Automatic Static Cost Analysis for Parallel Programs

Jan Hoffmann and Zhong Shao

Yale University

Abstract. Static analysis of the evaluation cost of programs is an extensively studied problem that has many important applications. However, most automatic methods for static cost analysis are limited to sequential evaluation while programs are increasingly evaluated on modern multicore and multiprocessor hardware. This article introduces the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. The analysis is performed by a novel type system for amortized resource analysis. The main innovation is a technique that separates the reasoning about sizes of data structures and evaluation cost within the same framework. The cost semantics of parallel programs is based on call-by-value evaluation and the standard cost measures *work* and *depth*. A soundness proof of the type system establishes the correctness of the derived cost bounds with respect to the cost semantics. The derived bounds are multivariate resource polynomials which depend on the sizes of the arguments of a function. Type inference can be reduced to linear programming and is fully automatic. A prototype implementation of the analysis system has been developed to experimentally evaluate the effectiveness of the approach. The experiments show that the analysis infers bounds for realistic example programs such as quick sort for lists of lists, matrix multiplication, and an implementation of sets with lists. The derived bounds are often asymptotically tight and the constant factors are close to the optimal ones.

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

1 Introduction

Static analysis of the resource cost of programs is a classical subject of computer science. Recently, there has been an increased interest in formally proving cost bounds since they are essential in the verification of safety-critical real-time and embedded systems.

For sequential functional programs there exist many automatic and semi-automatic analysis systems that can statically infer cost bounds. Most of them are based on sized types [1], recurrence relations [2], and amortized resource analysis [3, 4]. The goal of these systems is to automatically compute easily-understood arithmetic expressions in the sizes of the inputs of a program that bound resource cost such as time or space usage. Even though an automatic

computation of cost bounds is undecidable in general, novel analysis techniques are able to efficiently compute tight time bounds for many non-trivial programs [5–9].

For functional programs that are evaluated in parallel, on the other hand, no such analysis system exists to support programmers with computer-aided derivation of cost bounds. In particular, there are no type systems that derive cost bounds for parallel programs. This is unsatisfying because parallel evaluation is becoming increasingly important on modern hardware and referential transparency makes functional programs ideal for parallel evaluation.

This article introduces an automatic type-based resource analysis for deriving cost bounds for parallel first-order functional programs. Automatic cost analysis for sequential programs is already challenging and it might seem to be a long shot to develop an analysis for parallel evaluation that takes into account low-level features of the underlying hardware such as the number of processors. Fortunately, it has been shown [10, 11] that the cost of parallel functional programs can be analyzed in two steps. First, we derive cost bounds at a high abstraction level where we assume to have an unlimited number of processors at our disposal. Second, we prove once and for all how the cost on the high abstraction level relates to the actual cost on a specific system with limited resources.

In this work, we derive bounds on an abstract cost model that consists of the *work* and the *depth* of an evaluation of a program [10]. Work measures the evaluation time of sequential evaluation and depth measures the evaluation time of parallel evaluation assuming an unlimited number of processors. It is well-known [12] that a program that evaluates to a value using work w and depth d can be evaluated on a shared-memory multiprocessor (SMP) system with p processors in time $O(\max(w/p, d))$ (see Section 2.3). The mechanism that is used to prove this result is comparable to a scheduler in an operating system.

A novelty in the cost semantics in this paper is the definition of work and depth for terminating and non-terminating evaluations. Intuitively, the non-deterministic big-step evaluation judgement that is defined in Section 2 expresses that *there is a (possibly partial) evaluation with work n and depth m*. This statement is used to prove that a typing derivation for bounds on the depth or for bounds on the work ensures termination.

Technically, the analysis computes two separate typing derivations, one for the work and one for the depth. To derive a bound on the work, we use multivariate amortized resource analysis for sequential programs [13]. To derive a bound on the depth, we develop a novel multivariate amortized resource analysis for programs that are evaluated in parallel. The main challenge in the design of this novel parallel analysis is to ensure the same high compositionality as in the sequential analysis. The design and implementation of this novel analysis for bounds on the depth of evaluations is the main contribution of our work. The technical innovation that enables compositionality is an analysis method that separates the static tracking of size changes of data structures from the cost analysis while using the same framework. We envision that this technique will find further applications in the analysis of other non-additive cost such as stack-space usage and recursion depth.

