Type-Based Amortized Resource Analysis with Integers and Arrays

Jan Hoffmann and Zhong Shao

Yale University

Abstract. Proving bounds on the resource consumption of a program by statically analyzing its source code is an important and well-studied problem. Automatic approaches for numeric programs with side effects usually apply abstract interpretation–based invariant generation to derive bounds on loops and recursion depths of function calls.
This paper presents an alternative approach to resource-bound analysis for numeric, heap-manipulating programs that uses type-based amortized resource analysis. As a first step towards the analysis of imperative code, the technique is developed for a first-order ML-like language with unsigned integers and arrays. The analysis automatically derives bounds that are multivariate polynomials in the numbers and the lengths of the arrays in the input. Experiments with example programs demonstrate two main advantages of amortized analysis over current abstract interpretation–based techniques. For one thing, amortized analysis can handle programs with non-linear intermediate values like $f((n+m)^2)$. For another thing, amortized analysis is compositional and works naturally for compound programs like $f(g(x))$.

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

1 Introduction

The quantitative performance characteristics of a program are among the most important aspects that determine whether the program is useful in practice. Manually proving concrete (non-asymptotic) resource bounds with respect to a formal machine model is tedious and error-prone. This is especially true if programs evolve over time when bugs are fixed or new features are added. As a result, automatic methods for inferring resource bounds are extensively studied. The most advanced techniques for imperative programs with integers and arrays apply abstract interpretation to generate numerical invariants [1–4], that is, bounds on the values of variables. These invariants form the basis of the computation of actual bounds on loop iterations and recursion depths.

For reasons of efficiency, many abstract interpretation–based resource-analysis systems rely on abstract domains such as polyhedra [5] which enable the inference of invariants through linear constraint solving. The downside of this approach is that the resulting tools only work effectively for programs in which all relevant variables are bounded by *linear invariants.* This is, for example, not the case if programs perform non-linear arithmetic operations such as multiplication or division. A linear abstract domain can be used to derive non-linear invariants using domain lifting operations [6]. Another possibility is to use disjunctive abstract domains to generate non-linear invariants [7]. This technique has been experimentally implemented in the COSTA analysis system [8]. However, it is less mature than polyhedra-based invariant generation and it is unclear how it scales to larger examples.

In this paper, we study an alternative approach to infer resource bounds for numeric programs with side effects. Instead of abstract interpretation, it is based on type-based amortized resource analysis [9, 10]. It has been shown that this analysis technique can infer tight polynomial bounds for functional programs with nested data structures while relying on linear constraint solving only [11, 10]. A main innovation in this *polynomial amortized analysis* is the use of *multivariate resource polynomials* that have good closure properties and behave well under common size-change operations. Advantages of amortized resource analysis include precision, efficiency, and compositionality.

Our ultimate goal is to transfer the advantages of amortized resource analysis to imperative (C-like) programs. As a first important step, we develop a multivariate amortized resource analysis for numeric ML-like programs with mutable arrays in this work. We present the new technique for a simple language with unsigned integers, arrays, and pairs as the only data types in this paper. However, we implemented the analysis in Resource Aware ML (RAML) [12] which features more data types such as lists and binary trees. Our experiments (see Section 6) show that our implementation can automatically and efficiently infer complex polynomial bounds for programs that contain non-linear size changes like $f(8128 * x * x)$ and composed functions like $f(g(x))$ where the result of the function $g(x)$ is non-linear. RAML is publicly available and all of our examples as well as user-defined code can be tested in an easy-to-use online interface [12].

Technically, we treat unsigned integers like unary lists in multivariate amortized analysis [10]. However, we do not just instantiate the previous framework by providing a pattern matching for unsigned integers and implementing recursive functions. In fact, this approach would be possible but it has several shortcomings (see Section 2) that make it unsuitable in practice. The key for making amortized resource analysis work for numeric code is to give direct typing rules for the arithmetic operations *addition, subtraction, multiplication, division, and modulo.* The most interesting aspect of the rules we developed is that they can be readily represented with very succinct linear constraint systems. Moreover, the rules precisely capture the size changes in the corresponding operations in the sense that no precision (or potential) is lost in the analysis.

To deal with mutable data, the analysis ensures that the resource consumption does not depend on the size of data that has been stored in a mutable heap cell. While it would be possible to give more involved rules for array operations, all examples we considered could be analyzed with our technique. Hence we found that the additional complexity of more precise rules was not justified by the gain of expressivity in practice.

To prove the soundness of the analysis, we model the resource consumption of programs with a big-step operational semantics for terminating and non-terminating programs. This enables us to show that bounds derived within the type system hold for terminating and non-terminating programs. Refer to the literature for more detailed explanations of type-based amortized resource analysis [9, 11, 10], the soundness proof [13], and Resource Aware ML [12, 14].

The full version of this article is available online [15] and includes all technical details and additional explanations.

2 Informal Account

In this section we briefly introduce type-based amortized resource analysis. We then motivate and describe the novel developments for programs with integers and arrays.

Amortized Resource Analysis. The idea of type-based amortized resource analysis [9, 10] is to annotate each program point with a *potential function* which maps sizes of reachable data structures to non-negative numbers. The potential functions have to ensure that, for every input and every possible evaluation, the potential at a program point is sufficient to pay for the resource cost of the following transition and the potential at the next point. It then follows that the initial potential function describes an upper bound on the resource consumption of the program.

It is natural to build a practical amortized resource analysis on top of a type system because types are compositional and provide useful information about the structure of the data. In a series of papers [11, 10, 13, 14], it has been shown that *multivariate resource polynomials* are a good choice for the set of possible potential functions. Multivariate resource polynomials are a generalization of non-negative linear combinations of binomial coefficients that includes tight bounds for many typical programs [13]. At the same time, multivariate resource polynomials can be incorporated into type systems so that type inference can be efficiently reduced to LP solving [13].

