Under consideration for publication in J. Functional Programming

Type-Based Amortized Resource Analysis with Integers and Arrays

JAN HOFFMANN and ZHONG SHAO
Yale University
(*e-mail:* {jan.hoffmann,zhong.shao}@yale.edu)

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 and 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 $f(g(x))$.

1 Introduction

The quantitative performance characteristics of a program are among the most important aspects that determine whether the program is useful in practice. Even the most elegant solution to a programming problem is useless if its clock-cycle or memory consumption exceeds the available resources. While resource consumption is relevant for every program, it is particularly critical in embedded and real-time systems where resources are often extremely limited. If such systems operate in a safety-critical context then formal verification of a bound on the worst-case resource behavior is an effective method to increase the trust in the system.

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, that is, bounds on the values of variables. The obtained *size-change information* forms the basis of the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation (Gulwani *et al.*, 2009), ranking functions (Alias *et al.*, 2010; Albert *et al.*, 2011; Brockschmidt *et al.*, 2014; Sinn *et al.*, 2014), recurrence relations (Albert *et al.*, 2012b; Albert *et al.*, 2012a), and abstract interpretation itself (Gulavani & Gulwani, 2008; Zuleger *et al.*, 2011).

For reasons of efficiency, many abstract interpretation–based resource-analysis systems rely on abstract domains that enable the inference of invariants through linear constraint solving (Cousot & Halbwachs, 1978; Miné, 2004). 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. However, a linear abstract domain can be used to derive non-linear invariants using domain lifting operations (Gulavani & Gulwani, 2008). Another possibility is to use disjunctive abstract domains to generate non-linear invariants (Sankaranarayanan *et al.*, 2006). This technique has been experimentally implemented in the COSTA analysis system (Alonso-Blas *et al.*, 2011). However, 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. It is based on type-based amortized resource analysis (Hofmann & Jost, 2003; Hoffmann *et al.*, 2011) and tracking of size-changes does not require abstract interpretation. 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 (Hoffmann & Hofmann, 2010b; Hoffmann *et al.*, 2011). 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. Connections with the aforementioned other approaches are discussed in (Alonso-Blas & Genaim, 2012). Size-changes of variables can also be tracked using ranking functions and local size bounds derived by SMT solving (Brockschmidt *et al.*, 2014). However, type-based amortized resource analysis reduces inference of non-linear bounds and tracking of non-linear size changes to linear programming.

Our ultimate goal is to transfer the advantages of amortized resource analysis to imperative (C-like) programs. We have already shown that techniques based on amortized resource analysis can be integrated into a program logic and the verified C compiler CompCert (Leroy, 2006) to derive stack bounds on compiled x86 code (Carbonneaux *et al.*, 2014). Moreover, we have developed a version of amortized resource analysis for C code that automatically derives *linear* bounds that are functions of signed integers (Carbonneaux *et al.*, 2015).

The next important step is to extend our analysis for C programs to polynomial bounds. It is beneficial to study this extension carefully in a functional setting before moving to C programs and signed integers where things are more involved. Therefore 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) (Aehlig *et al.*, 2010-2013) which features more data types such as lists and binary trees. Our experiments (see Section 7) 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 inner function is non-linear in its arguments. 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 (Aehlig *et al.*, 2010-2013).

Technically, we treat unsigned integers like unary lists in multivariate amortized analysis (Hoffmann *et al.*, 2011). 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. This includes a generalized additive shift (see (Hoffmann & Hofmann, 2010b)) for subtraction with a constant, and two convolutions for addition and multiplication (see Section 5). 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.

Arrays are manipulated with the standard operations A.make, A.get, A.set, and A.length. 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. There are three main reasons for this. First, in many programs data stored in arrays is not used in the control flow. Second, nested arrays often have fixed dimensions like an $n \times m$ matrix. So the iteration cost does not only depend on the sizes of the inner arrays but also the dimensions like n and m. Third, it is often possible to replace arrays with lists if resource usage depends on the size of the elements.

The main difficulty with inner potential for arrays is the non-linear access of the array elements. If we obtained potential from an array access $\mathsf{A.get}(\mathsf{a},\mathsf{i})$ then we would need to keep track of the array elements from which we had obtained potential already and the elements which still carry potential for future use. It would then be necessary to abstract common iteration patterns and identify these patterns in the program before the analysis. Automatically identifying such patterns and integrating them into the analysis is currently an open problem.

In the implementation, we also have *signed* integers and a successful analysis proves that the resource usage of a program cannot depend on the values of signed integers. If the resource usage depends on the value of a signed integer in a non-trivial way then the analysis terminates without deriving a bound.

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 (Hofmann & Jost, 2003; Hoffmann & Hofmann, 2010b; Hoffmann *et al.*, 2011), the soundness proof (Hoffmann *et al.*, 2012a), and Resource Aware ML (Aehlig *et al.*, 2010-2013; Hoffmann *et al.*, 2012b).

This article is the extended journal version of a conference article with the same title that appeared earlier (Hoffmann & Shao, 2014). The changes with respect to the conference version include additional lemmas and theorems, proofs, the complete sets of inference rules, and additional experimental results.

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 (Hofmann & Jost, 2003; Hoffmann *et al.*, 2011) 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 (Hoffmann & Hofmann, 2010b; Hoffmann *et al.*, 2011; Hoffmann *et al.*, 2012a; Hoffmann *et al.*, 2012b), 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 (Hoffmann *et al.*, 2012a). At the same time, multivariate resource polynomials can be incorporated into type systems so that type inference can be efficiently reduced to LP solving (Hoffmann *et al.*, 2012a).

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 $\text{Ind}(A) = \{(0,0),(1,0),(2,0),(1,1),(0,1),(0,2)\}$. A family $Q = (q_i)_{i \in \text{Ind}(A)}$ denotes the resource polynomial that maps two lists n and m to the number $\sum_{(i,j)\in\text{Ind}(A)} = q_{(i,j)} \binom{|n|}{i}\binom{|m|}{j}$. 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. The annotation Q of the arguments corresponds then to the potential function $|n||m|+2|n|+1$ and the annotation P of the result corresponds to the potential function 0.

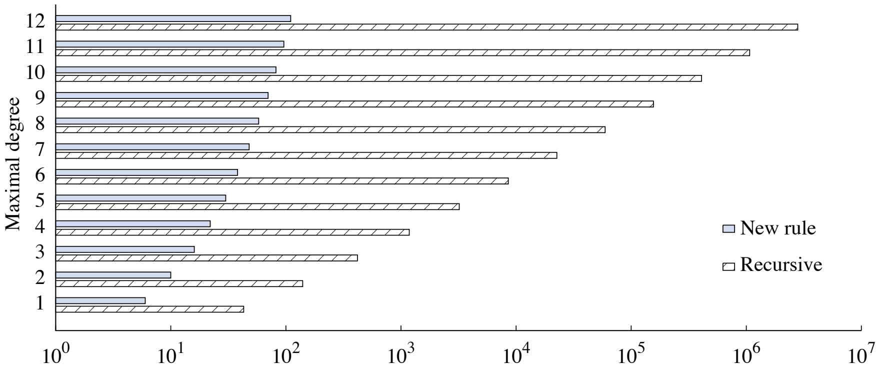

Fig. 1. Number of constraints generated by RAML for the program $\mathsf{a} * \mathsf{b}$ as a function of the maximal degree. The solid bars show the number of constraints generated using the novel type rule for multiplication. The striped bars show the number of constrained generated using an recursive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

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. This latter typing is needed to pass potential to the result of $\mathsf{mult}(\mathsf{n},\mathsf{m})$. Here, Q corresponds to the potential function $2|n||m| + 2|n| + 2$ and P corresponds to $t + 1$ where $t = |\mathsf{mult}(\mathsf{n},\mathsf{m})|$ is the length of the result of the multiplication.

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 (Hoffmann & Hofmann, 2010b; Hoffmann *et al.*, 2012a) 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 (Hoffmann & Hofmann, 2010b; Hoffmann *et al.*, 2012a) (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 gcd algorithm cannot be analyzed with the usual, recursive definition of mod but can be analyzed with our new 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 $\mathsf{n} > \mathsf{C}$ then ... $\mathsf{recCall}(\mathsf{n}-\mathsf{C})$ 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) (Hoffmann *et al.*, 2012b; Aehlig *et al.*, 2010-2013) 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 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. The sharing expression is similar to a let binding and 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.

Simple Types and Programs. Data types A,B and function types F are defined as follows.

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

Let $\mathscr{A}$ be the set of data types and let $\mathscr{F}$ be the set of function types. A signature $\Sigma : \text{FID} \rightharpoonup \mathscr{F}$ is a partial finite mapping from function identifiers to function types. A context is a partial finite mapping $\Gamma : \mathit{Var} \rightharpoonup \mathscr{A}$ 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. Basically, the rules are obtained by erasing the potential annotations of the syntax-directed rules in Section 5.

A *(well-typed) program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \text{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash e_f{:}B$ if $\Sigma(f) = A \to B$.

Cost Semantics. In the following, we define an operational big-step semantics for our subset of RAML. The semantics is standard except that it defines a cost of an evaluation. This cost depends on a resource metric that assigns a cost to each atomic operation.

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.