We describe the new type analysis for parallel evaluation for a simple first-order language with lists, pairs, pattern matching, and sequential and parallel composition. This is already sufficient to study the cost analysis of parallel programs. However, we implemented the analysis system in Resource Aware ML (RAML), which also includes other inductive data types and conditionals [14]. To demonstrate the universality of the approach, we also implemented NESL's [15] parallel list comprehensions as a primitive in RAML (see Section 6). Similarly, we can define other parallel sequence operations of NESL as primitives and correctly specify their work and depth. RAML is currently extended to include higher-order functions, arrays, and user-defined inductive types. This work is orthogonal to the treatment of parallel evaluation.

To evaluate the practicability of the proposed technique, we performed an experimental evaluation of the analysis using the prototype implementation in RAML. Note that the analysis computes worst-case bounds instead of average-case bounds and that the asymptotic behavior of many of the classic examples of Blelloch et al. [10] does not differ in parallel and sequential evaluations. For instance, the depth and work of quick sort are both quadratic in the worst-case. Therefore, we focus on examples that actually have asymptotically different bounds for the work and depth. This includes quick sort for lists of lists in which the comparisons of the inner lists can be performed in parallel, matrix multiplication where matrices are lists of lists, a function that computes the maximal weight of a (continuous) sublist of an integer list, and the standard operations for sets that are implemented as lists. The experimental evaluation can be easily reproduced and extended: RAML and the example programs are publicly available for download and through an user-friendly online interface [16].

In summary we make the following contributions.

1. We introduce the first automatic static analysis for deriving bounds on the depth of parallel functional programs. Being based on multivariate resource polynomials and type-based amortized analysis, the analysis is compositional. The computed type derivations are easily-checkable bound certificates.
2. We prove the soundness of the type-based amortized analysis with respect to an operational big-step semantics that models the work and depth of terminating and non-terminating programs. This allows us to prove that work and depth bounds ensure termination. Our inductively defined big-step semantics is an interesting alternative to coinductive big-step semantics.
3. We implemented the proposed analysis in RAML, a first-order functional language. In addition to the language constructs like lists and pairs that are formally described in this article, the implementation includes binary trees, natural numbers, tuples, Booleans, and NESL's parallel list comprehensions.
4. We evaluated the practicability of the implemented analysis by performing reproducible experiments with typical example programs. Our results show that the analysis is efficient and works for a wide range of examples. The derived bounds are usually asymptotically tight if the tight bound is expressible as a resource polynomial.

The full version of this article [17] contains additional explanations, lemmas, and details of the technical development.

2 Cost Semantics for Parallel Programs

In this section, we introduce a first-order functional language with parallel and sequential composition. We then define a big-step operational semantics that formalizes the cost measures *work* and *depth* for terminating and non-terminating evaluations. Finally, we prove properties of the cost semantics and discuss the relation of work and depth to the run time on hardware with finite resources.

2.1 Expressions and Programs

Expressions are given in let-normal form. This means that term formers are applied to variables only when this does not restrict the expressivity of the language. Expressions are formed by integers, variables, function applications, lists, pairs, pattern matching, and sequential and parallel composition.

$$\begin{aligned} e, e_1, e_2 ::= {} & n \mid x \mid f(x) \mid (x_1, x_2) \mid\mid \mathsf{match}\, x \,\mathsf{with}\, (x_1, x_2) \Rightarrow e \\ & \mid \mathsf{nil} \mid \mathsf{cons}(x_1, x_2) \mid \mathsf{match}\, x \,\mathsf{with}\, \langle \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(x_1, x_2) \Rightarrow e_2 \rangle \\ & \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \mid \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \end{aligned}$$

The parallel composition $\mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e$ is used to evaluate e_1 and e_2 in parallel and bind the resulting values to the names x_1 and x_2 for use in e.

In the prototype, we have implemented other inductive types such as trees, natural numbers, and tuples. Additionally, there are operations for primitive types such as Booleans and integers, and NESL's parallel list comprehensions [15]. Expressions are also transformed automatically into let normal form before the analysis. In the examples in this paper, we use the syntax of our prototype implementation to improve readability.