The basic idea of amortized resource analysis is best explained by example. Assume we represent natural numbers as unary lists and implement addition and multiplication as follows.

```
add (n,m) = match n with | nil  →  m
    | _::xs  →  () :: (add (xs,m));

mult (n,m) = match n with | nil  →  nil
    | _::xs  →  add(m,mult(xs,m));
```

Assume furthermore that we are interested in the number of pattern matches that are performed by these functions. The evaluation of the expression $\mathsf{add}(\mathsf{n},\mathsf{m})$ performs $|n| + 1$ pattern matches and evaluating $\mathsf{mult}(\mathsf{n},\mathsf{m})$ needs $|n||m| + 2|n| + 1$ pattern matches. To represent these bounds in an amortized resource analysis, we annotate the argument and result types of the functions with indexed families of non-negative rational coefficients of our resource polynomials. The index set depends on the type and on the maximal degree of the bounds, which has to be fixed to make the analysis feasible. For our example mult we need degree 2. The index set for the argument type $A = L(\text{unit}) * L(\text{unit})$ is then $\mathcal{I}(A) = \{(0,0),(1,0),(2,0),(1,1),(0,1),(0,2)\}$. A family $Q = (q_i)_{i \in \mathcal{I}(A)}$ denotes the resource polynomial that maps two lists n and m to the number $\sum_{(i,j)\in\mathcal{I}(A)} q_{(i,j)} \binom{|n|}{i}\binom{|m|}{j}$. Similarly, an indexed family $P = (p_i)_{i\in\{0,1,2\}}$ describes the resource polynomial $\ell \mapsto p_0 + p_1|\ell| + p_2\binom{|\ell|}{2}$ for a list $\ell : L(\text{unit})$.

A valid typing for the multiplication would be for instance $\mathsf{mult} : (L(\text{unit}) * L(\text{unit}), Q) \to (L(\text{unit}), P)$, where $q_{(0,0)} = 1, q_{(1,0)} = 2, q_{(1,1)} = 1$, and $q_i = p_j = 0$ for all other i and all j. Another valid instantiation of P and Q, which would be needed in a larger program such as $\mathsf{add}(\mathsf{mult}(\mathsf{n},\mathsf{m}),\mathsf{k})$, is $q_{(0,0)} = q_{(1,0)} = q_{(1,1)} = 2$, $p_0 = p_1 = 1$ and $q_i = p_j = 0$ for all other i and all j.

The challenge in designing an amortized resource analysis is to develop a type rule for each syntactic construct of a program that describes how the potential before the evaluation relates to the potential after the evaluation. It has been shown [11, 13] that the structure of multivariate resource polynomials facilitates the development of relatively simple type rules. These rules enable the generation of linear constraint systems such that a solution of a constraint system corresponds to a valid instantiation of the rational coefficients q_i and p_j.

Numerical Programs and Side Effects. Previous work on polynomial amortized analysis [11, 13] (that is implemented in RAML) focused on inductive data structures such as trees and lists. In this paper, we are extending the technique to programs with unsigned integers, arrays, and the usual atomic operations such as $*$, $+$, $-$, mod, div, set, and get. Of course, it would be possible to use existing techniques and a *code transformation* that converts a program with these operations into one that uses recursive implementations such as the previously defined functions add and mult. However, this approach has multiple shortcomings.

Efficiency In programs with many arithmetic operations, the use of recursive implementations causes the analysis to generate large constraint systems that are challenging to solve. Figure 1 shows the number of constraints that are generated by the analysis for a program with a single multiplication $a * b$ as a function of the maximal degree of the bounds. With our novel handcrafted rule for multiplication the analysis creates for example 82 constraints when searching for bounds of maximal degree 10. With the recursive implementation, 408653 constraints are generated. IBM's Cplex can still solve this constraint system in a few seconds but a precise analysis of a larger RAML program currently requires to copy the 408653 constraints for every multiplication in the program. This makes the analysis infeasible.

Effectivity A straightforward recursive implementation of the arithmetic operations on unary lists in RAML would not allow us to analyze the same range of functions we can analyze with handcrafted typing rules for the operations. For example, the fast Euclidean algorithm cannot be analyzed with the usual, recursive definition of mod but can be analyzed with our new

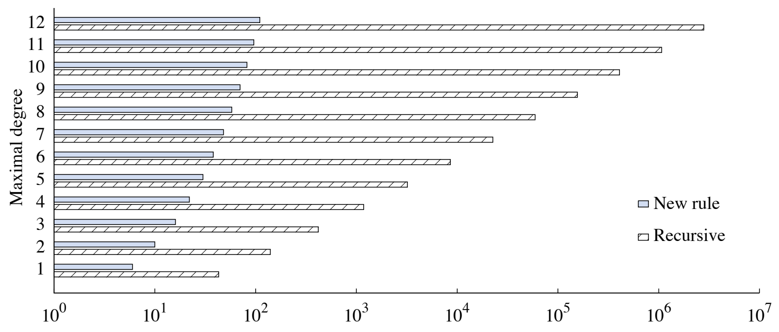

Fig. 1. Number of constraint generated by RAML for the program a ∗ b as a function of the maximal degree. The solid bars show the number of constraints generated using the novel type rule for multiplication. The striped bars show the number of constrained generated using an recursive implementation. The scale on the x-axis is logarithmic.

rule. Similarly, we cannot define a recursive function so that the analysis is as effective as with our novel rule for minus. For example, the pattern if n $>$ C then ... recCall(n $-$ C) else ... for a constant $C > 0$ can be analyzed with our new rule but not with a recursive definition for minus.

Conception A code transformation prior to the analysis complicates the soundness proof since we would have to show that the resource usage of the modified code is equivalent to the resource usage of the original code. More importantly, handling new language features merely by code transformations into well-understood constructs is conceptually less attractive since it often does not advance our understanding of the new features.

To derive a typing rule for an arithmetic operation in amortized resource analysis, we have to describe how the potential of the arguments of the operation relates to the potential of the result. For $x, y \in \mathbb{N}$ and a multiplication $x * y$ we start with a potential of the form $\sum_{(i,j)\in I} q_{(i,j)} \binom{x}{i}\binom{y}{j}$ (where $I = \{(0,0),(1,0),(2,0),(1,1),(0,1),(0,2)\}$ in the case of degree 2). We then have to ensure that this potential is always equal to the constant resource consumption M^{mult} of the multiplication and the potential $\sum_{i\in\{0,1,2\}} p_i \binom{x\cdot y}{i}$ of the result $x{\cdot}y$. This is the case if $q_{(0,0)} = M^{\mathsf{mult}} + p_0$, $q_{(1,1)} = p_1$, $q_{(1,2)} = q_{(2,1)} = p_2$, $q_{(2,2)} = 2p_2$, and $q_{(i,j)} = 0$ otherwise. We will show that such relations can be expressed for resource polynomials of arbitrary degree in a type rule for amortized resource analysis that corresponds to a succinct linear constraint system.

The challenge with arrays is to account for side effects of computations that influence the resource consumption of later computations in the presence of aliasing. We can analyze such programs but ensure that the potential of data that is stored in arrays is always 0. In this way, we prove that the influence of aliasing on the resource usage is accounted for without using the size of mutable data. As for all language features, we could achieve the same with some abstraction of the program that does not use arrays. However, this is not necessarily a simpler approach.