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

$$\frac{}{V,H \vdash^{M} e \Downarrow \circ \mid 0}\;(\text{E:Zero}) \qquad \frac{[y_f \mapsto V(x)],H \vdash^{M} e_f \Downarrow \rho \mid (q,q')}{V,H \vdash^{M} f(x) \Downarrow \rho \mid M^{\mathsf{app}}\cdot(q,q')}\;(\text{E:App})$$

$$\frac{V(x) = \ell}{V,H \vdash^{M} x \Downarrow (\ell,H) \mid M^{\mathsf{var}}}\;(\text{E:Var}) \qquad \frac{H' = H,\ell \mapsto (V(x_1),V(x_2))}{V,H \vdash^{M} (x_1,x_2) \Downarrow (\ell,H') \mid M^{\mathsf{pair}}}\;(\text{E:Pair})$$

$$\frac{H(V(x)) = (\ell_1,\ell_2) \qquad V[x_1 \mapsto \ell_1, x_2 \mapsto \ell_2],H \vdash^{M} e \Downarrow \rho \mid (q,q')}{V,H \vdash^{M} \mathsf{match}\;x\;\mathsf{with}\;(x_1,x_2) \Rightarrow e \Downarrow \rho \mid M^{\mathsf{matP}}\cdot(q,q')}\;(\text{E:MatP})$$

$$\frac{V(x) = \ell \qquad V[x_1 \mapsto \ell, x_2 \mapsto \ell],H \vdash^{M} e \Downarrow \rho \mid (q,q')}{V,H \vdash^{M} \mathsf{share}\;x\;\mathsf{as}\;(x_1,x_2)\;\mathsf{in}\;e \Downarrow \rho \mid M^{\mathsf{share}}\cdot(q,q')}\;(\text{E:Share})$$

$$\frac{}{V,H \vdash^{M} \mathsf{undefined} \Downarrow \circ \mid M^{\mathsf{undef}}}\;(\text{E:Undef}) \qquad \frac{V,H \vdash^{M} e_1 \Downarrow \circ \mid (q,q')}{V,H \vdash^{M} \mathsf{let}\;x = e_1\;\mathsf{in}\;e_2 \Downarrow \circ \mid M^{\mathsf{let1}}\cdot(q,q')}\;(\text{E:Let1})$$

$$\frac{V,H \vdash^{M} e_1 \Downarrow (\ell,H') \mid (q,q') \qquad V[x \mapsto \ell],H' \vdash^{M} e_2 \Downarrow \rho \mid (p,p')}{V,H \vdash^{M} \mathsf{let}\;x = e_1\;\mathsf{in}\;e_2 \Downarrow \rho \mid M^{\mathsf{let1}}\cdot(q,q')\cdot M^{\mathsf{let2}}\cdot(p,p')}\;(\text{E:Let2})$$

$$\frac{n \in \mathbb{N} \qquad H' = H,\ell \mapsto n}{V,H \vdash^{M} n \Downarrow (\ell,H') \mid M^{\mathsf{nat}}}\;(\text{E:Nat}) \qquad \frac{n = H(V(x_1)) + H(V(x_2)) \qquad H' = H,\ell \mapsto n}{V,H \vdash^{M} x_1 + x_2 \Downarrow (\ell,H') \mid M^{\mathsf{add}}}\;(\text{E:Add})$$

$$\frac{n = H(V(x_1)) - H(V(x_2)) \qquad n \geq 0 \qquad H' = H,\ell \mapsto (V(x_2),\ell'),\ell' \mapsto n}{V,H \vdash^{M} \mathsf{minus}(x_1,x_2) \Downarrow (\ell,H') \mid M^{\mathsf{sub}}}\;(\text{E:Sub})$$

$$\frac{n' = H(V(x)) - n \qquad H' = H,\ell \mapsto (\ell_1,\ell_2),\ell_1 \mapsto n,\ell_2 \mapsto n'}{V,H \vdash^{M} \mathsf{minus}(x,n) \Downarrow (\ell,H') \mid M^{\mathsf{sub}}}\;(\text{E:SubC})$$

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

$$\frac{\begin{array}{c} n_1 = H(V(x_1)) \qquad n_2 = H(V(x_2)) \\ H' = H,\ell \mapsto (\ell',\ell_3),\ell' \mapsto (V(x_2),\ell_2),\ell_2 \mapsto (n_1 \div n_2),\ell_3 \mapsto (n_1 \bmod n_2) \end{array}}{V,H \vdash^{M} \mathsf{divmod}(x_1,x_2) \Downarrow (\ell,H') \mid M^{\mathsf{div}}}\;(\text{E:Div})$$

$$\frac{H(V(x)) = 0 \qquad V,H \vdash^{M} e_1 \Downarrow \rho \mid (q,q')}{V,H \vdash^{M} \mathsf{match}\;x\;\mathsf{with}\;\langle 0 \Rightarrow e_1 \mid \mathsf{S}(y) \Rightarrow e_2 \rangle \Downarrow \rho \mid M^{\mathsf{matZ}}\cdot(q,q')}\;(\text{E:MatN1})$$

$$\frac{H(V(x)) = n+1 \qquad V[y \mapsto \ell],H,\ell \mapsto n \vdash^{M} e_2 \Downarrow \rho \mid (q,q')}{V,H \vdash^{M} \mathsf{match}\;x\;\mathsf{with}\;\langle 0 \Rightarrow e_1 \mid \mathsf{S}(y) \Rightarrow e_2 \rangle \Downarrow \rho \mid M^{\mathsf{matS}}\cdot(q,q')}\;(\text{E:MatN2})$$

Fig. 2. Rules of the operational big-step semantics (part 1).

$$\frac{H(V(x_1)) = n \qquad \forall i : \sigma(i) = V(x_2) \qquad H' = H, \ell' \mapsto (\sigma, n)}{V,H \overset{M}{\vdash} \mathsf{A.make}(x_1,x_2) \Downarrow (\ell',H') \mid (n{\cdot}M^{\mathsf{AmakeL}} + M^{\mathsf{Amake}}, 0)} \;(\text{E:AMAKE})$$

$$\frac{\begin{array}{c} V(x_1) = \ell_1 \qquad H(\ell_1) = (\sigma,n) \\ H(V(x_2)) = i \qquad 0 \leq i < n \qquad H' = H[\ell_1 \mapsto (\sigma[i \mapsto V(x_3)], n), \ell_2 \mapsto 0] \end{array}}{V,H \overset{M}{\vdash} \mathsf{A.set}(x_1,x_2,x_3) \Downarrow (\ell_2,H') \mid M^{\mathsf{Aset}}} \;(\text{E:ASET})$$

$$\frac{H(V(x_1)) = (\sigma,n) \qquad H(V(x_2)) \geq n}{V,H \overset{M}{\vdash} \mathsf{A.set}(x_1,x_2,x_3) \Downarrow \circ \mid M^{\mathsf{Afail}}} \;(\text{E:ASFAIL})$$

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

$$\frac{H(V(x_1)) = (\sigma,n) \qquad H(V(x_2)) \geq n}{V,H \overset{M}{\vdash} \mathsf{A.get}(x_1,x_2) \Downarrow \circ \mid M^{\mathsf{Afail}}} \;(\text{E:AGFAIL}) \qquad \frac{H(V(x)) = (\sigma,n) \qquad H' = H, \ell \mapsto n}{V,H \overset{M}{\vdash} \mathsf{A.length}(x) \Downarrow (\ell,H') \mid M^{\mathsf{Alen}}} \;(\text{E:ALEN})$$

Fig. 3. Rules of the operational big-step semantics (part 2).

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 in Figure 2 and Figure 3 are formulated with respect to a resource metric M. 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 (see below), 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'. The location ℓ then contains the value to which the expression has evaluated. 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*). In this paper, we combine erroneous behavior with non-terminating behavior since we are only interested in the resource consumption. In other applications it might be more useful to introduce a separate error value $\bot$. This can be done without problems.

The evaluation judgment has the general form

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

An intuition for the judgement $V,H \overset{M}{\vdash} e \Downarrow \circ \mid (q,q')$ is that there is a partial evaluation of e that runs without failure, needs q resources (high watermark), has momentarily q' resources available, and has not yet reached a value. This is similar to a small-step judgement.

$$\frac{\begin{array}{c}H(\ell)=(\sigma,n)\\ \mathrm{dom}(\sigma)=\mathrm{dom}(\alpha)=\{0,\ldots,n-1\} \qquad \forall\, 0{\le}i{<}n : H\vDash \sigma(i)\mapsto a_i \text{ and } \alpha(i)=a_i\end{array}}{H\vDash \ell\mapsto(\alpha,n) : A \text{ array}}\ \text{(V:ARRAY)}$$

$$\frac{H(\ell)=n \qquad n\in\mathbb{N}}{H\vDash \ell\mapsto n : \text{nat}}\ \text{(V:NAT)} \qquad \frac{H(\ell)=(\ell_1,\ell_2) \qquad H\vDash \ell_1\mapsto a_1 : A_1 \qquad H\vDash \ell_2\mapsto a_2 : A_2}{H\vDash \ell\mapsto(a_1,a_2) : (A_1,A_2)}\ \text{(V:PAIR)}$$

Fig. 4. Relating heap cells to semantic values.

The cost semantics is non-deterministic. The idea is to use the rule E:ZERO to approximate (non-existing) infinite evaluation trees that correspond to diverging computations with an infinite number of finite trees. The rules E:ZERO and E:LET1 can be used in combination to obtain a snapshot of the resource usage during a diverging or converging computation.

A resource metric $M : K \to \mathbb{Q}$ defines the resource consumption of each evaluation step of the big-step semantics. Here, K is a finite set of constant symbols. We define

$$\begin{aligned}K =\{&\mathsf{nat},\mathsf{var},\mathsf{app},\mathsf{matchL},\mathsf{undef},\mathsf{pair},\mathsf{matP},\mathsf{let1},\mathsf{let2},\mathsf{nat},\mathsf{add},\mathsf{mult},\mathsf{sub},\mathsf{div}\\ &\mathsf{matS},\mathsf{matZ},\mathsf{minus},\mathsf{divmod},\mathsf{Amake},\mathsf{Aset},\mathsf{Aget},\mathsf{Alength},\mathsf{Afail}\}\ .\end{aligned}$$

We write M^k for $M(k)$.

We view the pairs (q,q') in the evaluation judgments as elements of a monoid $\mathscr{Q}=(\mathbb{Q}_0^+\times\mathbb{Q}_0^+,\cdot)$. The neutral element is $(0,0)$ which means that resources are neither needed nor refunded. 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

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

If resources are never restored (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 $\mathscr{Q}$ as follows: $q\ge 0$ 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.

Proposition 3.1
Let $(q,q')=(r,r')\cdot(s,s')$.

