$

$$\frac{\begin{array}{c}\Gamma; R \vdash e_1 : (B, R') \quad R + K_1^{\text{matTL}} = \pi_0^{\Gamma}(Q) \\ R' = Q' + K_2^{\text{matTL}} \quad \Gamma, x_0{:}A, x_1{:}T(A), x_2{:}T(A); P \vdash e_2 : (B, P') \quad P + K_1^{\text{matTN}} = \lhd_T(Q) \quad P' = Q' + K_2^{\text{matTN}}\end{array}}{\Gamma, x{:}T(A); Q \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 : (B, Q')}\ (\text{T:MatT})$$

$$\frac{\Gamma, x{:}A, y{:}A; P \vdash e : (B, Q') \quad Q = \curlyvee(P)}{\Gamma, z{:}A; Q \vdash e[z/x, z/y] : (B, Q')}\ (\text{T:Share}) \qquad \frac{\Gamma; P \vdash e : (B, P') \quad Q \geq P \quad Q' \leq P'}{\Gamma; Q \vdash e : (B, Q')}\ (\text{T:Weaken})$$

$$\frac{\Gamma; P \vdash e : (B, P') \quad Q = P + c \quad Q' = P' + c}{\Gamma; Q \vdash e : (B, Q')}\ (\text{T:Offset}) \qquad \frac{\Gamma; P \vdash e : (B, P') \quad \forall i \in I(\Gamma)\colon\ p_i = q_{(i,0)}}{\Gamma, x{:}A; Q \vdash e : (B, Q')}\ (\text{T:Augment})$$

Figure 2. Type rules for annotated types.

Lemma 6 is a consequence of Corollary 1. Moreover, the coefficients $c_k^{(i,j)}$ can be computed effectively and are *natural* numbers. For a context $\Gamma, x_1{:}A, x_2{:}A; Q$ we define $\curlyvee(Q)$ to be the Q' from Lemma 6.

Type Rules Fig. 2 shows the annotated type rules for RAML expressions. We assume a fixed global signature Σ that we omit from the rules. The last four rules are structural rules that apply to every expression. The other rules are syntax-driven and there is one rule for every construct of the syntax. In the implementation we incorporated the structural rules in the syntax-driven ones. The most interesting rules are explained below.

T:Share has to be applied to expressions that contain a variable twice (z in the rule). The sharing operation $\curlyvee(P)$ transfers the annotation P for the context $\Gamma, x{:}A, y{:}A$ into an annotation Q for the context $\Gamma, z{:}A$ without loss of potential (Lemma 6). This is crucial for the accuracy of the analysis since instances of T:Share are quite frequent in typical examples. The remaining rules are affine linear in the sense that they assume that every variable occurs at most once.

T:Cons assigns potential to a lengthened list. The additive shift $\lhd_L(Q')$ transforms the annotation Q' for a list type into an annotation for the context $x_h{:}A, x_t{:}L(A)$. Lemma 4 shows that potential is neither gained nor lost by this operation. The potential Q of the context has to pay for both the potential Q' of the resulting list and the resource cost K^{cons} for list cons.

T:MatL shows how to treat pattern matching of lists. The initial potential defined by the annotation Q of the context $\Gamma, x{:}L(A)$ has to be sufficient to pay the costs of the evaluation of e_1 or e_2 (depending on whether the matched list is empty or not) and the potential defined by the annotation Q' of the result type. To type the expression e_1 of the nil case we use the projection $\pi_0^{\Gamma}(Q)$ that results in an annotation for the context Γ. Since the matched list is empty in this case no potential is lost by the discount of the annotations $q_{(i,j)}$ of Q where $j \neq 0$. To type the expression e_2 of the cons case we rely on the shift operation $\lhd_L(Q)$ for lists that results in an annotation for the context $\Gamma, x_h{:}A, x_t{:}L(A)$. Again there is no loss of potential (see Lemma 4). The equalities relate the potential before and after the evaluation of e_1 or e_2, to the potential before the and after the evaluation of the match operation by incorporating the respective resource cost for the matching.

T:NODE and T:MATT are similar to the corresponding rules for lists but use the shift operator $\lhd_T$ for trees (see Lemma 5).

T:LET comprises essentially an application of Proposition 2 (with $f = e_1$ and $C = \Gamma_2$) followed by an application of Proposition 1 (with f being the parallel composition of e_1 and the identity on Γ_2 and g being e_2). Of course, the rigorous soundness proof takes into account partiality and additional constant costs for dispatching a let. It is part of the inductive soundness proof for the entire type system (Theorem 1).