3 A Simple Language with Side Effects

We present our analysis for a minimal first-order functional language that only contains the features we are interested in, namely operations for integers and arrays. However, we implemented the analysis in Resource Aware ML (RAML) [14, 12] which also includes (signed) integers, lists, binary trees, Booleans, conditionals and pattern matching on lists and trees.

Syntax. The subset of RAML we use in this article includes variables x, unsigned integers n, function calls, pairs, pattern matching for unsigned integers and pairs, let bindings, an undefined expression, a sharing expression, and the built in operations for arrays and unsigned integers.

$$\begin{aligned} e ::= {} & x \mid f(x) \mid (x_1,x_2) \mid \mathsf{match}\, x\, \mathsf{with}\, (x_1,x_2) \Rightarrow e \mid \mathsf{undefined} \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \\ & \mid \mathsf{share}\, x\, \mathsf{as}\, (x_1,x_2)\, \mathsf{in}\, e \mid \mathsf{match}\, x\, \mathsf{with}\, \langle 0 \Rightarrow e_1 \mid \mathsf{S}(y) \Rightarrow e_2 \rangle \\ & \mid n \mid x_1 + x_2 \mid x_1 * x_2 \mid \mathsf{minus}(x_1,x_2) \mid \mathsf{minus}(x_1,n) \mid \mathsf{divmod}(x_1,x_2) \\ & \mid \mathsf{A.make}(x_1,x_2) \mid \mathsf{A.set}(x_1,x_2,x_3) \mid \mathsf{A.get}(x_1,x_2) \mid \mathsf{A.length}(x) \end{aligned}$$

We present the language in, what we call, share-let normal form which simplifies the type system without hampering expressivity. In the implementation, we transform input programs to share-let normal form before the analysis. Like in Haskell, the $\mathsf{undefined}$ expression simply aborts the program without consuming any resources. The meaning of the sharing expression $\mathsf{share}\, x\, \mathsf{as}\, (x_1,x_2)\, \mathsf{in}\, e$ is that the value of the free variable x is bound to the variables x_1 and x_2 for use in the expression e. We use it to inform the (affine) type system of multiple uses of a variable.

While all array operations as well as multiplication and addition are standard, subtraction, division, and modulo differ from the standard operations. To give stronger typing rules in our analysis system, we combine division and modulo in one operation divmod. Moreover, minus and divmod return their second argument, that is, $\mathsf{minus}(n,m) = (m, n-m)$ and $\mathsf{divmod}(n,m) = (m, n \div m, n \bmod m)$. We also distinguish two syntactic forms of minus; one in which we subtract a variable and another one in which we subtract a constant. More explanations are given in Section 5. If $m > n$ then the evaluation of $\mathsf{minus}(n,m)$ fails without consuming resources. That means that it is the responsibility of the user or other static analysis tools to show the absence of overflows.

Types and Programs. We define data types A, B and function types F.

$$A, B ::= \mathrm{nat} \mid A \text{ array} \mid A * B \qquad\qquad F ::= A \to B$$

Let $\mathcal{A}$ be the set of data types and let $\mathcal{F}$ be the set of function types. A signature $\Sigma : \mathrm{FID} \rightharpoonup \mathcal{F}$ is a partial finite mapping from function identifiers to function types. A context is a partial finite mapping $\Gamma : \mathit{Var} \rightharpoonup \mathcal{A}$ from variable identifiers to data types. A simple type judgment $\Sigma; \Gamma \vdash e : A$ states that the expression e has type A in the context Γ under the signature Σ. The definition of typing rules for this judgment is standard and we omit the rules. A *(well-typed) program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash e_f{:}B$ if $\Sigma(f) = A \to B$.

$$\frac{}{V, H \overset{M}{\vdash} e \Downarrow \circ \mid 0}\ (\text{E:Zero}) \qquad \frac{n = H(V(x_1)) \cdot H(V(x_2)) \qquad H' = H, \ell \mapsto n}{V, H \overset{M}{\vdash} x_1 * x_2 \Downarrow (\ell, H') \mid M^{\mathsf{mult}}}\ (\text{E:Mult})$$

$$\frac{n = H(V(x_1)) - H(V(x_2)) \qquad H' = H, \ell \mapsto (V(x_2), \ell'), \ell' \mapsto n}{V, H \overset{M}{\vdash} \mathsf{minus}(x_1, x_2) \Downarrow (\ell, H') \mid M^{\mathsf{sub}}}\ (\text{E:Sub})$$

$$\frac{[y_f \mapsto V(x)], H \overset{M}{\vdash} e_f \Downarrow \rho \mid (q, q')}{V, H \overset{M}{\vdash} f(x) \Downarrow \rho \mid M^{\mathsf{app}} \cdot (q, q')}\ (\text{E:App}) \qquad \frac{H(V(x_1)) = (\sigma, n) \quad H(V(x_2)) \geqslant n}{V, H \overset{M}{\vdash} \mathsf{A.get}(x_1, x_2) \Downarrow \circ \mid M^{\mathsf{Afail}}}\ (\text{E:AGFail})$$

$$\frac{H(V(x_1)) = (\sigma, n) \qquad H(V(x_2)) = i \qquad 0 \leqslant i < n}{V, H \overset{M}{\vdash} \mathsf{A.get}(x_1, x_2) \Downarrow (\sigma(i), H) \mid M^{\mathsf{Aget}}}\ (\text{E:AGet})$$

Fig. 2. Selected rules of the operational big-step semantics.

Cost Semantics. Figure 2 contains representative rules of the operational cost semantics. The full version of this article [15] contains all rules of the semantics for our subset of RAML. The semantics is standard except that it defines the cost of an evaluation. This cost depends on a resource metric $M : K \to \mathbb{Q}$ that assigns a cost to each evaluation step of the big-step semantics. Here, K is a finite set of constant symbols. We write M^k for $M(k)$.

The semantics is formulated with respect to a stack and a heap. Let *Loc* be an infinite set of *locations* modeling memory addresses on a heap. The set of RAML *values* *Val* is given as follows.

$$\mathit{Val} \ni v ::= n \mid (\ell_1, \ell_2) \mid (\sigma, n)$$

A value $v \in \mathit{Val}$ is either a natural number n, a pair of locations (ℓ_1, ℓ_2), or an array (σ, n). An array (σ, n) consists of a size n and a mapping $\sigma : \{0, \ldots, n-1\} \to \mathit{Loc}$ from the set $\{0, \ldots, n-1\}$ of natural numbers to locations. A *heap* is a finite partial mapping $H : \mathit{Loc} \rightharpoonup \mathit{Val}$ that maps locations to values. A *stack* is a finite partial mapping $V : \mathit{Var} \rightharpoonup \mathit{Loc}$ from variable identifiers to locations.