1. $q\ge r$ and $q-q'=r-r'+s-s'$
2. If $(p,p')=(\bar{r},r')\cdot(s,s')$ and $\bar{r}\ge r$ then $p\ge q$ and $p'=q'$
3. If $(p,p')=(r,r')\cdot(\bar{s},s')$ and $\bar{s}\ge s$ then $p\ge q$ and $p'\le q'$
4. $(r,r')\cdot((s,s')\cdot(t,t'))=((r,r')\cdot(s,s'))\cdot(t,t')$

In the semantic rules we use the notation $H'=H,\ell{\mapsto}v$ to indicate that $\ell\notin\mathrm{dom}(H)$, $\mathrm{dom}(H')=\mathrm{dom}(H)\cup\{\ell\}$, $H'(\ell)=v$, and $H'(x)=H(x)$ for all $x\neq\ell$.

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

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

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 Figure 4.

If we fix a simple type A and a heap H then there exists at most one semantic value a such that $H \vDash \ell \mapsto a : A$.

Proposition 3.2
Let H be a heap, $\ell \in Loc$, and let A be a simple type. If $H \vDash \ell \mapsto a : A$ and $H \vDash \ell \mapsto a' : A$ then $a = a'$.

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 $H \vDash V : \Gamma$.

Theorem 3.1 shows that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment. A proof of a similar theorem can be found in a previous article (Hoffmann *et al.*, 2012a).

Theorem 3.1
If $\Gamma \vdash e : B$, $H \vDash V : \Gamma$ and $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (q, q')$ then $H' \vDash V : \Gamma$ and $H' \vDash \ell : B$.

4 Resource Polynomials and Annotated Types

Compared with multivariate amortized resource analysis for nested inductive data types (Hoffmann *et al.*, 2012a), 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. To emphasize that these potential functions are a special case of general multivariate resource polynomials we nevertheless use the terminology that has been developed for the general case (Hoffmann *et al.*, 2012a). In this way, it is straightforward to see that the present development could be readily implemented in Resource Aware ML.

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.,

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

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

Example 4.1
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 $\mathrm{nat} * \mathrm{nat}$.

Names for Base Polynomials. To assign a unique name to each base polynomial, we define the *index set* $\mathrm{Ind}(A)$ to denote resource polynomials for a given data type A. Basically, $\mathrm{Ind}(A)$ is the meaning of A when we identify arrays with their lengths.

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

The *degree* $\deg(i)$ of an index $i \in \mathrm{Ind}(A)$ is defined as follows.

$$\begin{aligned} \deg(k) &= k && \text{if } k \in \mathbb{N} \\ \deg(i_1, i_2) &= \deg(i_1) + \deg(i_2) \end{aligned}$$

Let $\mathrm{Ind}_k(A) = \{i \in \mathrm{Ind}(A) \mid \deg(i) \le k\}$. The indexes $i \in \mathrm{Ind}_k(A)$ are an enumeration of the base polynomials $p_i \in P(A)$ of degree at most k. For each $i \in \mathrm{Ind}(A)$, we define a base polynomial $p_i \in P(A)$ as follows: If $A = \mathrm{nat}$ 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 $\mathrm{Ind}(A)$ such that $p_{0_A}(a) = 1$ for all a. We identify the index $(i_1, \dots, i_n)$ with the index $(i_1, (i_2, (\cdots (i_{n-1}, i_n))))$.

Example 4.2
Our previous example $h : [\![\mathrm{nat} * \mathrm{nat}]\!] \to \mathbb{Q}_0^+$ from Example 4.1 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

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

We say Q_A is of *degree (at most) k* if $q_i = 0$ for every $i \in \mathrm{Ind}(A)$ 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 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*

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

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

Example 4.3
Consider the resource polynomial $h(n,m)$ from Example 4.1. We have $\Phi((n,m) : (\mathrm{nat} * \mathrm{nat}, Q)) = h(n,m)$ 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 $(i,j) \in \mathrm{Ind}(\mathrm{nat} * \mathrm{nat})$.

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 $\mathrm{Ind}(\Gamma)$ is defined as

$$\mathrm{Ind}(\Gamma) = \{(i_1, \ldots, i_n) \mid i_j \in \mathrm{Ind}(A_j)\} .$$

The degree of $i = (i_1, \ldots, i_n) \in \mathrm{Ind}(\Gamma)$ is defined as $\deg(i) = \deg(i_1) + \cdots + \deg(i_n)$. As for data types, we define $\mathrm{Ind}_k(\Gamma) = \{i \in \mathrm{Ind}(\Gamma) \mid \deg(i) \le k\}$. A *type annotation* Q for Γ is a family

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

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

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

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

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., $Q = (q_i)_{i \in I}$ and $Q' = (q'_i)_{i \in I}$.

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 $Q \le P + R$ if $q_i \le p_i + r_i$ for every $i \in I$.

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

We write $\Sigma; \Gamma; Q \overset{\mathsf{cf}}{\vdash} e : (A, Q')$ to refer to cost-free type judgments where cf is the cost-free metric with $\mathsf{cf}(K) = 0$ for constants K. We use it to assign potential to an extended context in the let rule.

Let Q be an annotation for a context Γ_1, Γ_2. For $j \in \mathrm{Ind}(\Gamma_2)$ 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.

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.

Let $\Gamma, y{:}\text{nat}$ be a context and let $Q = (q_i)_{i \in \text{Ind}(\Gamma, y:\text{nat})}$ be a context annotation of degree k. The *additive shift for natural numbers* $\lhd(Q)$ of Q is an annotation Q' of degree k for a context $\Gamma, x{:}\text{nat}$ that is defined as

$$\lhd(Q) = (q'_{(i,j)})_{(i,j)\in\text{Ind}(\Gamma,x:\text{nat})} \qquad \text{if} \qquad q'_{(i,j)} = q_{(i,j)} + q_{(i,j+1)} \; .$$

The additive shift for natural numbers reflects the identity

$$\sum_{0\le i\le k} q_i \binom{n+1}{i} = \sum_{0\le i\le k} (q_i{+}q_{i+1}) \binom{n}{i} \tag{1}$$

where $q_{k+1} = 0$. It is used in cases when a natural number is incremented or decremented by one. This is the case in the successor function (not presented here but implemented in RAML) or in the type rule T:MATN for pattern matching on natural numbers. This is a special case of the additive shift that has been introduced for lists and trees in previous articles (Hoffmann *et al.*, 2012a).

Example 4.4
Consider again our running example, the resource polynomial $h(m,n)$ that is given by Q where $q_{(0,0)} = 7$, $q_{(0,1)} = 2.5$, $q_{(2,3)} = 5$, $q_{(4,0)} = 8$, and $q_{(i,j)} = 0$ for all other $(i,j) \in \text{Ind}(\text{nat} * \text{nat})$. Then the additive shift $\lhd(Q)$ of Q in direction n is given by $Q' = \lhd(Q)$ where $q'_{(0,0)} = 9.5$, $q'_{(0,1)} = 2.5$, $q'_{(2,2)} = 5$, $q'_{(2,3)} = 5$, $q'_{(4,0)} = 8$, and $q'_{(i,j)} = 0$ for all other $(i,j) \in \text{Ind}(\text{nat} * \text{nat})$. It reflects the identity $h(m,n{+}1) = 7 + 2.5{\cdot}(n{+}1) + 5\binom{m}{2}\binom{n+1}{3} + 8\binom{m}{4} = 9.5 + 2.5{\cdot}n + 5\binom{m}{2}\binom{n}{2} + 5\binom{m}{2}\binom{n}{3} + 8\binom{m}{4}$.

Lemma 4.1 states the soundness of the shift operation.

Lemma 4.1
Let $\Gamma, x{:}\text{nat}; Q$ be an annotated context, $H \vDash V : \Gamma, x{:}\text{nat}$, and $H(V(x)) = n{+}1$. Let furthermore $V' = V[y{\mapsto}\ell]$ and $H' = H, \ell{\mapsto}n$. Then $H' \vDash V' : \Gamma, y{:}\text{nat}$ and $\Phi_{V,H}(\Gamma, x{:}\text{nat}; Q) = \Phi_{V',H'}(\Gamma, y{:}\text{nat}; \lhd(Q))$.

Proof
By definition we have $\Phi_{V,H}(\Gamma, x{:}\text{nat}; Q) = \sum_{(i,j)} q_{(i,j)} {\cdot} \phi_i {\cdot} \binom{n+1}{j}$ where ϕ_i is a product of base polynomials that depends on the data that is referenced by Γ. From the premises $V' = V[y \mapsto \ell]$ and $H' = H, \ell \mapsto n$, and the definition of the additive shift it follows that $\Phi_{V',H'}(\Gamma, y{:}\text{nat}; \lhd(Q)) = \sum_{(i,j)} (q_{(i,j)} + q_{(i,j+1)}) {\cdot} \phi_i {\cdot} \binom{n}{j}$. But then we use (1) (for every i) to derive

$$\begin{aligned}
\sum_{(i,j)} (q_{(i,j)} + q_{(i,j+1)}) {\cdot} \phi_i {\cdot} \binom{n}{j} \quad &= \quad \sum_i \phi_i \left(\sum_j (q_{(i,j)} + q_{(i,j+1)}) {\cdot} \binom{n}{j} \right) \\
&= \quad \sum_i \phi_i \left(\sum_j q_{(i,j)} {\cdot} \binom{n+1}{j} \right) \\
&= \quad \sum_{(i,j)} q_{(i,j)} {\cdot} \phi_i {\cdot} \binom{n+1}{j} \\
&= \quad \Phi_{V,H}(\Gamma, x{:}\text{nat}; Q) \qquad \square
\end{aligned}$$

For addition and subtraction (compare rules T:ADD and T:SUB in Figure 5) 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.

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

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

$$\binom{n_1+n_2}{k} = \sum_{i+j=k} \binom{n_1}{i}\binom{n_2}{j} \tag{2}$$

This can be viewed as an explicit representation of the type of the append function for unit lists in multivariate amortized analysis (Hoffmann *et al.*, 2012a).