The derivation of the type judgment $\Gamma_1, \Gamma_2; Q \vdash$ *let* $x = e_1$ *in* $e_2 : (B, Q')$ can be explained in two steps. The first starts with the derivation of the judgment $\Gamma_1; P \vdash e_1 : (A, P')$ for the sub-expression e_1. The annotation P corresponds to the potential that is exclusively attached to Γ_1 by the annotation Q plus some resource cost for the *let*, namely $P = \pi_{\vec{0}}^{\Gamma_1}(Q) + K_1^{\text{let}}$. Now we derive the judgment $\Gamma_2, x{:}A; R \vdash e_2 : (B, R')$. The potential that is assigned by R to $x{:}A$ is the potential that resulted from the judgment for e_1 plus some cost that might occur when binding the variable x to the value of e_1, namely $P' = \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}}$. The potential that is assigned by R to Γ_2 is essentially the potential that is assigned by to Γ_2 by Q, namely $\pi_{\vec{0}}^{\Gamma_2}(Q) = \pi_0^{\Gamma_2}(R)$. The second step of the derivation is to relate the annotations in R that refer to mixed potential between $x{:}A$ and Γ_2 to the annotations in Q that refer to potential that is mixed between Γ_1 and Γ_2. To this end we remember that we can derive from a judgment $\Gamma_1; S \vdash e_1 : (A, S')$ that $\Phi(\Gamma_1; S) \geq \Phi(v{:}(A, S'))$ if e_1 evaluates to v. This inequality remains valid if multiplied with a potential for $\phi_{\Gamma_2} = \Phi(\Gamma_2; T)$, i.e., $\Phi(\Gamma_1; S) \cdot \phi_{\Gamma_2} \geq \Phi(v{:}(A, S')) \cdot \phi_{\Gamma_2}$. To relate the mixed potential annotations we thus derive a cost-free judgment $\Gamma_1; P_j \overset{\text{cf}}{\vdash} e_1 : (A, P_j')$ for every $\vec{0} \neq j \in I(\Gamma_2)$. (We use cost-free judgments to avoid paying multiple times for the evaluation of e_1.) Then we equate P_j to the corresponding annotations in Q and equate P_j' to the corresponding annotations in R, i.e., $P_j = \pi_j^{\Gamma_1}(Q)$ and $P_j' = \pi_j^{x:A}(R)$. The intuition is that j corresponds to ϕ_{Γ_2}. Note that we use a fresh signature Σ in the derivation of each cost-free judgment for e_1.

Soundness The main theorem of this paper states that type derivations establish correct bounds: an annotated type judgment for an expression e shows that if e evaluates to a value v in a well-formed environment then the initial potential of the context is an upper bound on the watermark of the resource usage and the difference between initial and final potential is an upper bound on the consumed resources.

Note that it is possible to prove that the bounds also hold for non-terminating evaluations as we did for the univariate system [17] in a companion paper (see the discussion in §3).

Theorem 1 (Soundness) *Let* $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ *and* $\Sigma; \Gamma; Q \vdash e{:}(A, Q')$. *If* $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ *then* $p \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q)$ *and* $p - p' \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma; Q) - \Phi_{\mathcal{H}'}(v{:}(A, Q'))$.

Theorem 1 is proved by a nested induction on the derivation of the evaluation judgment $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ and the type judgment $\Gamma; Q \vdash e{:}(A, Q')$. The inner induction on the type judgment is needed because of the structural rules. There is one proof for all possible instantiations of the resource constants. It is technically involved but conceptually unsurprising. Compared to earlier works [16], further complexity arises from the new rich potential annotations. It is mainly dealt with in Lemmas 4, 5, and 6 and the concept of projections as explained in Propositions 2 and 3.

7. Type Inference and Experiments

Type Inference The type-inference algorithm for RAML extends the algorithm that we have developed for the univariate polynomial system [17]. It is not complete with respect to the type rules in §6 but it works well for the example programs we tested.

Its basis is a classic type inference generating simple linear constraints for the annotations that are collected during the inference, and that can be solved later by linear programming. In order to obtain a finite set of constraints one has to provide a maximal degree of the resource bounds. If the degree is too low then the generated linear program is unsolvable. The maximal degree can either be specified by the user or can be incremented successively after an unsuccessful analysis.