The big-step operational evaluation rules are defined in the full version of this article. They define an evaluation judgment of the form $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (q, q')$. It expresses the following. Under resource metric M, if the stack V and the initial heap H are given then the expression e evaluates to the location ℓ and the new heap H'. To evaluate e one needs at least $q \in \mathbb{Q}_0^+$ resource units and after the evaluation there are $q' \in \mathbb{Q}_0^+$ 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.

In fact, the evaluation judgment is slightly more complicated because there are two other behaviors that we have to express in the semantics: failure (i.e., array access outside its bounds) and divergence. To this end, our semantics judgment does not only evaluate expressions to values but also expresses incomplete computations by using $\circ$ (pronounced *busy*). The evaluation judgment has the general form

$$V, H \overset{M}{\vdash} e \Downarrow \rho \mid (q, q') \qquad \text{where} \qquad \rho ::= (\ell, H) \mid \circ \,.$$

Well-Formed Environments. For each simple type A we inductively define a set $[\![A]\!]$ of values of type A.

$$\begin{aligned}[\![\text{nat}]\!] &= \mathbb{N} \\ [\![A \text{ array}]\!] &= \{(\alpha, n) \mid n \in \mathbb{N} \text{ and } \alpha : \{0, \ldots, n-1\} \to [\![A]\!]\} \\ [\![A * B]\!] &= [\![A]\!] \times [\![B]\!]\end{aligned}$$

If H is a heap, ℓ is a location, A is a type, and $a \in [\![A]\!]$ then we write $H \vDash \ell \mapsto a : A$ to mean that ℓ defines the semantic value $a \in [\![A]\!]$ when pointers are followed in H in the obvious way. The judgment is formally defined in the full version.

We write $H \vDash \ell : A$ to indicate that there exists a necessarily unique, semantic value $a \in [\![A]\!]$ so that $H \vDash \ell \mapsto a : A$. A stack V and a heap H are *well-formed* with respect to a context Γ if $H \vDash V(x) : \Gamma(x)$ holds for every $x \in \text{dom}(\Gamma)$. We then write $H \vDash V : \Gamma$.

4 Resource Polynomials and Annotated Types

Compared with multivariate amortized resource analysis for nested inductive data types [13], the resource polynomials that are needed for the data types in this article are relatively simple. They are multivariate, non-negative linear combinations of binomial coefficients.

Resource Polynomials. For each data type A we first define a set $P(A)$ 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 non-negative rational linear combinations of these *base polynomials*. We define $P(A)$ as follows.

$$P(\text{nat}) = \{\lambda n \,.\, \binom{n}{k} \mid k \in \mathbb{N}\} \qquad P(A \text{ array}) = \{\lambda(\alpha, n) \,.\, \binom{n}{k} \mid k \in \mathbb{N}\}$$

$$P(A_1 * A_2) = \{\lambda(a_1, a_2) \,.\, p_1(a_1) \cdot p_2(a_2) \mid p_1 \in P(A_1) \wedge p_2 \in P(A_2)\}$$

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 $p_i \in P(A)$. We write $R(A)$ for the set of resource polynomials of data type A.

For example, $h(n, m) = 7 + 2.5 \cdot n + 5\binom{n}{3}\binom{m}{2} + 8\binom{m}{4}$ is a resource polynomial for the data type nat $*$ nat.

Names for Base Polynomials. To assign a unique name to each base polynomial, we define the *index set* $\mathcal{I}(A)$ to denote resource polynomials for a given data type A.

$$\begin{aligned}\mathcal{I}(\text{nat}) = \mathcal{I}(A \text{ array}) &= \mathbb{N} \\ \mathcal{I}(A_1 * A_2) &= \{(i_1, i_2) \mid i_1 \in \mathcal{I}(A_1) \text{ and } i_2 \in \mathcal{I}(A_2)\}\end{aligned}$$

For each $i \in \mathcal{I}(A)$, we define a base polynomial $p_i \in P(A)$ as follows: If $A = \text{nat}$ then $p_k(n) = \binom{n}{k}$. If $A = A'$ array then $p_k(\sigma, n) = \binom{n}{k}$. If $A = (A_1 * A_2)$ is a pair type and $v = (v_1, v_2)$ then $p_{(i_1,i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2)$. We use the notation 0_A (or just 0) for the index in $\mathcal{I}(A)$ such that $p_{0_A}(a) = 1$ for all a.

Our previous example $h : [\![\text{nat} * \text{nat}]\!] \to \mathbb{Q}_0^+$ can for instance be written as $h(n, m) = 7p_{(0,0)}(n, m) + 2.5p_{(1,0)}(n, m) + 5p_{(3,2)}(n, m) + 8p_{(0,4)}(n, m)$.

Annotated Types and Potential Functions. A *type annotation* for a data type A is defined to be a family $Q_A = (q_i)_{i\in\mathcal{I}(A)}$ with $q_i \in \mathbb{Q}_0^+$. An *annotated data type* is a pair (A, Q_A) of a data type A and a type annotation Q_A.

Let H be a heap and let ℓ be a location with $H \vDash \ell \mapsto a : A$ for a data type A. Then the type annotation Q_A defines the *potential*

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

If $a \in [\![A]\!]$ then we also write $\Phi(a : (A, Q_A))$ for $\sum_i q_i{\cdot}p_i(a)$.

For example, consider the resource polynomial $h(n, m)$ again. We have $\Phi((n, m) : (\text{nat}*\text{nat}, Q)) = h(n, m)$ if $q_{(0,0)} = 7$, $q_{(1,0)} = 2.5$, $q_{(3,2)} = 5$, $q_{(0,4)} = 8$, and $q_{(i,j)} = 0$ for all other $(i, j) \in \mathcal{I}(\text{nat} * \text{nat})$.

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 $\mathcal{I}(\Gamma)$ is defined as $\mathcal{I}(\Gamma) = \{(i_1, \ldots, i_n) \mid i_j \in \mathcal{I}(A_j)\}$. A *type annotation* Q for Γ is a family $Q = (q_i)_{i\in\mathcal{I}(\Gamma)}$ with $q_i \in \mathbb{Q}_0^+$.