Example 4.5
Consider the resource polynomial $g(n) = n + 2.2\binom{n}{2}$, that is, $q_1 = 1$, $q_2 = 2.2$, and $q_i = 0$ otherwise. Then we have $\boxplus(Q) = (q'_{(i,j)})_{(i,j)\in\mathrm{Ind}(\mathrm{nat}*\mathrm{nat})}$, where the non-zero coefficients of Q' are $q'_{(1,0)} = q'_{(0,1)} = 1$ and $q'_{(1,1)} = q'_{(0,2)} = q'_{(2,0)} = 2.2$. This reflects the identity $g(n) = m + k + 2.2(\binom{m}{2} + \binom{k}{2} + mk)$ for every m,k with $m + k = n$.

Lemma 4.2
Let Q be an annotation for type nat, $H \vDash \ell \mapsto n_1{+}n_2 : \mathrm{nat}$, and $H' \vDash \ell' \mapsto (n_1,n_2) : \mathrm{nat}*\mathrm{nat}$. Then $\Phi_H(\ell{:}(\mathrm{nat},Q)) = \Phi_{H'}(\ell'{:}(\mathrm{nat}*\mathrm{nat},\boxplus(Q)))$.

Proof
By definition we have $\Phi_H(\ell{:}(\mathrm{nat},Q)) = \sum_k q_k\binom{n_1+n_2}{k}$ and also $\Phi_{H'}(\ell'{:}(\mathrm{nat}*\mathrm{nat},\boxplus(Q))) = \sum_{(i,j)} q'_{(i,j)}\binom{n_1}{i}\binom{n_2}{j}$ for some coefficients $q'_{(i,j)} \in \mathbb{Q}_0^+$. From the definition of the convolution $\boxplus(Q)$ it follows that $q'_{(i,j)} = q_{i+j}$. Thus we can use (2) to derive the statement of the lemma.

$$\begin{aligned}
\sum_k q_k \binom{n_1+n_2}{k} &= \sum_k q_k \left(\sum_{i+j=k}\binom{n_1}{i}\binom{n_2}{j}\right) \\
&= \sum_{i+j=k} q_k \binom{n_1}{i}\binom{n_2}{j} \\
&= \sum_{(i,j)} q'_{(i,j)}\binom{n_1}{i}\binom{n_2}{j} \qquad \square
\end{aligned}$$

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.

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

Recall that $\binom{n}{m} = 0$ if $m > n$. The K-times shift is a generalization of the additive shift which is equivalent to the 1-times shift. It corresponds to the following identity that holds if

$q_j = 0$ for all $j > k$.

$$\sum_{0 \le i \le k} q_i \binom{n+K}{i} = \sum_{0 \le i \le k} \left(\sum_{j=i+\ell} q_j \binom{K}{\ell} \right) \binom{n}{i} \tag{3}$$

It can be derived from Vandermonde's convolution as follows.

$$\begin{aligned}
\sum_{0 \le i \le k} q_i \binom{n+K}{i} &= \sum_{0 \le i \le k} q_i \left(\sum_{i=j+\ell} \binom{n}{j} \binom{K}{\ell} \right) \\
&= \sum_{i=j+\ell} q_i \binom{n}{j} \binom{K}{\ell} && \text{where } 0 \le i, j, \ell \le k \\
&= \sum_{0 \le j \le k} \left(\sum_{i=j+\ell} q_i \binom{n}{j} \binom{K}{\ell} \right) && \text{where } 0 \le i, \ell \le k \\
&= \sum_{0 \le j \le k} \left(\sum_{i=j+\ell} q_i \binom{K}{\ell} \right) \binom{n}{j} && \text{where } q_i = 0 \text{ for } i > k
\end{aligned}$$

Lemma 4.3

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

Proof

By definition, $\Phi_H(\ell{:}(\mathrm{nat}, Q)) = \sum_i q_i \binom{n+K}{i}$ and $\Phi_{H'}(\ell'{:}(\mathrm{nat}, \lhd^K Q)) = \sum_i q'_i \binom{n}{i}$ for some coefficients $q'_i \in \mathbb{Q}_0^+$. From the definition of the K-times shift $\lhd^K Q$ it follows that $q'_i = \sum_{j=i+\ell} q_j \binom{K}{\ell}$. Thus we can use (3) to argue as follows.

$$\sum_i q_i \binom{n+K}{i} = \sum_i \left(\sum_{j=i+\ell} q_j \binom{K}{\ell} \right) \binom{n}{i} = \sum_i q'_i \binom{n}{i} = \Phi_{H'}(\ell'{:}(\mathrm{nat}, \lhd^K Q)) \qquad \square$$

For multiplication and division, things are more interesting. Our goal is to define an operation $\boxdot(Q)$ that defines an annotation for the arguments (x_1, x_2) : nat $*$ nat if given an annotation Q of a product $x_1 * x_2$: nat. For this purpose, we are interested in the coefficients $A(i, j, k)$ in the following identity. Note that ranges for i and j are not necessary since $A(i, j, k)$ will be 0 if $i > k$ or $j > k$.

$$\binom{nm}{k} = \sum_{i,j} A(i, j, k) \binom{n}{i} \binom{m}{j} \tag{4}$$

Fortunately, this problem has been carefully studied by Riordan and Stein (Riordan & Stein, 1972).[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.

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

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

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

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

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

Note that ranges for the indices $i,j,k \in \mathbb{N}$ are not necessary since $A(i,j,k) = 0$ if $i > k$, $j > k$, or $k > i*j$. In the implementation, we have to select some bounds to obtain a finite number of indices. We currently fix a maximal degree D and require $i+j \leq D$ and $k \leq D$.

Lemma 4.4
Let Q be an annotation for type nat, $H \vDash \ell \mapsto n_1\cdot n_2 : \mathrm{nat}$, and $H' \vDash \ell' \mapsto (n_1,n_2) : \mathrm{nat}*\mathrm{nat}$. Then $\Phi_H(\ell{:}(\mathrm{nat},Q)) = \Phi_{H'}(\ell'{:}(\mathrm{nat}*\mathrm{nat},\boxdot(Q)))$.

Proof
By definition we have $\Phi_H(\ell{:}(\mathrm{nat},Q)) = \sum_k q_k \binom{n_1\cdot n_2}{k}$. Moreover, $\Phi_{H'}(\ell'{:}(\mathrm{nat}*\mathrm{nat},\boxdot(Q))) = \sum_{(i,j)} q'_{(i,j)} \binom{n_1}{i}\binom{n_2}{j}$ for some coefficients $q'_{(i,j)} \in \mathbb{Q}_0^+$. From the definition of the multiplicative convolution $\boxdot(Q)$ it follows that $q'_{(i,j)} {=} \sum_k A(i,j,k)\, q_k$. Thus we can use (4) to obtain

$$\begin{aligned}
\sum_k q_k \binom{n_1\cdot n_2}{k} &= \sum_k q_k \left(\sum_{i,j} A(i,j,k)\binom{n_1}{i}\binom{n_2}{j}\right)\\
&= \sum_{k,i,j} q_k{\cdot}A(i,j,k)\binom{n_1}{i}\binom{n_2}{j}\\
&= \sum_{i,j}\left(\sum_k A(i,j,k) q_k \binom{n_1}{i}\binom{n_2}{j}\right)\\
&= \sum_{(i,j)} q'_{(i,j)}\binom{n_1}{i}\binom{n_2}{j} \qquad \Box
\end{aligned}$$

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. A proof can be found for instance in (Hoffmann *et al.*, 2012a).

Lemma 4.5
Let A be a data type. Then there are natural numbers $c_k^{(i,j)}$ for $i,j,k \in \mathrm{Ind}(A)$ with $\deg(k) \leq \deg(i,j)$ such that the following holds. For every context $\Gamma, x_1{:}A, x_2{:}A; Q$ and every H, V with $H \vDash V : \Gamma, x{:}A$ it holds that $\Phi_{V,H}(\Gamma, x{:}A; Q') = \Phi_{V',H}(\Gamma, x_1{:}A, x_2{:}A; Q)$ where $V' = V[x_1, x_2 \mapsto V(x)]$ and $q'_{(\ell,k)} = \sum_{i,j\in\mathrm{Ind}(A)} c_k^{(i,j)} q_{(\ell,i,j)}$.

The coefficients $c_k^{(i,j)}$ can be computed effectively. We were however not able to derive a closed formula for the coefficients. The proof can be found in a previous article (Hoffmann *et al.*, 2012a).

For a context $\Gamma, x_1{:}A, x_2{:}A; Q$ we define $\curlyvee Q$ to be the Q' from Lemma 4.5.

5 Resource-Aware Type System

We now describe the type-based amortized analysis for programs with unsigned integers and arrays.

Type Judgments. The declarative type rules for RAML expressions in Figure 5 and Figure 6 define a *resource-annotated typing judgment* of the form

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

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 of function identifiers to *sets of* resource-annotated function types. We need multiple types per function since it is often necessary to pass on potential to the function result for later use in the program as demonstrated with the function mult in Section 2. The potential that needs to be passed on—and thus the annotated function type—depends on the call site.

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 $\Sigma;y_f{:}A;Q \stackrel{M}{\vdash} e_f : (B,Q')$ for every function type $(A,Q) \to (B,Q') \in \Sigma(f)$.

Type Rules. Figure 5 and Figure 6 contain the annotated type rules. The rules T:WEAK-A and T:WEAK-C 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 rules T:VAR, T:APP, T:PAIR, T:MATP, T:SHARE, T:LET, T:MATN, and T:WEAK-* are similar to the corresponding rules in previous work (Hoffmann *et al.*, 2011).