A main challenge in the inference is the handling of resource-polymorphic recursion which we believe to be of very high complexity if not undecidable in general. To deal with it practically, we employ a heuristic that has been developed for the univariate system.

In a nutshell, a function is allowed to invoke itself recursively with a type different from the one that is being justified (polymorphic recursion) provided that the two types differ only in lower-degree terms. In this way, one can successively derive polymorphic type schemes for higher and higher degrees; for details, see [17]. The generalisation of this approach to the multivariate setting poses no extra difficulties.

The number of multivariate polynomials our type system takes into account (e.g., $nm, n\binom{m}{2}, n\binom{m}{3}, m\binom{n}{2}, m\binom{n}{3}, \binom{n}{2}\binom{m}{2}$ for a pair of integer lists if the max. degree is 4) grows exponentially in the maximal degree. Thus the number of inequalities we collect for a fixed program grows also exponentially in the given maximal degree. Moreover, one often has to analyze function applications context-sensitively with respect to the call stack. Recall, e.g., the expression *filter(a,filter(b,l))* from §2 where we had to use two different types for *filter*.

Experimental Evaluation In our prototype implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph. For larger programs this can lead to large linear constraint systems if the maximal degree is high. Sometimes they are infeasible for the LP solver[6] we use.

Our emphasis was on correctness of the prototype, not on performance. There certainly is room for improvement, either by tuning the configuration of the current LP solver[7] or by experimenting with alternative solvers. Further improvement is possible by finding a suitable heuristic that is in between the (maybe too) flexible method we use here and the inference for the univariate system that also works efficiently with high maximal degree for large programs. For example, we could set certain coefficients q_i to zero before even generating the constraints. Alternatively, we could limit the number of different types for each function.

However, we are satisfied with the performance of the prototype on the example programs that do not require high degrees. For instance, we successfully analyzed longer examples with up to degree 4 (multiplication of a list of matrices).

Table 1 shows a compilation of the computation of evaluation-step bounds for several example functions. All computed bounds are asymptotically tight. The run-time of the analysis varies from 0.02 to 1.96 seconds on an 3.6 GHz Intel Core 2 Duo iMac with 4 GB RAM depending on the needed degree and the complexity of the source program.

Our experiments show that the constant factors in the computed bounds are generally quite tight and even match the measured

[6] lp_solve version 5.5.0.1

[7] Currently, we use the standard configuration with no additional options.

Function	Computed Evaluation-Step Bound	Simplified Computed Bound	Act. Behav.	Run Time
isortlist:$L(L(\mathit{int}))\to L(L(\mathit{int}))$	$\sum_{1\leq i<j\leq n} 16m_i+16\binom{n}{2}+12n+3$	$8n^2m+8n^2-8nm+4n+3$	$O(n^2m)$	0.91 s
nub:$L(L(\mathit{int}))\to L(L(\mathit{int}))$	$\sum_{1\leq i<j\leq n} 12m_i+18\binom{n}{2}+12n+3$	$6n^2m+9n^2-6nm+3n+3$	$O(n^2m)$	0.97 s
transpose:$L(L(\mathit{int}))\to L(L(\mathit{int}))$	$\sum_{1\leq i\leq n} 32m_i+2n+13$	$32nm+2n+13$	$O(nm)$	0.04 s
mmult:$(L(L(\mathit{int})))^2\to L(L(\mathit{int}))$	$(\sum_{1\leq i\leq x} y_i)(32+28n)+14n+2x+21$	$28xyn+32xy+2x+14n+21$	$O(nxy)$	1.96 s
dyade:$(L(\mathit{int}),L(\mathit{int}))\to L(L(\mathit{int}))$	$10nx+14n+3$	$10nx+14n+3$	$O(nx)$	0.03 s
lcs:$(L(\mathit{int}),L(\mathit{int}))\to \mathit{int}$	$39nx+6x+21n+19$	$39nx+6x+21n+19$	$O(nx)$	0.36 s
subtrees:$T(\mathit{int})\to L(T(\mathit{int}))$	$8\binom{n}{2}+23n+3$	$4n^2+19n+3$	$O(n^2)$	0.06 s
eratos:$L(\mathit{int})\to L(\mathit{int})$	$16\binom{n}{2}+12n+3$	$8n^2+4n+3$	$O(n^2)$	0.02 s

Table 1. The computed evaluation-step bounds, the actual worst-case time behavior, and the run time of the analysis in seconds. All computed bounds are asymptotically tight and the constant factors are close to the worst-case behavior. In the bounds n is the size of the first argument, m_i are the sizes of the elements of the first argument, x is the size of the second argument, y_i are the sizes of the elements of the second argument, $m = \max_{1\leq i\leq n} m_i$, and $y = \max_{1\leq i\leq x} y_i$.

worst-case running times of many functions. The univariate analysis [16, 17] infers identical bounds for the functions *subtrees* and *eratos*. In contrast, it can infer bounds for the other functions only after manual source-code transformations. Even then, the resulting bounds are not asymptotically tight.