We denote a *resource-annotated context* with $\Gamma; Q$. Let H be a heap and V be a stack with $H \vDash V : \Gamma$ where $H \vDash V(x_j) \mapsto a_{x_j} : \Gamma(x_j)$. The potential of $\Gamma; Q$ with respect to H and V is $\Phi_{V,H}(\Gamma; Q) = \sum_{(i_1,\ldots,i_n)\in\mathcal{I}(\Gamma)} q_{\vec{\imath}} \prod_{j=1}^{n} p_{i_j}(a_{x_j})$. In particular, if $\Gamma = \varnothing$ then $\mathcal{I}(\Gamma) = \{()\}$ and $\Phi_{V,H}(\Gamma; q_{()}) = q_{()}$. We sometimes also write q_0 for $q_{()}$.

Operations on Annotations. For each arithmetic operation such as $n - 1$, $n * m$, and $n + m$, we define a corresponding operation on annotations that describes how to transfer potential from the arguments to the result.

For addition and subtraction (compare rules T:Add and T:Sub in Figure 3) we need to express the potential of a natural number n in terms of two numbers n_1 and n_2 such that $n = n_1 + n_2$. To this end, let $Q = (q_i)_{i\in\mathbb{N}}$ be an annotation for data of type nat. We define the *convolution* $\boxplus(Q)$ of the annotation Q to be the following annotation Q' for the type nat $*$ nat.

$$\boxplus(Q) = (q'_{(i,j)})_{(i,j)\in\mathcal{I}(\text{nat}*\text{nat})} \qquad \text{if} \qquad q'_{(i,j)} = q_{i+j}$$

The convolution $\boxplus(Q)$ for type annotations corresponds to Vandermonde's convolution for binomial coefficients:

$$\binom{n_1 + n_2}{k} = \sum_{i+j=k} \binom{n_1}{i}\binom{n_2}{j}$$

Using Vandermonde's convolution we derive Lemma 1.

Lemma 1. *Let Q be an annotation for type nat, $H \vDash \ell \mapsto n_1{+}n_2 : nat$, and $H' \vDash \ell' \mapsto (n_1, n_2) : nat * nat$. Then $\Phi_H(\ell{:}(nat, Q)) = \Phi_{H'}(\ell'{:}(nat * nat, \boxplus(Q)))$.*

In the type rule for subtraction of a constant K we can distribute the potential in two different ways. We can either use the convolution to distribute the potential between two numbers or we can perform K additive shifts. Of course, we can describe K shift operations directly: Let $Q = (q_i)_{i \in \mathbb{N}}$ be an annotation for data of type nat. The *K-times shift for natural numbers* $\lhd^K(Q)$ of the annotation Q is an annotation Q' for data of type nat that is defined as follows.

$$\lhd^K(Q) = (q'_i)_{i \in \mathcal{I}(\mathrm{nat})} \qquad \text{if} \qquad q'_i = \sum_{j=i+\ell} q_j \binom{K}{\ell} .$$

Recall that $\binom{n}{m} = 0$ if $m > n$. The K-times shift corresponds to the following identity (where $q_{k+1} = 0$ again) that can be derived from Vandermonde's convolution.

$$\sum_{0 \leqslant i \leqslant k} q_i \binom{n+K}{i} = \sum_{0 \leqslant i \leqslant k} \left(\sum_{j=i+\ell} q_j \binom{K}{\ell} \right) \binom{n}{i}$$

The K-times shift is a generalization of the additive shift (see [13]) which is equivalent to the 1-times shift. Using the previous identity we prove Lemma 2.

Lemma 2. *Let Q be an annotation for type nat, $H \models \ell \mapsto n{+}K : nat$, and $H' \models \ell' \mapsto n : nat$. Then $\Phi_H(\ell{:}(nat, Q)) = \Phi_{H'}(\ell'{:}(nat, \lhd^K Q))$.*

For multiplication and division, things are more interesting. Our goal is to define a convolution-like operation $\boxdot(Q)$ that defines an annotation for the arguments $(x_1, x_2) : \mathrm{nat} * \mathrm{nat}$ if given an annotation Q of a product $x_1 * x_2 : \mathrm{nat}$. For this purpose, we are interested in the coefficients $A(i,j,k)$ in the following identity.

$$\binom{nm}{k} = \sum_{i,j} A(i,j,k) \binom{n}{i} \binom{m}{j}$$

Fortunately, this problem has been carefully studied by Riordan and Stein [16].[1] Intuitively, the coefficient $A(i,j,k)$ is number of ways of arranging k pebbles on an $i{\times}j$ chessboard such that every row and every column has at least one pebble. Riordan and Stein obtain the following closed formulas.

$$A(i,j,k) = \sum_{r,s} (-1)^{i+j+r+s} \binom{i}{r} \binom{j}{s} \binom{rs}{k} = \sum_n \frac{i!j!}{k!} S(n,i)\, S(n,j)\, s(k,n)$$

Here, $S(\cdot,\cdot)$ and $s(\cdot,\cdot)$ denote the Stirling numbers of first and second kind, respectively. Furthermore they report the recurrence relation $A(i,j,k{+}1)(k{+}1) = (A(i,j,k) + A(i-1,j,k) + A(i,j-1,k) + A(i-1,j-1,k))ij - k\,A(i,j,k)$.

Equipped with a closed formula for $A(i,j,k)$, we now define the *multiplicative convolution* $\boxdot(Q)$ of an annotation Q for type nat as

$$\boxdot(Q) = (q'_{(i,j)})_{(i,j) \in \mathcal{I}(\mathrm{nat}*\mathrm{nat})} \qquad \text{if} \qquad q'_{(i,j)} = \sum_k A(i,j,k)\, q_k .$$

Lemma 3 is then a direct consequence of the identity of Riordan and Stein.

[1] Thanks to Mike Spivey for pointing us to that article.

Lemma 3. *Let Q be an annotation for type nat, $H \models \ell \mapsto n_1 \cdot n_2 : nat$, and $H' \models \ell' \mapsto (n_1, n_2) : nat * nat$. Then $\Phi_H(\ell{:}(nat, Q)) = \Phi_{H'}(\ell'{:}(nat * nat, \boxdot(Q)))$.*

5 Resource-Aware Type System

We now describe the type-based amortized analysis for programs with unsigned integers and arrays. We only present the novel rules for arrays and arithmetic expressions. The complete set of rules can be found in the full version of the article [15].

Type Judgments. The type rules for RAML expressions in Figure 3 define a *resource-annotated typing judgment* of the form

$$\Sigma; \Gamma; Q \overset{M}{\vdash} e : (A, Q')$$

where e is a RAML expression, M is a metric, 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 cover the evaluation cost of e in metric M. In addition, there are at least $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

Programs with Annotated Types. Resource-annotated function 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 from function identifiers to *sets of* resource-annotated function types.

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

Notations. If Q, P and R are annotations with the same index set I then we extend operations on $\mathbb{Q}$ pointwise to Q, P and R. For example, we write $Q \leqslant P + R$ if $q_i \leqslant p_i + r_i$ for every $i \in I$.