In the following, we define a standard type system for expressions and programs. Data types A, B and function types F are defined as follows.

$$A, B ::= \mathrm{int} \mid L(A) \mid A * B \qquad\qquad F ::= A \to B$$

Let $\mathcal{A}$ be the set of data types and let $\mathcal{F}$ be the set of function types. A signature $\Sigma : \mathrm{FID} \rightharpoonup \mathcal{F}$ is a partial finite mapping from function identifiers to function types. A context is a partial finite mapping $\Gamma : \mathit{Var} \rightharpoonup \mathcal{A}$ from variable identifiers to data types. A simple type judgement $\Sigma; \Gamma \vdash e : A$ states that the expression e has type A in the context Γ under the signature Σ. The definition of typing rules for this judgement is standard and we omit the rules.

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

2.2 Big-Step Operational Semantics

We now formalize the resource cost of evaluating programs with a big-step operational semantics. The focus of this paper is on time complexity and we only define the cost measures *work* and *depth*. Intuitively, the work measures the time that is needed in a sequential evaluation. The depth measures the time that is needed in a parallel evaluation. In the semantics, time is parameterized by a metric that assigns a non-negative cost to each evaluation step.

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow \circ \mid (w, d)}{V, H \overset{M}{\vdash} \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow \circ \mid (M^{\mathsf{let}}{+}w, M^{\mathsf{let}}{+}d)}\,(\text{E:LET1}) \qquad \frac{}{V, H \overset{M}{\vdash} e \Downarrow \circ \mid (0,0)}\,(\text{E:ABORT})$$

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow (\ell, H') \mid (w_1, d_1) \qquad V[x \mapsto \ell], H' \overset{M}{\vdash} e_2 \Downarrow \rho \mid (w_2, d_2)}{V, H \overset{M}{\vdash} \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow \rho \mid (M^{\mathsf{let}}{+}w_1{+}w_2, M^{\mathsf{let}}{+}d_1{+}d_2)}\,(\text{E:LET2})$$

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow \rho_1 \mid (w_1, d_1) \qquad V, H \overset{M}{\vdash} e_2 \Downarrow \rho_2 \mid (w_2, d_2) \qquad \rho_1{=}\circ \vee \rho_2{=}\circ}{V, H \overset{M}{\vdash} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \Downarrow \circ \mid (M^{\mathsf{Par}}{+}w_1{+}w_2, M^{\mathsf{Par}}{+}\max(d_1, d_2))}\,(\text{E:PAR1})$$

$$\text{(E:PAR2)}\quad \frac{\begin{array}{l} V, H \overset{M}{\vdash} e_1 \Downarrow (\ell_1, H_1) \mid (w_1, d_1) \qquad (w', d'){=}(M^{\mathsf{Par}}{+}w_1{+}w_2{+}w, M^{\mathsf{Par}}{+}\max(d_1, d_2){+}d) \\ V, H \overset{M}{\vdash} e_2 \Downarrow (\ell_2, H_2) \mid (w_2, d_2) \qquad V[x_1{\mapsto}\ell_1, x_2{\mapsto}\ell_2], H_1 \uplus H_2 \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d) \end{array}}{V, H' \overset{M}{\vdash} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \Downarrow (\ell, H') \mid (w', d')}$$

Fig. 1. Interesting rules of the operational big-step semantics.

Motivation. A distinctive feature of our big-step semantics is that it models terminating, failing, and diverging evaluations by inductively describing finite subtrees of (possibly infinite) evaluation trees. By using an inductive judgement for diverging and terminating computations while avoiding intermediate states, it combines the advantages of big-step and small-step semantics. This has two benefits compared to standard big-step semantics. First, we can model the resource consumption of diverging programs and prove that bounds hold for terminating and diverging programs. (In some cost metrics, diverging computations can have finite cost.) Second, for a cost metric in which all diverging computations have infinite cost we are able to show that bounds imply termination.