We present the experimental evaluation of two functions below. The source code and the experimental validation for the other examples is available online[8]. It is also possible to download the source code of the prototype and to analyze user generated examples directly on the web.

Example 1: Lexicographic Sorting of Lists of Lists The following RAML code implements the well-known sorting algorithm insertion sort that lexicographically sorts lists of lists. To lexicographically compare two lists one needs linear time in length of the shorter one. Since insertion sort does quadratic many comparisons in the worst-case it has a running time of $O(n^2m)$ if n is the length of the outer list and m is the maximal length of the inner lists.

```
leq (l1,l2) = match l1 with | nil -> true
   | (x::xs) -> match l2 with | nil -> false
        | (y::ys) -> (x<y) or ((x == y) and leq (xs,ys));


insert (x,l) = match l with | nil -> [x]
               | (y::ys) -> if leq(x,y) then x::y::ys
                            else y::insert(x,ys);

isortlist l = match l with | nil -> nil
              | (x::xs) -> insert (x,isortlist xs);
```

Below is the analysis' output for the function *isortlist* when instantiated to bound the number of needed evaluation steps. The computation needs less then a second on typical desktop computers.

```
isortlist: L(L(int)) --> L(L(int))
Positive annotations of the argument
0     --> 3.0        2     --> 16.0
1     --> 12.0       [1,0] --> 16.0

The number of evaluation steps consumed by isortlist
is at most: 8.0*n^2*m + 8.0*n^2 - 8.0*n*m + 4.0*n + 3.0
where
   n is the length of the input
   m is the length of the elements of the input
```

The more precise bound implicit in the positive annotations of the argument is presented in mathematical notation in Table 1.

[8] http://raml.tcs.ifi.lmu.de

We manually identified inputs for which the worst-case behavior of *isortlist* emerges (namely reversely sorted lists with similar inner lists). Then we measured the needed evaluation steps and compared the results to our computed bound. Fig. 3 shows a plot of this comparison. Our experiments indicate that the computed bound exactly matches the actual worst-case behavior.

Example 2: Longest Common Subsequence An example of dynamic programming that can be found in many textbooks is the computation of (the length of) the longest common subsequence (LCS) of two given lists (sequences). If the sequences $a_1,\ldots,a_n$ and $b_1,\ldots,b_m$ are given then an $n\times m$ matrix (here a list of lists) A is successively filled such that $A(i,j)$ contains the length of the LCS of $a_1,\ldots,a_i$ and $b_1,\ldots,b_j$. The following recursion is used in the computation.

$$A(i,j)=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ A(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j \\ \max(A(i,j-1),A(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j \end{cases}$$

The run time of the algorithm is thus $O(nm)$. Below is the RAML implementation of the algorithm.

```
lcs(l1,l2) = let m = lcstable(l1,l2) in
    match m with | nil -> 0
      | (l1::_) -> match l1 with | nil -> 0
                     | (len::_) -> len;

lcstable (l1,l2) = match l1 with | nil -> [firstline l2]
      | (x::xs) -> let m = lcstable (xs,l2) in
                match m with | nil -> nil
                  | (l::ls) -> (newline (x,l,l2))::l::ls;

newline (y,lastline,l) = match l with | nil -> nil
   | (x::xs) -> match lastline with | nil -> nil
       | (belowVal::lastline') ->
           let nl = newline(y,lastline',xs) in
           let rightVal = right nl in
           let diagVal = right lastline' in
           let elem = if x == y then diagVal+1
                      else max(belowVal,rightVal)
           in elem::nl;

firstline(l) = match l with | nil -> nil
               | (x::xs) -> 0::firstline xs;


right l = match l with | nil -> 0 | (x::xs) -> x;
```

The analysis of the program takes less then a second on a usual desktop computer and produces the following output for the function *lcs*.