For $K \in \mathbb{Q}$ we write $Q = Q' + K$ to state that $q_0 = q'_0 + K \geqslant 0$ and $q_i = q'_i$ for $i \neq 0 \in I$. Let $\Gamma = \Gamma_1, \Gamma_2$ be a context, let $i = (i_1, \ldots, i_k) \in \mathcal{I}(\Gamma_1)$ and $j = (j_1, \ldots, j_l) \in \mathcal{I}(\Gamma_2)$. We write (i, j) for the index $(i_1, \ldots, i_k, j_1, \ldots, j_l) \in \mathcal{I}(\Gamma)$.

Let Q be an annotation for a context Γ_1, Γ_2. For $j \in \mathcal{I}(\Gamma_2)$ we define the *projection* $\pi_j^{\Gamma_1}(Q)$ of Q to Γ_1 to be the annotation Q' with $q'_i = q_{(i,j)}$. Sometimes we omit Γ_1 and just write $\pi_j(Q)$ if the meaning follows from the context.

Type Rules. Figure 3 contains the annotated type rules for arithmetic operations, array operations, the undefined expression, variables, and function application. The rules T:Var and T:App for variables and function application are similar to the corresponding rules in previous work [10].

In the rule T:Undef, we only require that the constant potential M^{undef} is available. In contrast to the other rules we do not relate the initial potential Q

$$\frac{Q = Q' + M^{\mathsf{var}}}{\Sigma; x{:}A; Q \overset{M}{\vdash} x : (A, Q')}\ (\text{T:Var}) \qquad \frac{P{+}M^{\mathsf{app}} = Q \qquad (A,P) \to (A',Q') \in \Sigma(f)}{\Sigma; x{:}A; Q \overset{M}{\vdash} f(x) : (A', Q')}\ (\text{T:App})$$

$$\frac{Q = \boxplus(Q') + M^{\mathsf{add}}}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} x_1 + x_2 : (\mathrm{nat}, Q')}\ (\text{T:Add})$$

$$\frac{Q' + M^{\mathsf{sub}} = \boxplus(\pi_0^{x_1:\mathrm{nat}}(Q))}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{minus}(x_1, x_2) : (\mathrm{nat} * \mathrm{nat}, Q')}\ (\text{T:Sub})$$

$$\frac{q_0 = M^{\mathsf{nat}} + \sum_{i \geqslant 0} q'_i \binom{n}{i}}{\Sigma; \cdot; Q \overset{M}{\vdash} n : (\mathrm{nat}, Q')}\ (\text{T:Nat}) \qquad \frac{\begin{array}{c} Q{=}M^{\mathsf{sub}}{+}P{+}R \qquad P'{=}\boxplus(P) \qquad R'{=}\lhd^n(R) \\ q'_{(i,0)} = r'_i + p'_{(i,0)} \qquad q'_{(i,j)} = p'_{(i,j)} \text{ if } j > 0 \end{array}}{\Sigma; x{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{minus}(x, n) : (\mathrm{nat} * \mathrm{nat}, Q')}\ (\text{T:SubC})$$

$$\frac{q_0 = M^{\mathsf{undef}}}{\Sigma; \cdot; Q \overset{M}{\vdash} \mathsf{undefined} : (B, Q')}\ (\text{T:Undef}) \qquad \frac{Q = \boxdot(Q') + M^{\mathsf{mult}}}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} x_1{*}x_2{:}(\mathrm{nat}, Q')}\ (\text{T:Mult})$$

$$\frac{R + M^{\mathsf{dif}} = \boxplus(\pi_0^{x_1:\mathrm{nat}}(Q)) \qquad \forall i \in \mathbb{N} : \pi_i(R) = \boxdot(\pi_i(Q'))}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{divmod}(x_1, x_2) : ((\mathrm{nat} * \mathrm{nat}) * \mathrm{nat}, Q')}\ (\text{T:Div})$$

$$\frac{\forall\, i{>}1 : q_{(i,0)} = q'_i \qquad q_{(0,0)} = q'_0 + M^{\mathsf{Amake}} \qquad q_{(1,0)} = q'_1 + M^{\mathsf{AmakeL}}}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}A; Q \overset{M}{\vdash} \mathsf{A.make}(x_1, x_2) : (A\ \mathrm{array}, Q')}\ (\text{T:AMake})$$

$$\frac{q_0 = q'_0 + M^{\mathsf{Aget}}}{\Sigma; x_1{:}A\,\mathrm{array}, x_2{:}\mathrm{nat}, x_3{:}A; Q \overset{M}{\vdash} \mathsf{A.set}(x_1, x_2, x_3) : (\mathrm{nat}, Q')}\ (\text{T:ASet})$$

$$\frac{\forall\, i{\neq}0 : q'_i = 0 \qquad q_0 = q'_0 + M^{\mathsf{Aset}}}{\Sigma; x_1{:}A\,\mathrm{array}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{A.get}(x_1, x_2) : (A, Q')}\ (\text{T:AGet})$$

$$\frac{Q = Q' + M^{\mathsf{Alen}}}{\Sigma; x : A\,\mathrm{array}; Q \overset{M}{\vdash} \mathsf{A.length}(x) : (\mathrm{nat}, Q')}\ (\text{T:ALen})$$

Fig. 3. Annotated type rules for arithmetic and array operations.

with the resulting potential Q'. Intuitively, this is sound because the program is aborted when evaluating the expression $\mathsf{undefined}$. A consequence of the rule T:Undef is that we can type the expression $\mathsf{let}\, x = \mathsf{undefined}\, \mathsf{in}\, e$ with constant initial potential M^{undef} regardless of the resource cost of the expression e.

The rule T:Nat shows how to transfer constant potential to polynomial potential of a non-negative integer constant n. Since n is statically available, we simply compute the coefficients $\binom{n}{i}$ for the linear constraint system.

In the rule T:Add, we use the convolution operation $\boxplus(\cdot)$ that we describe in Section 4. The potential defined by the annotation $\boxplus(Q')$ for the context $x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}$ is equal to the potential Q' of the result.

Subtraction is handled by the rules T:Sub and T:SubC. To be able to conserve all the available potential, we have to ensure that subtraction is the inverse operation to addition. To this end, we abort the program if $x_2 > x_1$ and otherwise return the pair $(n, m) = (x_2, x_1 - x_2)$. This enables us to transfer the potential of x_1 to the pair (n, m) where $n + m = x_1$. This is inverse to the rule T:Add for addition.