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 with the resulting potential Q'. Intuitively, this is sound because the program is aborted when evaluating the expression 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

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

$$\frac{Q = Q' + M^{\mathsf{pair}}}{\Sigma; x_1{:}A_1, x_2{:}A_2; Q \overset{M}{\vdash} (x_1, x_2) : (A_1 {*} A_2, Q')}\ (\text{T:PAIR})$$

$$\frac{\Sigma; \Gamma, x_1{:}A_1, x_2{:}A_2; P \overset{M}{\vdash} e : (B, Q') \qquad P + M^{\mathsf{matP}} = Q}{\Sigma; \Gamma, x{:}A; Q \overset{M}{\vdash} \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \Rightarrow e : (B, Q')}\ (\text{T:MATP})$$

$$\frac{\Sigma; \Gamma, x_1{:}A, x_2{:}A; P \overset{M}{\vdash} e : (B, Q') \qquad \curlyvee P + M^{\mathsf{share}} = Q}{\Sigma; \Gamma, x{:}A; Q \overset{M}{\vdash} \mathsf{share}\ x\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e : (B, Q')}\ (\text{T:SHARE})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma_1, \Gamma_2; R \overset{M}{\vdash} e_1 \rightsquigarrow \Gamma_2, x{:}A; R' \\ \Sigma; \Gamma_2, x{:}A; P \overset{M}{\vdash} e_2 : (B, Q') \qquad Q = R + M^{\mathsf{let1}} \qquad R' = P + M^{\mathsf{let2}}\end{array}}{\Sigma; \Gamma_1, \Gamma_2; Q \overset{M}{\vdash} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (B, Q')}\ (\text{T:LET})$$

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

$$\frac{Q' + M^{\mathsf{sub}} = \boxplus(\pi_0^{x_1:\mathrm{nat}}(Q))}{\Sigma; x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{minus}(x_1, x_2) : (\mathrm{nat} * \mathrm{nat}, Q')}\ (\text{T:SUB}) \qquad \frac{q_0 = M^{\mathsf{nat}} + \sum_{i \geq 0} q'_i \binom{n}{i}}{\Sigma; \cdot; Q \overset{M}{\vdash} n : (\mathrm{nat}, Q')}\ (\text{T:NAT})$$

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

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

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

$$\frac{\begin{array}{c}\Sigma; \Gamma; R \overset{M}{\vdash} e_1 : (B, Q') \\ R + M^{\mathsf{matZ}} = \pi_0^{\Gamma}(Q) \qquad \Sigma; \Gamma, y{:}\mathrm{nat}; P \overset{M}{\vdash} e_2 : (B, Q') \qquad P + M^{\mathsf{matS}} = \lhd(Q)\end{array}}{\Sigma; \Gamma, x{:}\mathrm{nat}; Q \overset{M}{\vdash} \mathsf{match}\ x\ \mathsf{with}\ \langle 0 \Rightarrow e_1 \mid \mathsf{S}(y) \Rightarrow e_2 \rangle : (B, Q')}\ (\text{T:MATN})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma; P \overset{M}{\vdash} e : (B, P') \\ Q \geq P + c \qquad Q' \leq P' + c\end{array}}{\Sigma; \Gamma; Q \overset{M}{\vdash} e : (B, Q')}\ (\text{T:WEAK-A}) \qquad \frac{\Sigma; \Gamma; \pi_0^{\Gamma}(Q) \overset{M}{\vdash} e : (B, Q')}{\Sigma; \Gamma, x{:}A; Q \overset{M}{\vdash} e : (B, Q')}\ (\text{T:WEAK-C})$$

Fig. 5. Annotated type rules.

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

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

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

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

Fig. 6. Annotated Type rules for array operations.

$$\frac{\begin{array}{c}\forall\, j \in \mathrm{Ind}(\Delta): \qquad j{=}\vec{0} \implies \Sigma;\Gamma;\pi^{\Gamma}_{j}(Q) \;\vdash^{\underline{M}}\; e : (A, \pi^{x:A}_{j}(Q')) \\ j{\neq}\vec{0} \implies \Sigma_j;\Gamma;\pi^{\Gamma}_{j}(Q) \;\vdash^{\underline{\mathsf{cf}}}\; e : (A, \pi^{x:A}_{j}(Q'))\end{array}}{\Sigma;\Gamma,\Delta;Q \;\vdash^{\underline{M}}\; e \rightsquigarrow \Delta, x{:}A; Q'}\ \text{(B:BIND)}$$

Fig. 7. The binding rule for multivariate variable binding.

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 anyway 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 integer 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. The only reason why we use minus also for constants is to present a unified syntax to the user.

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 A.get and 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 A.set returns 0 in this paper. In RAML, 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 unsigned integer.

In the rule T:LET for let bindings, we bind the result of the evaluation of an expression e to a variable x. The problem that arises is that the resulting annotated context $\Delta, x{:}A, Q'$ features potential functions whose domain consists of data that is referenced by x as well as data that is referenced by Δ. This potential has to be related to data that is referenced by Δ and the free variables in the expression e.

To express the relations between mixed potentials before and after the evaluation of e, we introduce a new auxiliary binding judgment of the from

$$\Sigma; \Gamma, \Delta; Q \overset{M}{\vdash} e \rightsquigarrow \Delta, x{:}A; Q'$$

in the rule B:BIND in Figure 7. The intuitive meaning of the judgment is the following. Assume that e is evaluated in the context Γ, Δ, that $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$, and that e evaluates to a value that is bound to the variable x. Then the initial potential $\Phi(\Gamma, \Delta; Q)$ is larger than the cost of evaluating e in the metric M plus the potential of the resulting context $\Phi(\Delta, x{:}A; Q')$. Lemma 5.1 formalizes this intuition.

Lemma 5.1
Let $H \vDash V{:}\Gamma, \Delta$ and $\Sigma; \Gamma, \Delta; Q \overset{M}{\vdash} e \rightsquigarrow \Delta, x{:}A; Q'$.

1. If $V,H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w,d)$ then $\Phi_{V,H}(\Gamma, \Delta; Q) \geq d + \Phi_{V',H'}(\Delta, x{:}A; Q')$ where $V' = V[x \mapsto \ell]$.
2. If $V,H \overset{M}{\vdash} e \Downarrow \rho \mid (w,d)$ then $d \leq \Phi_{V,H}(\Gamma; Q)$.

Formally, Lemma 5.1 is a consequence of the soundness of the type system (Theorem 5.1). In the inductive proof of Theorem 5.1, we use a weaker version of Lemma 5.1 in which the soundness of the type judgments in Lemma 5.1 is an additional precondition.

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

Theorem 5.1 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 (Hoffmann *et al.*, 2012a). The novel type rules are mainly proved by the Lemmas 4.2, 4.3, and 4.4. For example, the induction case for multiplication in the first part of the Theorem 5.1 works as follows.

(T:MULT) Assume that the type derivation ends with an application of the rule T:MULT. Then e has the form $x_1 * x_2$ and the evaluation consists of a single application of the rule E:MULT. Therefore we can apply Lemma 4.4 and derive $\Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};\boxdot(Q')) = \Phi_{V,H'}(v : (\text{nat},Q'))$ where $v = H(V(x_1)) \cdot H(V(x_2))$. Then it follows from the rule T:MULT that $\Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};Q) - \Phi_{V,H'}(v : (\text{nat},Q')) = q_{(0,0)} - q'_0 = M^{\mathsf{mult}}$.

If $M^{\mathsf{mult}} \geq 0$ then it follows $p = M^{\mathsf{mult}}$ and $p' = 0$. Thus we have $p = K^{op} \leq q_{(0,0)} = \Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};Q)$ and $p - p' = M^{\mathsf{mult}} = \Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};Q) - (\Phi_{V,H'}(v{:}(\text{nat},Q'))$.

If $M^{\mathsf{mult}} < 0$ then it follows that $p = 0$ and $p' = -M^{\mathsf{mult}}$. Therefore we have $p \leq q = \Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};Q)$ and $p - p' = M^{\mathsf{mult}} = \Phi_{V,H}(x_1{:}\text{nat},x_2{:}\text{nat};Q) - (\Phi_{V,H'}(v : (\text{nat},Q')))$.

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, which is used in the proof of Theorem 5.1.

Lemma 5.2
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 $\Phi_{V,H}(\Gamma;Q) = \Phi_{V,H'}(\Gamma;Q)$.

To prove Lemma 5.2 we need a definition and two propositions. We say that two heaps H_1 and H_2 are *compatible (modulo arrays)* if for all $\ell \in \text{dom}(H_1 \cap H_2)$

$$H_1(\ell) = H_2(\ell) \quad \text{or,} \quad H_1(\ell) = (\sigma_1,n) \quad \text{and} \quad H_2(\ell) = (\sigma_2,n)\ .$$

Proposition 5.1
Let H_1 and H_2 be compatible heaps, $H_1 \vDash \ell \mapsto a_1 : A$, and $\ell \in \text{dom}(H_2)$. Then $H_2 \vDash \ell \mapsto a_2 : A$; moreover $a_1 = a_2$, or $a_1 = (\alpha_1,n)$ and $a_2 = (\alpha_2,n)$ for some α_1, α_2, and n.

Proposition 5.1 follows by induction on the derivation of the judgement $H_1 \vDash \ell \mapsto a_1 : A$.

Proposition 5.2

Let H_1 and H_2 be compatible heaps, $H_1 \vDash \ell{:}A$, and Q a potential annotation for type A. If $\ell \in \mathrm{dom}(H_2)$ then $\Phi_{H_1}(\ell{:}(A,Q)) = \Phi_{H_2}(\ell{:}(A,Q))$.

Proof
By definition of $H_1 \vDash \ell{:}A$ there exists an a_1 such that $H_1 \vDash \ell \mapsto a_1 : A$. Thus it follows from Proposition 5.1 that there exists an a_2 such that $H_2 \vDash \ell \mapsto a_2 : A$. If $a_1 = a_2$ then by definition $\Phi_{H_1}(\ell{:}(A,Q)) = \Phi_{H_2}(\ell{:}(A,Q))$. Otherwise it follows by Proposition 5.1 that $a_1 = (\alpha_1, n)$ and $a_2 = (\alpha_2, n)$ for some α_1, α_2, and n. But then $\Phi_{H_1}(\ell{:}(A,Q)) = \Phi_{H_2}(\ell{:}(A,Q))$ since the potential of an array depends by definition only on the size n of an array.