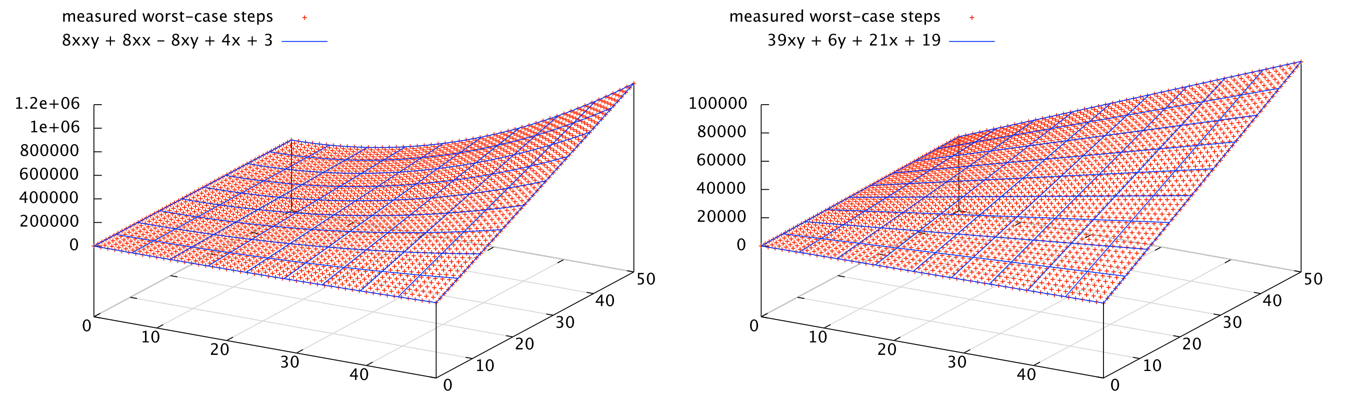

Figure 3. The computed evaluation-step bound (lines) compared to the actual worst-case number of evaluation-steps for sample inputs of various sizes (crosses) used by *isortlist* (on the left) and *lcs* (on the right).

```
lcs: (L(int),L(int)) --> int
Positive annotations of the argument
(0,0) --> 19.0      (1,0) --> 21.0
(0,1) --> 6.0       (1,1) --> 39.0

The number of evaluation steps consumed by lcs is at
most:     39.0*m*n + 6.0*m + 21.0*n + 19.0
where
   n is the length of the first component of the input
   m is the length of the second component of the input
```

Fig. 3 shows that the computed bound is close to the measured number of evaluation steps needed. In the case of *lcs* the run time exclusively depends on the lengths of the input lists.

8. Related Work

Most closely related is the previous work on automatic amortized analysis [17, 16, 18, 19, 20, 23, 24] (see §1). This paper describes the first system that can compute multivariate polynomial bounds.

Other resource analyses that can in principle obtain polynomial bounds are approaches based on recurrences pioneered by Grobauer [12] and Flajolet [11]. In those systems, an a priori unknown resource bounding function is introduced for each function in the code; by a straightforward intraprocedural analysis a set of recurrence equations or inequalities for these functions is then derived. Even for relatively simple programs the resulting recurrences are quite complicated and difficult to solve with standard methods.

In the COSTA project [1, 2, 3] progress has been made with the solution of those recurrences. In an automatic complexity analysis for higher-order Nuprl terms Benzinger uses Mathematica to solve the generated recurrence equations [5]. The size measures used in these approaches (like the length of the longest path in the input data) are less precise for nested data structures than our resource polynomials which comprise the sizes of all inner data structures. As a result, our method can deal with compositions of functions more accurately and is able to express a wider range of relations between parts of the input. We also find that amortization yields better results in cases where resource usage of intermediate functions depends on factors other than input size, e.g., sizes of partitions in quick sort.

A successful method to estimate time bounds for C++ procedures with loops and recursion was recently developed by Gulwani et al. [15, 13] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. A recent innovation for non-recursive programs is the combination of disjunctive invariant generation via abstract interpretation with proof rules that employ SMT-solvers [14]. In contrast to our method, these techniques can not fully automatically analyze iterations over data structures. Instead, the user needs to define numerical "quantitative functions". This seems to be less modular for nested data structures where the user needs to specify an "owner predicate" for inner data structures. It is also unclear if quantitative functions can represent complex mixed bounds such as $\sum_{1\leq i<j\leq n}(10m_i{+}2m_j){+}16\binom{n}{2}{+}12n{+}3$ for *isortlist*. Moreover, our method infers tight bounds for functions such as insertion sort that admit a worst-case time usage of the form $\sum_{1\leq i\leq n} i$. In contrast, [15] indicates that a nested loop on $1 \leq i \leq n$ and $1 \leq j \leq i$ is over-approximated with the bound n^2.