In the rule T:Sub, we only use the potential of x_1 by applying the projection $\pi_0^{x_1:\mathrm{nat}}(Q)$. The potential of x_2 and the mixed potential of x_1 and x_2 can be arbitrary and is wasted by the rule. This is usually not problematic since it would just be zero anyways in most useful type derivations. By using the convolution $\boxplus(\pi_0^{x_1:\mathrm{nat}}(Q))$ we then distribute the potential of x_1 to the result of $\mathsf{minus}(x_1, x_2)$.

The rule T:SubC specializes the rule T:Sub. We can use T:SubC to simulate T:Sub but we also have the possibility to exploit the fact that we subtract a constant. This puts us in a position to use the K-times shift that we introduced in Section 4. So we split the initial potential Q into P and R. We then assign the convolution $P' = \boxplus(P)$ to the pair of unsigned integers that is returned by minus and the n-times shift $\lhd^n(R)$ to the first component of the returned pair. In fact, it would not hamper the expressivity of our system to only use the conventional subtraction $x - n$ and the n-times shift in the case of subtraction of constants.

In practice, it would be beneficial not to expose this non-standard minus function to users and instead apply a code transformation that converts the usual subtraction $\mathsf{let}\, x = x_1 - x_2 \,\mathsf{in}\, e$ into an equivalent expression $\mathsf{let}\ (x_2, x) = \mathsf{minus}(x_1, x_2)\,\mathsf{in}\, e$ that overshadows x_2 in e. In this way, it is ensured that the potential that is returned by minus can be used within e.

The rule T:Mult is similar to T:Add. We just use the multiplicative convolution $\boxdot(\cdot)$ (see Section 4) instead of the additive convolution $\boxplus(\cdot)$. The rule T:Div is inverse to T:Mult in the same way that T:Sub is inverse to T:Add. We use both, the additive and multiplicative convolution to express the fact that $n * m + r = x_1$ if $(n, m, r) = \mathsf{divmod}(x_1, x_2)$.

In the rule T:AMake, we transfer the potential of x_1 to the created array. We discard the potential of x_2 and the mixed potential of x_1 and x_2. At this point, it would in fact be not problematic to use mixed potential to assign it to the newly created elements of the array. We refrain from doing so solely because of the complexity that would be introduced by tracking the potential in the functions $\mathsf{A.get}$ and $\mathsf{A.set}$. Another interesting aspect of T:AMake is that we have a constant cost that we deduce from the constant coefficient as usual, as well as a linear cost that we deduce from the linear coefficient. This is represented by the constraints $q_{(0,0)} = q'_0 + M^{\mathsf{Amake}}$ and $q_{(1,0)} = q'_1 + M^{\mathsf{AmakeL}}$, respectively.

For convenience, the operation $\mathsf{A.set}$ returns 0 in this paper. In RAML, $\mathsf{A.set}$ has however the return type unit. This makes no difference for the typing rule T:ASet in which we simply pay for the cost of the operation and discard the potential that is assigned to the arguments. Since the return value is 0, we do not need require that the non-constant annotations of Q' are zero.

In the rule T:AGet, we again discard the potential of the arguments and also require that the non-linear coefficients of the annotation of the result are zero. In the rule T:ALen, we simply assign the potential of the array in the argument to the resulting integer.

Soundness. An annotated type judgment for an expression e establishes a bound on the resource cost of all evaluations of e in a well-formed environment; regardless of whether the evaluation terminates, diverges, or fails.

Additionally, the soundness theorem states a stronger property for terminating evaluations. If an expression e evaluates to a value v in a well-formed environment then the difference between initial and final potential is an upper bound on the resource usage of the evaluation.

Theorem 1 (Soundness). *Let $H \vDash V{:}\Gamma$ and $\Sigma; \Gamma; Q \overset{M}{\vdash} e{:}(B, Q')$.*

1. *If $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (p, p')$ then we have $p \leqslant \Phi_{V,H}(\Gamma; Q)$ and $p - p' \leqslant \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(\ell{:}(B, Q'))$.*
2. *If $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (p, p')$ then $p \leqslant \Phi_{V,H}(\Gamma; Q)$.*

Theorem 1 is proved by a nested induction on the derivation of the evaluation judgment and the type judgment $\Gamma; Q \vdash e{:}(B, Q')$. 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.

The proof of most rules is similar to the proof of the rules for multivariate resource analysis for sequential programs [13]. The novel type rules are mainly proved by the Lemmas 1, 2, and 3. We deal with the mutable heap by requiring that array elements do not influence the potential of an array. As a result, we can prove the following lemma.

Lemma 4. *If $H \vDash V{:}\Gamma$, $\Sigma; \Gamma; Q \overset{M}{\vdash} e : (B, Q')$ and $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (p, p')$ then $\Phi_{V,H}(\Gamma; Q) = \Phi_{V,H'}(\Gamma; Q)$.*

If the metric M is simple (all constants are 1) then it follows from Theorem 1 that the bounds on the resource usuage also prove the termination of programs.

Corollary 1. *Let M be a simple metric. If $H \vDash V{:}\Gamma$ and $\Sigma; \Gamma; Q \overset{M}{\vdash} e{:}(A, Q')$ then there are $w \in \mathbb{N}$ and $d \leqslant \Phi_{V,H}(\Gamma; Q)$ such that $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ for some ℓ and H'.*

Type Inference. In principle, type inference consists of four steps. First, we perform a classic type inference for the simple types such as nat array. Second, we fix a maximal degree of the bounds and annotate all types in the derivation of the simple types with variables that correspond to type annotations for resource polynomials of that degree. Third, we generate a set of linear inequalities, which express the relationships between the added annotation variables as specified by the type rules. Forth, we solve the inequalities with an LP solver such as CLP. A solution of the linear program corresponds to a type derivation in which the variables in the type annotations are instantiated according to the solution.

In practice, the type inference is slightly more complex. Most importantly, we have to deal with resource-polymorphic recursion in many examples. This means that we need a type annotation in the recursive call that differs from the annotation in the argument and result types of the function. To infer such types we successively infer type annotations of higher and higher degree. Details can be found in previous work [17]. Moreover, we have to use algorithmic versions of the type rules in the inference in which the non-syntax-directed rules are integrated into the syntax-directed ones [13]. Finally, we use several optimizations to reduce the number of generated constraints.

An concrete example of a type derivation can be found in previous work [13].

6 Experimental Evaluation

We have implemented our analysis system in Resource Aware ML (RAML) [12, 14] and tested the new analysis on multiple classic examples algorithms. In this section we describe the results of our experiments with the evaluation-step metric that counts the number of steps of an evaluation in the operational semantics.