Proof of Lemma 5.2
Let $H \vDash V{:}\Gamma$, $\Sigma;\Gamma;Q \overset{M}{\vdash} e : (B,Q')$ and $V,H \overset{M}{\vdash} e \Downarrow (\ell,H') \mid (p,p')$. By induction on the derivation of the judgement $V,H \overset{M}{\vdash} e \Downarrow (\ell,H') \mid (p,p')$ it follows that H and H' are compatible and that $\mathrm{dom}(H) \subseteq \mathrm{dom}(H')$. Thus $H' \vDash V{:}\Gamma$ and $\Phi_{V,H'}(\Gamma;Q)$ is defined. Thus

$$\begin{array}{rcl} \Phi_{V,H}(\Gamma;Q) & = & \sum_{(i_1,\dots,i_n)\in \mathrm{Ind}_k(\Gamma)} q_{\vec{\imath}} \prod_{j=1}^{n} p_{i_j}(a_{x_j}) \\ & = & \sum_{(i_1,\dots,i_n)\in \mathrm{Ind}_k(\Gamma)} q_{\vec{\imath}} \prod_{j=1}^{n} p_{i_j}(a'_{x_j}) \quad = \quad \Phi_{V,H'}(\Gamma;Q) \end{array}$$

where $H \vDash V(x_j) \mapsto a_{x_j} : \Gamma(x_j)$ and $H' \vDash V(x_j) \mapsto a'_{x_j} : \Gamma(x_j)$. The second equation holds because the sums are pointwise equal. This is a direct consequence of Proposition 5.2.

If the metric M is simple (all constants are 1) then it follows from Theorem 5.1 that the bounds on the resource usage also prove the termination of programs.

Corollary 5.1
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 $d \leq \Phi_{V,H}(\Gamma;Q)$ such that $V,H \overset{M}{\vdash} e \Downarrow (\ell,H') \mid (w,d)$ for some ℓ and H'.

6 Type Inference

In principle, type inference consists of four steps. First, we perform a classic type inference for the simple types such as (Arr(nat)). 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 (Hoffmann & Hofmann, 2010a). 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 (Hoffmann *et al.*, 2012a). Finally, we use several optimizations to reduce the number of generated constraints.

For the most part, our constraints have the form of a so-called network (or network-flow) problem (Vanderbei, 2001). LP solvers can handle network problems very efficiently and CPLEX solves the constraints RAML generates in linear time in practice. Because our problem sizes are large we can save memory and time by reducing the number of constraints that are generated during typing. A representative example of an optimization is that we try to reuse constraint names instead of producing constraints like $p = q$.

Example Inference. In the following, we illustrate the constraint generation during type inference with an example. As in Section 2, we use the resource metric that counts the number of pattern matches that are performed during an evaluation. The function quad below is a particularly simple example that uses the new rule for addition and performs a quadratic number of match operations.

```
quad(n,m) = match n with
              | 0  → 0
              | S n'  → match m with
                           | 0  → let y = 0 in quad(n',y)
                           | S m'  → share m' as (m1,m2) in
                                        let x = n' + m1 in
                                        quad(x,m2);
```

The purpose of quad is only to illustrate the type inference and the computed value is not interesting: The function takes two unsigned integers n and m and always returns 0. Nevertheless, the number of performed match operations is $1+2n+2\binom{m}{2}$ in the worst case. A valid type in our type system is thus

$$\mathsf{quad} : (\mathrm{nat} * \mathrm{nat}, Q) \to (\mathrm{nat}, Q')\ ,$$

where $q_{(0,0)} = 1, q_{(1,0)} = 2, q_{(0,2)} = 2$, and $q_i = q'_j = 0$ for all other i and all j.

The first step in the type inference is to select a maximal degree to limit the search space and to obtain a finite set of constraints. For simplicity, we set the maximal degree to 2. So we have $Q = (q_{(0,0)}, q_{(1,0)}, q_{(2,0)}, q_{(1,1)}, q_{(0,1)}, q_{(0,2)})$ and $Q' = (q'_0, q'_1, q'_2)$, where the q_i and q'_j are yet unknown rational variables that are to be determined by the LP solver.

Since the function quad is tail-recursive, we do not need resource-polymorphic recursion and simply assume the (yet unknown) type $\mathsf{quad} : (\mathrm{nat} * \mathrm{nat}, Q) \to (\mathrm{nat}, Q')$ at the recursive calls. Polymorphic recursion is needed to pass-on potential for consumption in the code that follows a recursive call in the function body. For more information about resource-polymorphic recursion refer to (Hoffmann & Hofmann, 2010a).

We now use the type rules defined in Section 5 to generate a set of linear constraints so that every valid instantiation of the constraints results in a valid type for quad. To this end, we set

$$\Sigma(\mathrm{quad}) = \{(\mathrm{nat}{*}\mathrm{nat}, Q) \to (\mathrm{nat}, Q')\}$$

and produce a type derivation

$$\Sigma; n : \mathrm{nat}, m : \mathrm{nat}; Q \overset{M}{\vdash} e_{\mathrm{quad}} : (nat, Q')$$

for the function body e_{quad} that contains variables in the places of the coefficients of the potential functions.

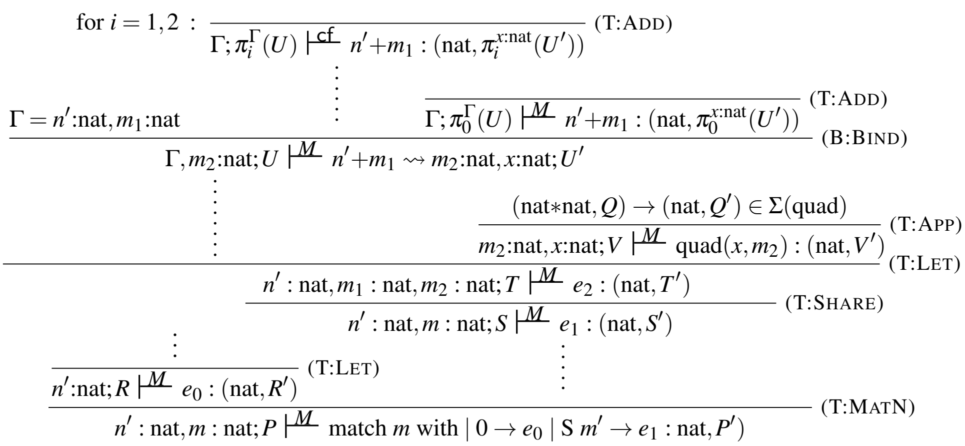
for $i = 1,2$:
$\Gamma;\pi_i^{\Gamma}(U) \overset{\mathrm{cf}}{\vdash} n'{+}m_1 : (\mathrm{nat},\pi_i^{x:\mathrm{nat}}(U'))$
(T:ADD)
$\Gamma = n':\mathrm{nat}, m_1:\mathrm{nat}$
$\Gamma;\pi_0^{\Gamma}(U) \overset{M}{\vdash} n'{+}m_1 : (\mathrm{nat},\pi_0^{x:\mathrm{nat}}(U'))$
(T:ADD)
(B:BIND)
$\Gamma, m_2:\mathrm{nat}; U \overset{M}{\vdash} n'{+}m_1 \rightsquigarrow m_2:\mathrm{nat}, x:\mathrm{nat}; U'$
$(\mathrm{nat}{*}\mathrm{nat}, Q) \to (\mathrm{nat}, Q') \in \Sigma(\mathrm{quad})$
(T:APP)
$m_2:\mathrm{nat}, x:\mathrm{nat}; V \overset{M}{\vdash} \mathrm{quad}(x, m_2) : (\mathrm{nat}, V')$
(T:LET)
$n' : \mathrm{nat}, m_1 : \mathrm{nat}, m_2 : \mathrm{nat}; T \overset{M}{\vdash} e_2 : (\mathrm{nat}, T')$
(T:SHARE)
$n' : \mathrm{nat}, m : \mathrm{nat}; S \overset{M}{\vdash} e_1 : (\mathrm{nat}, S')$
$n':\mathrm{nat}; R \overset{M}{\vdash} e_0 : (\mathrm{nat}, R')$
(T:LET)
(T:MATN)
$n' : \mathrm{nat}, m : \mathrm{nat}; P \overset{M}{\vdash} \mathrm{match}\ m\ \mathrm{with}\ \mid 0 \to e_0 \mid \mathrm{S}\ m' \to e_1 : \mathrm{nat}, P')$

Fig. 8. Type derivation for the main parts of the function quad.

We illustrate this process in detail for the most interesting part of the derivation, namely the inner match expression. Figure 8 shows the type derivation for the expression

$$\mathrm{match}\ m\ \mathrm{with}\ \mid 0 \to e_0 \mid \mathrm{S}\ m' \to e_1$$

where we use the following abbreviations.

$$\begin{aligned} e_0 &\equiv \mathrm{let}\ y = 0\ \mathrm{in}\ \mathrm{quad}(n', y) \\ e_1 &\equiv \mathrm{share}\ m'\ \mathrm{as}\ (m_1, m_2)\ \mathrm{in}\ e_2 \\ e_2 &\equiv \mathrm{let}\ x = n' + m_1\ \mathrm{in}\ \mathrm{quad}(x, m_2) \end{aligned}$$

The constraints are generated according to the preconditions of the respective type rules. In the rightmost application of the rule T:ADD, we have $\pi_0^{\Gamma}(U) = \boxplus(\pi_0^{x:\mathrm{nat}}U')$ (recall that $M^{\mathsf{add}} = 0$ in our metric). This corresponds to the following set of linear constraints.

$$\begin{array}{lll} u_{(0,0,0)} = u'_{(0,0)} & u_{(0,1,0)} = u'_{(1,0)} & u_{(1,0,0)} = u'_{(1,0)} \\ u_{(1,1,0)} = u'_{(2,0)} & u_{(0,2,0)} = u'_{(2,0)} & u_{(2,0,0)} = u'_{(2,0)} \end{array}$$

The constraints generated in the two cost-free applications of T:ADD are

$$u_{(0,0,2)} = u'_{(0,2)} \qquad u_{(0,0,1)} = u'_{(0,1)} \qquad u_{(0,1,1)} = u'_{(1,1)} \qquad u_{(1,0,1)} = u'_{(1,1)} \ .$$

Similarly, since $M^{\mathsf{app}} = 0$, we have $Q = V$ and $Q' = V'$ in the rule T:APP, that is,

$$\begin{array}{lllll} v_{(0,0)} = q_{(0,0)} & v_{(1,0)} = q_{(1,0)} & v_{(0,1)} = q_{(0,1)} & v'_0 = q'_0 & \\ v_{(1,1)} = q_{(1,1)} & v_{(2,0)} = q_{(2,0)} & v_{(0,2)} = q_{(0,2)} & v'_1 = q'_1 & v'_2 = q'_2\ . \end{array}$$

In our metric, the rule T:LET requires simply that $T = U$, $U' = V$, and $V' = T'$, which can be directly expressed in linear constraints as well.