A methodological difference to techniques based on abstract interpretation is that we infer (using linear programming) an abstract potential function which indirectly yields a resource-bounding function. The potential-based approach may be favorable in the presence of compositions and data scattered over different locations (partitions in quick sort). As any type system, our approach is naturally compositional and lends itself to the smooth integration of components whose implementation is not available. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [6]. On the other hand, our method does not model the interaction of integer arithmetic with resource usage.

Other related works use type systems to validate resource bounds. Crary and Weirich [9] presented a (monomorphic) type system capable of specifying and certifying resource consumption. Danielsson [10] provided a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of purely functional data structures and algorithms. In contrast, our focus is on the inference of bounds.

Another related approach is the use of sized types [22, 21, 8] which provide a general framework to represent the size of the data in its type. Sized types are a very important concept and we also employ them indirectly. Our method adds a certain amount of data dependency and dispenses with the explicit manipulation of symbolic expressions in favour of numerical potential annotations.

Polynomial resource bounds have also been studied in [25] that addresses the derivation of polynomial size bounds for functions whose exact growth rate is polynomial. Besides this strong restriction, the efficiency of inference remains unclear.

9. Conclusion and Directions for Future Work

We have introduced a quantitative amortized analysis for first-order functions with multiple arguments. For the first time, we have been able to fully automatically derive complex multivariate resource bounds for recursive functions on nested inductive data structures such as lists and trees. Our experiments have shown that the analysis is sufficiently efficient for the functions we have tested, and that the resulting bounds are not only asymptotically tight but are also surprisingly precise in terms of constant factors.

The system we have developed will be the basis of various future projects. A challenging unsolved problem we are interested in is the computation of precise heap-space bounds in the presence of automatic memory management.

We have first ideas for extending the type system to derive bounds that contain not only polynomial but also involve logarithmic and exponential functions. The extension of linear amortized analysis to polymorphic and higher-order programs [24] seems to be compatible with our system and it would be interesting to integrate it. Finally, we plan to investigate to what extent our multivariate amortized analysis can be used for programs with cyclic data structures (following [19, 20, 4]) and recursion (including loops) on integers. For the latter it might be beneficial to merge the amortized method with successful existing techniques on abstract interpretation [15, 3].

Another very interesting and rewarding piece of future work would be an adaptation of our method to imperative languages without built-in inductive types such as C. One could try to employ pattern-based discovery of inductive data structures as is done, e.g., in separation logic.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*, pages 157–172, 2007.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *15th Static Analysis Symp. (SAS'08)*, pages 221–237, 2008.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.*, 258(1):109–121, 2009.

[4] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[5] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Log. f. Prog., AI, and Reas., 11th Conf. (LPAR'04)*, pages 347–362, 2004.

[7] B. Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *18th Euro. Symp. on Prog. (ESOP'09)*, pages 190–204, 2009.

[8] W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *High.-Ord. and Symb. Comp.*, 14(2-3):261–300, 2001.

[9] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

[10] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, pages 133–144, 2008.

[11] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-case Analysis of Algorithms. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.

[12] B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.

[13] B. S. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Comp. Aid. Verification, 20th Int. Conf. (CAV '08)*, pages 370–384, 2008.

[14] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

[15] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[16] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 287–306, 2010.

[17] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *8th Asian Symp. on Prog. Langs. (APLAS'10)*, 2010. To appear.

[18] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[19] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

[20] M. Hofmann and D. Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *18th Conf. on Comp. Science Logic (CSL'09)*. LNCS, 2009.

[21] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *4th Int. Conf. on Funct. Prog. (ICFP'99)*, pages 70–81, 1999.

[22] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *23th ACM Symp. on Principles of Prog. Langs. (POPL'96)*, pages 410–423, 1996.

[23] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, pages 354–369, 2009.

[24] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, pages 223–236, 2010.

[25] O. Shkaravska, R. van Kesteren, and M. C. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calc. Apps. (TLCA'07)*, pages 351–365, 2007.

[26] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[27] R. Wilhelm et al. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.