Table 1 contains a compilation of analyzed functions together with their simple types, the computed bounds, the run times of the analysis, and the number of generated linear constraints. We write Mat for the type (Arr(Arr(int)),nat,nat). The dimensions of the matrices are needed since array elements do not carry potential. The variables in the computed bounds correspond to the sizes of different parts of the input. The naming convention is that we use the order n, m, x, y, z, u of the variables to name the sizes in a depth-first way: n is the size of the first argument, m is the maximal size of the elements of the first argument, x is the size of the second argument, etc. The experiments were performed on an iMac with a 3.4 GHz Intel Core i7 and 8 GB memory.

All but one of the reported bounds are asymptotically tight (gcdFast is actually $O(\log m)$). We also measured the evaluation cost of the functions for several inputs in the RAML interpreter. Our experiments indicate that all constant factors in the bounds for the functions dyadAllM and mmultAll are optimal. The bounds for the other functions seem to be off by ca. $2\% - 20\%$. However, it is sometimes not straightforward to find worst-case inputs. The full version of this article [15] contains plots for the functions dijkstra, quicksort, dyadAllM, and mmultAll that compare the measured evaluation cost for inputs of different sizes with the inferred bounds.

The function dijkstra is an implementation of Dijkstra's single-source shortest-path algorithm which uses a simple priority queue; gcdFast is an implementation of the Euclidean algorithm using modulo; pascal(n) computes the first $n{+}1$ lines of Pascal's triangle; quicksort is an implementation of Hoare's in-place quick sort for arrays; and mmultAll takes a matrix (an accumulator) and a list of matrices, and multiplies all matrices in the list with the accumulator.

The last three examples are composed functions that highlight interesting capabilities of the analysis. The function blocksort(a, n) takes an array a of length m and divides it into n/m blocks (and a last block containing the remainder) using the build-in function divmod, and sorts all blocks in-place with quicksort. The function dyadAllM(n) computes a matrix of size $(i^2{+}9i{+}28) \times (ij{+}6j)$ for every pair of numbers i, j such that $1 \leqslant j \leqslant i \leqslant n$ (the polynomials are just

Function / Type	Computed Bound	Time	#Constr.
dijkstra : (Arr(Arr(int)),nat) → Arr(int)	$79.5n^2 + 31.5n + 38$	0.1 s	2178
gcdFast : (nat,nat) → nat	$12m + 7$	0.1 s	105
pascal : nat → Arr(Arr(int))	$19n^2 + 95n + 30$	0.4 s	998
quicksort : (Arr(int),nat,nat) → unit	$12.25x^2 + 52.75x + 3$	0.7 s	2080
blocksort : (Arr(int),nat) → unit	$12.25n^2 + 90.25n + 18$	0.4 s	27795
mmultAll : (L(Mat),Mat) → Mat	$18nuyx + 31nuy + 38nu + 38n + 3$	5.6 s	184270
dyadAllM : nat → unit	$1.\bar{6}n^6 + 334.8n^4 + 1485.1n^3 + 37n^5 + 2963.5n^2 + 1789.92n + 3$	3.9 s	130236
mmultFlatSort : (Mat,Mat) → Arr(int)	$12.25u^2m^2 + 18umz + 28u + 127.25um + 49m + 66$	5.9 s	167603

Table 1. Compilation of RAML Experiments.

a random choice). Finally, the function mmultFlatSort takes two matrices and multiplies them to get a matrix of dimension $m \times u$. It then flattens the matrix into an array of length mu and sorts this array with quicksort.

We did not perform an experimental comparison with abstract interpretation–based resource analysis systems. Many systems that are described in the literature are not publicly available. The COSTA system [3, 4] is an exception but it is not straightforward to translate our examples to Java code that COSTA can handle. We know that the COSTA system can compute bounds for the Euclidean algorithm (when using an extension [8]), quick sort, and Pascal's triangle. The advantages of our method are the compositionality that is needed for the analysis of compound functions such as dyadAllM and mmultFlatSort, as well as for bounds that depend on integers as well as on sizes of data structures such as dijkstra (priority queue) and mmultAll.

7 Conclusion

We have presented a novel type-based amortized resource analysis for programs with arrays and unsigned integers. We have implemented the analysis in Resource Aware ML and our experiments show that the analysis works efficiently for many example programs. Moreover, we have demonstrated that the analysis has benefits in comparison to abstract interpretation–based approaches for programs with function composition and non-linear size changes.

While the developed analysis system for RAML is useful and interesting in its own right, we view this work mainly as an important step towards the application of amortized resource analysis to C-like programs. The developed rules for arithmetic expression can be reused when moving to a different language. Our next step is to develop an analysis system that applies the ideas of this work to an imperative language with while-loops, integers, and arrays.

Acknowledgments. This research is based on work supported in part by NSF grants 1319671 and 1065451, and DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
2. Zuleger, F., Sinn, M., Gulwani, S., Veith, H.: Bound Analysis of Imperative Programs with the Size-change Abstraction. In: 18th Int. Static Analysis Symp. (SAS'11). (2011) 280–297
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theor. Comput. Sci. **413**(1) (2012) 142 – 159
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Automatic Inference of Resource Consumption Bounds. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'18). (2012) 1–11
5. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: 5th ACM Symp. on Principles Prog. Langs. (POPL'78). (1978) 84–96
6. Gulavani, B.S., Gulwani, S.: 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). (2008) 370–384
7. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static Analysis in Disjunctive Numerical Domains. In: 13th Int. Static Analysis Symp. (SAS'06). (2006) 3–17
8. Alonso-Blas, D.E., Arenas, P., Genaim, S.: Handling Non-linear Operations in the Value Analysis of COSTA. Electr. Notes Theor. Comput. Sci. **279**(1) (2011) 3–17
9. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). (2003) 185–197
10. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th ACM Symp. on Principles of Prog. Langs. (POPL'11). (2011)
11. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010)
12. Aehlig, K., Hofmann, M., Hoffmann, J.: RAML Web Site. `http://raml.tcs.ifi.lmu.de` (2010-2013)
13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. (2012)
14. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource Aware ML. In: 24rd Int. Conf. on Computer Aided Verification (CAV'12). (2012)
15. Hoffmann, J., Shao, Z.: Type-Based Amortized Resource Analysis with Integers and Arrays. `http://cs.yale.edu/homes/hoffmann/papers/aa_imp2013TR.pdf` (2013) Full Version.
16. Riordan, J., Stein, P.R.: Arrangements on Chessboards. Journal of Combinatorial Theory, Series A **12**(1) (1972)
17. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Prog. Langs. and Systems - 8th Asian Symposium (APLAS'10). (2010)