For the application of T:SHARE we generate constraints expressing $T' = S'$ as well as the sharing following constraints.

$$\begin{array}{lll} s_{(0,0)} = t_{(0,0,0)} & s_{(1,0)} = t_{(1,0,0)} & s_{(0,1)} = t_{(0,1,0)} + t_{(0,1,0)} + 2t_{(0,1,1)} \\ s_{(2,0)} = t_{(2,0,0)} & s_{(1,1)} = t_{(0,1,0)} + t_{(0,1,0)} & s_{(0,2)} = 2{\cdot}t_{(0,1,1)} \end{array}$$

In the rule T:MATN we require $S' = P'$ and $S+1 = \triangleleft(P)$, which corresponds to the following constraint set. The constant 1 in the first equation reflects the fact that we count the number of match operations, that is, $M^{\mathsf{matS}} = 1$.

$$s_{(0,0)} + 1 = p_{(0,0)} + p_{(0,1)} \qquad s_{(0,1)} = p_{(0,1)} + p_{(0,2)} \qquad s_{(0,2)} = p_{(0,2)}$$
$$s_{(1,0)} = p_{(1,0)} + p_{(1,1)} \qquad s_{(1,1)} = p_{(1,1)} \qquad s_{(2,0)} = p_{(2,0)}$$

Finally, we generate constraints for $P' = Q'$. The remainder of the type derivation of e_{quad} (not shown here) would also relate the coefficients in P with the ones in Q.

7 Experimental Evaluation

We have implemented our analysis system in Resource Aware ML (RAML) (Aehlig *et al.*, 2010-2013; Hoffmann *et al.*, 2012b) and tested the new analysis on multiple classical example 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 $O(\log m)$). Experiments with example inputs 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 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 $1 \leq j \leq i \leq n$ (the polynomials are just 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. The function for the flattening is especially interesting since it requires a lexicographic order to prove termination.

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
mmultAll : (L(Mat),Mat) → Mat	$18nuyx+31nuy+38nu+38n+3$	5.6 s	184270
blocksort : (Arr(int),nat) → unit	$12.25n^2+90.25n+18$	0.4 s	27795
dyadAllM : nat → unit	$1.\bar{6}n^6+334.8n^4+1485.08n^3+$ $37n^5+2963.54n^2+1789.92n+3$	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.*

Figures 9 and 10 show a comparison of the computed evaluation-step bounds for the functions dijkstra, quicksort, dyadAllM, and mmultAll with the measured evaluation steps in the cost semantics for several input sizes. The experiments show that the bounds for dyadAllM and mmultAll are tight. The bounds for dijkstra and quicksort are only asymptotically tight. The relative looseness of the bound for quicksort (ca. 20%) is in some sense a result of the compositionality of the analysis: The worst-case behavior of quick sort's partition function materializes if the number of *swaps* performed in the partitioning is maximal. However, the number of swaps that are performed by the partitioning in a worst-case run of quicksort is relatively low. Nevertheless, the analysis has to assume the worst-case for each call of the partition function.

We did not perform an experimental comparison with abstract interpretation-based resource analysis systems. Many systems that are described in the literature are not publically available. The COSTA system (Albert *et al.*, 2012b; Albert *et al.*, 2012a) 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 (Alonso-Blas *et al.*, 2011)), 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. Note however that the LP solving during the inference of the potential functions is not modular if we want to derive the most precise bounds: To allow different potential at every function call, we have to copy the constraints for a function body to every call side.

A Case Study. In the remainder of this section, we present a larger case study that we successively develop. It highlights the advantages of our analysis; namely compositionality and the seamless analysis of non-linear size changes.

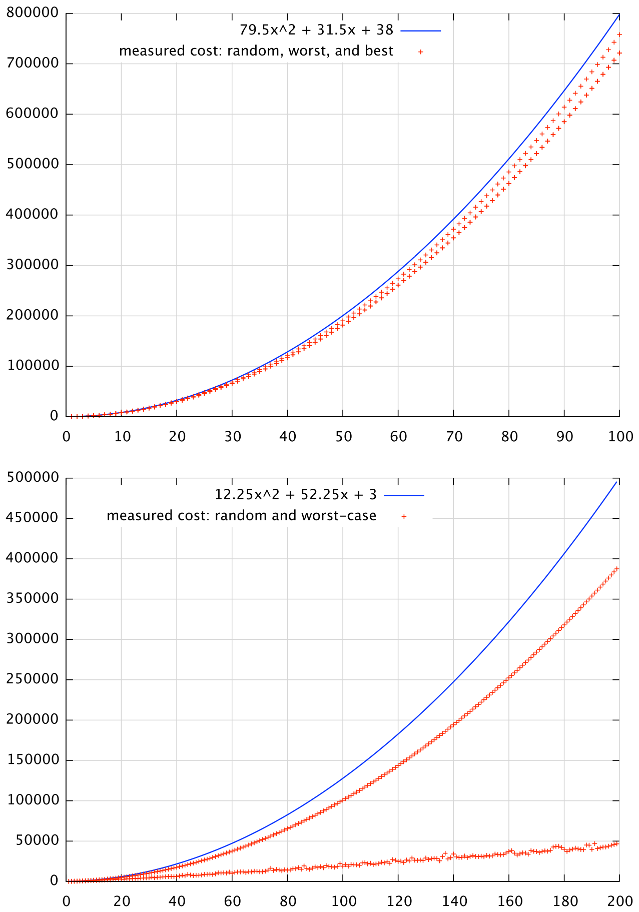

Fig. 9. Derived evaluation-step bounds in comparison with the measured evaluation steps for inputs of different sizes. On the top, the bound for dijkstra is compared with manually selected worst-case inputs (complete graphs with x nodes for $1 \leq x \leq 100$ and hand-picked edge weights), random inputs (graphs with randomly generated edge weights), and best-case inputs (empty graphs). The measured costs for the random and best-case inputs are very close. At the bottom, the bound for quicksort is compared to worst-case inputs (reversely-sorted arrays of sizes 1 to 200) and randomly filled arrays of the same sizes.

We start with the function dyad that takes two arrays a and b and two unsigned integers n and m. It then creates a matrix (an array of arrays) of size $n \times m$ by computing the dyadic product of the prefix of a of length n and the prefix of b of length m.

```
dyad : (Arr(int),nat,Arr(int),nat)  → Arr(Arr(int))
dyad (a,n,b,m) = let outerArr = A.make(n,A.make(0,+0)) in
                 let _ = fill(a,n,b,m,outerArr) in outerArr;

fill : (Arr(int),nat,Arr(int),nat,Arr(Arr(int)))  → unit
fill(a,n,b,m,outerArr) = match n with | 0  → ()
    | S n'  → let newLine = A.make(m,+0) in
              let _ = multArr(A.get(a,n'),b,newLine,m) in
              let _ = A.set(outerArr,n',newLine) in
```

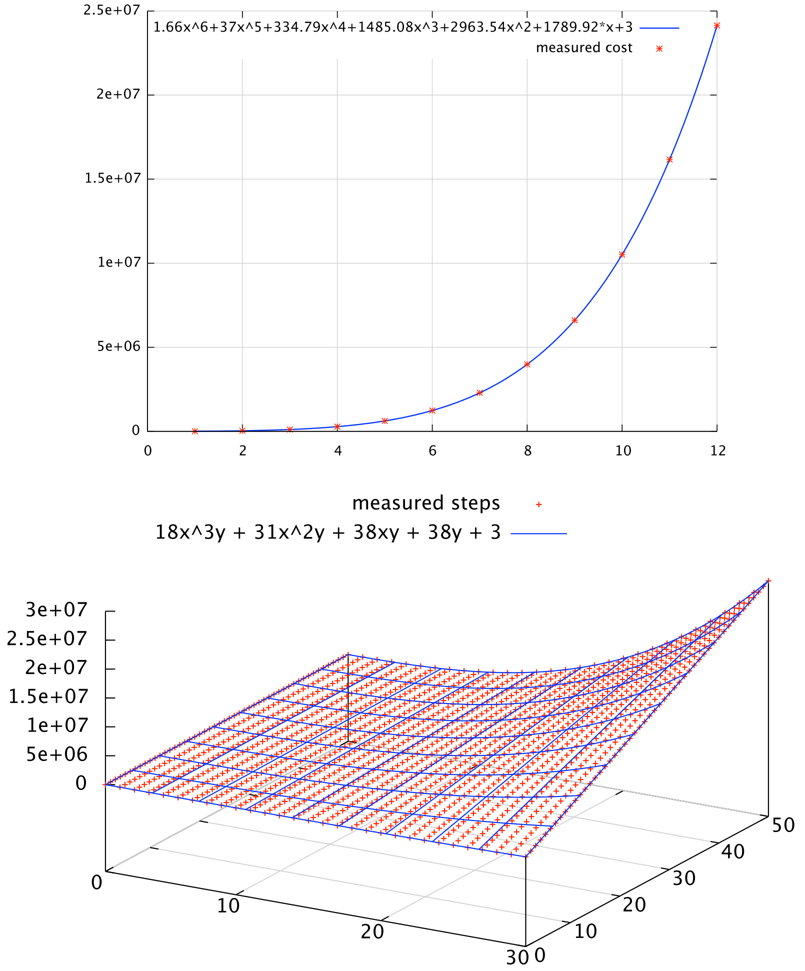

Fig. 10. Derived evaluation-step bounds in comparison with the measured evaluation steps for inputs of different sizes. On the top, the bound for dyadAllM is compared to the measured cost for inputs x where $x \in \{1,\dots,12\}$. At the bottom, the bound for mmultAll is compared to inputs that contain a list of y quadratic matrices of dimension $x \times x$ where $1 \leq x \leq 30$ and $1 \leq y \leq 50$. The experiments indicated that the derived bounds are optimal.

```
                fill(a,n',b,m,outerArr);

  multArr : (int,Arr(int),Arr(int),nat) → unit
  multArr(q,b,res,m) = match m with | 0 → ()
                              | S m' → let p = A.get(b,m') in
                                       let _ = A.set(res,m',q*p) in
                                       multArr(q,b,res,m');
```

The analysis computes the evaluation-step bound $20nm+31n+18$ for the function dyad. Our experiments with small example inputs indicate that this bound is tight. The analysis takes 0.5 seconds.

We now define the function matrix : (nat,nat) → Arr(Arr(int)) that takes two numbers n and m and computes a $(n^2{+}9n{+}28) \times (mn{+}6m)$ matrix using dyad. The polynomials are just a random choice and the analysis would work with any other polynomial as long as the coefficients created for the constants in the linear constraints (e.g. $\binom{28}{4}$) do not overflow the LP solver. Since the elements of the vectors that we use to create the matrix do not influence the evaluation cost we choose them arbitrarily.

```
matrix (n,m) = let size1 = n*n + 9*n + 28 in
               let size2 = m*n + 6*m in
               dyad(A.make(size1,+1),size1,A.make(size2,+1),size2);
```

Within 1 second, RAML computes the following evaluation-step bound for the function matrix. Our experiments indicate that the bound is tight.

$$20mn^3 + 300mn^2 + 1641mn + 3366m + 32n^2 + 288n + 942$$

Next, we implement the function dyadAll: nat → unit which, given an unsigned integer n, computes a dyadic product $\mathsf{dyad}(\mathsf{a},\mathsf{i},\mathsf{b},\mathsf{j})$ for every pair of numbers i, j such that $1 \le j \le i \le n$.

```
dyadAll n = match n with | 0 → ()
            | S n' → let _ = dyadP(n,n) in dyadAll(n');

dyadP(n,m) = match m with | 0 → ()
    | S m' → let mat = dyad(A.make(n,+1),n,A.make(m,+1),m) in
              dyadP(n,m');
```

In 1.5 seconds, RAML computes the following bound for dyadAll. Note that the bound function takes values in $\mathbb{N}$ if $n \in \mathbb{N}$. Again, our experiments indicate that the bound is tight.

$$2.5n^4 + 19.1\bar{6}n^3 + 41.5n^2 + 36.8\bar{3}n + 3$$

Now, we define the function dyadAllM that is identical to dyadAll except that we replace the call $\mathsf{dyad}(\mathsf{A.make}(\mathsf{n},+1),\mathsf{n},\mathsf{A.make}(\mathsf{m},+1),\mathsf{m})$ in dyadP with the call $\mathsf{matrix}(\mathsf{n},\mathsf{m})$. As a result, $\mathsf{dyadAllM}(\mathsf{n})$ computes a matrix of size $(i^2{+}9i{+}28) \times (ij{+}6j)$ for every pair of numbers i, j such that $1 \le j \le i \le n$. RAML computes the following tight evaluation-step bound in 5.8 seconds. Since the coefficients in the binomial basis are unsigned integers, the bound function takes values in $\mathbb{N}$.

$$1.\bar{6}n^6 + 37n^5 + 334.791\bar{6}n^4 + 1485.08\bar{3}n^3 + 2963.541\bar{6}n^2 + 1789.91\bar{6}n + 3$$

Finally, we show an application of the built-in function minus. The following function dyadSub : (nat,nat) → unit takes two numbers n and m, recursively subtracts $m+1$ from n, and calls $\mathsf{dyadAll}(\mathsf{m}+1)$ until $n \le m$. Then dyadSub calls $\mathsf{dyadAll}(\mathsf{n})$.

```
dyadSub (n,m) = if (n > m ) then
    let (m,d) = minus(n,m) in
    let (_,d) = minus(d,1) in
    let _ = dyadAll(m+1) in dyadSub(d,m)
  else dyadAll(n);
```

To be able to analyze the function, we have to execute the subtraction of $m+1$ in two steps. First we subtract m and then we subtract the constant 1. This is necessary because the analysis does not perform a value analysis and does not infer that $m+1 \geq 1$. So it cannot be aware of the fact that $n-(m+1) < n$ if $n > m$. If we split the subtraction into two parts then RAML computes the following bound in 1.4 seconds.

$$2.5n^4 + 19.1\bar{6}n^3 + 41.5n^2 + 60.8\bar{3}n + 11$$

Of course, the previous programs are somewhat artificial but they demonstrate quickly some of the capabilities of the analysis. We invite you to experiment with other examples in the web interface of RAML (Aehlig *et al.*, 2010-2013).

8 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. We are confident that the developed rules for arithmetic expression can be reused when moving to a different programming language. Our next step is to develop a type-and-effect system that applies the ideas of this work to an imperative language with while-loops, unsigned 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

Aehlig, Klaus, Hofmann, Martin, & Hoffmann, Jan. (2010-2013). *RAML Web Site*. `http://raml.tcs.ifi.lmu.de`.

Albert, Elvira, Arenas, Puri, Genaim, Samir, & Puebla, Germán. (2011). Closed-Form Upper Bounds in Static Cost Analysis. *Journal of automated reasoning*, 161–203.

Albert, Elvira, Arenas, Puri, Genaim, Samir, Gómez-Zamalloa, Miguel, & Puebla, Germán. (2012a). Automatic Inference of Resource Consumption Bounds. *Pages 1–11 of: Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12).*

Albert, Elvira, Arenas, Puri, Genaim, Samir, Puebla, German, & Zanardini, Damiano. (2012b). Cost Analysis of Object-Oriented Bytecode Programs. *Theor. comput. sci.*, **413**(1), 142 – 159.

Alias, Christophe, Darte, Alain, Feautrier, Paul, & Gonnord, Laure. (2010). Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. *Pages 117–133 of: 17th Int. Static Analysis Symposium (SAS'10).*

Alonso-Blas, Diego Esteban, & Genaim, Samir. (2012). On the limits of the classical approach to cost analysis. *Pages 405–421 of: 19th Int. Static Analysis Symp. (SAS'12).*

Alonso-Blas, Diego Esteban, Arenas, Puri, & Genaim, Samir. (2011). Handling Non-linear Operations in the Value Analysis of COSTA. *Electr. notes theor. comput. sci.*, **279**(1), 3–17.

Brockschmidt, Marc, Emmes, Fabian, Falke, Stephan, Fuhs, Carsten, & Giesl, Jürgen. (2014). Alternating Runtime and Size Complexity Analysis of Integer Programs. *Pages 140–155 of: Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14).*

Carbonneaux, Quentin, Hoffmann, Jan, Ramananandro, Tahina, & Shao, Zhong. (2014). End-to-End Verification of Stack-Space Bounds for C Programs. *Page 30 of: Conf. on Prog. Lang. Design and Impl. (PLDI'14).*

Carbonneaux, Quentin, Hoffmann, Jan, & Shao, Zhong. (2015). Compositional Certified Resource Bounds. *36th Conf. on Prog. Lang. Design and Impl. (PLDI'15).* Forthcoming.

Cousot, Patrick, & Halbwachs, Nicolas. (1978). Automatic Discovery of Linear Restraints Among Variables of a Program. *Pages 84–96 of: 5th ACM Symp. on Principles Prog. Langs. (POPL'78).*

Gulavani, Bhargav S., & Gulwani, Sumit. (2008). A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. *Pages 370–384 of: Comp. Aid. Verification, 20th Int. Conf. (CAV '08).*

Gulwani, Sumit, Mehra, Krishna K., & Chilimbi, Trishul M. (2009). SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. *Pages 127–139 of: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09).*

Hoffmann, Jan, & Hofmann, Martin. (2010a). Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. *Prog. Langs. and Systems - 8th Asian Symposium (APLAS'10).*

Hoffmann, Jan, & Hofmann, Martin. (2010b). Amortized Resource Analysis with Polynomial Potential. *19th Euro. Symp. on Prog. (ESOP'10).*

Hoffmann, Jan, & Shao, Zhong. (2014). Type-Based Amortized Resource Analysis with Integers and Arrays. *12th International Symposium on Functional and Logic Programming (FLOPS'14).*

Hoffmann, Jan, Aehlig, Klaus, & Hofmann, Martin. (2011). Multivariate Amortized Resource Analysis. *38th ACM Symp. on Principles of Prog. Langs. (POPL'11).*

Hoffmann, Jan, Aehlig, Klaus, & Hofmann, Martin. (2012a). Multivariate Amortized Resource Analysis. *Acm trans. program. lang. syst.*

Hoffmann, Jan, Aehlig, Klaus, & Hofmann, Martin. (2012b). Resource Aware ML. *24rd Int. Conf. on Computer Aided Verification (CAV'12).*

Hofmann, Martin, & Jost, Steffen. (2003). Static Prediction of Heap Space Usage for First-Order Functional Programs. *Pages 185–197 of: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03).*

Leroy, Xavier. (2006). Coinductive Big-Step Operational Semantics. *Pages 54–68 of: 15th Euro. Symp. on Prog. (ESOP'06).*

Miné, A. (2004). *Weakly relational numerical abstract domains*. Ph.D. thesis, École Polytechnique, Paris, France.

Riordan, John, & Stein, Paul R. (1972). Arrangements on Chessboards. *Journal of Combinatorial Theory, Series A*, **12**(1).

Sankaranarayanan, Sriram, Ivancic, Franjo, Shlyakhter, Ilya, & Gupta, Aarti. (2006). Static Analysis in Disjunctive Numerical Domains. *Pages 3–17 of: 13th Int. Static Analysis Symp. (SAS'06).*

Sinn, Moritz, Zuleger, Florian, & Veith, Helmut. (2014). A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. *Page 743–759 of: Computer Aided Verification - 26th Int. Conf. (CAV'14).*

Vanderbei, Robert J. (2001). *Linear Programming: Foundations and Extensions*. Springer US.

Zuleger, Florian, Sinn, Moritz, Gulwani, Sumit, & Veith, Helmut. (2011). Bound Analysis of Imperative Programs with the Size-change Abstraction. *Pages 280–297 of: 18th Int. Static Analysis Symp. (SAS'11).*