Note that we cannot achieve this by step-indexing a standard big-step semantics. The available alternatives to our approach are small-step semantics and coinductive big-step semantics. However, it is unclear how to prove the soundness of our type system with respect to these semantics. Small-step semantics is difficult to use because our type-system models an intentional property that goes beyond the classic type preservation: After performing a step, we have to obtain a refined typing that corresponds to a (possibly) smaller bound. Coinductive derivations are hard to relate to type derivations because type derivations are defined inductively.

Our inductive big-step semantics can not only be used to formalize resource cost of diverging computations but also for other effects such as event traces. It is therefore an interesting alternative to recently proposed coinductive operational big-step semantics [18].

Semantic Judgements. We formulate the big-step semantics with respect to a stack and a heap. Let *Loc* be an infinite set of *locations* modeling memory addresses on a heap. A value $v ::= n \mid (\ell_1, \ell_2) \mid (\mathsf{cons}, \ell_1, \ell_2) \mid \mathsf{nil} \in \mathit{Val}$ is either an integer $n \in \mathbb{Z}$, a pair of locations (ℓ_1, ℓ_2), a node $(\mathsf{cons}, \ell_1, \ell_2)$ of a list, or nil.

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. Thus we have boxed values. It is not important for the analysis whether values are boxed.

Figure 1 contains a compilation of the big-step evaluation rules (the full version contains all rules). They are formulated with respect to a resource metric M. They define the evaluation judgment

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

It expresses the following. In a fixed program $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$, if the stack V and the initial heap H are given then the expression e evaluates to ρ. Under the metric M, the work of the evaluation of e is w and the depth of the evaluation is d. Unlike standard big-step operational semantics, ρ can be either a pair of a location and a new heap, or $\circ$ (pronounced *busy*) indicating that the evaluation is not finished yet.

A resource metric $M : K \to \mathbb{Q}_0^+$ defines the resource consumption in each evaluation step of the big-step semantics with a non-negative rational number. We write M^k for $M(k)$.

An intuition for the judgement $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (w, d)$ is that there is a partial evaluation of e that runs without failure, has work w and depth d, and has not yet reached a value. This is similar to a small-step judgement.

Rules. For a heap H, we write $H, \ell \mapsto v$ to express that $\ell \notin \mathrm{dom}(H)$ and to denote the heap H' such that $H'(x) = H(x)$ if $x \in \mathrm{dom}(H)$ and $H'(\ell) = v$. In the rule E:Par2, we write $H_1 \uplus H_2$ to indicate that H_1 and H_2 agree on the values of locations in $\mathrm{dom}(H_1) \cap \mathrm{dom}(H_2)$ and to a combined heap H with $\mathrm{dom}(H) = \mathrm{dom}(H_1) \cup \mathrm{dom}(H_2)$. We assume that the locations that are allocated in parallel evaluations are disjoint. That is easily achievable in an implementation.

The most interesting rules of the semantics are E:Abort, and the rules for sequential and parallel composition. They allow us to approximate infinite evaluation trees for non-terminating evaluations with finite subtrees. The rule E:Abort states that we can partially evaluate every expression by doing zero steps. The work w and depth d are then both zero (i.e., $w = d = 0$).

To obtain an evaluation judgement for a sequential composition $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ we have two options. We can use the rule E:Let1 to partially evaluate e_1 using work w and depth d. Alternatively, we can use the rule E:Let2 to evaluate e_1 until we obtain a location and a heap (ℓ, H') using work w_1 and depth d_1. Then we evaluate e_2 using work w_2 and depth d_2. The total work and depth is then given by $M^{\mathsf{let}}{+}w_1{+}w_2$ and $M^{\mathsf{let}}{+}d_1{+}d_2$, respectively.

Similarly, we can derive evaluation judgements for a parallel composition $\mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e$ using the rules E:Par1 and E:Par2. In the rule E:Par1, we partially evaluate e_1 or e_2 with evaluation cost (w_1, d_1) and (w_2, d_2). The total work is then $M^{\mathsf{Par}}{+}w_1{+}w_2$ (the cost for the evaluation of the parallel binding plus the cost for the sequential evaluation of e_1 and e_2). The total depth is $M^{\mathsf{Par}} + \max(d_1, d_2)$ (the cost for the evaluation of the binding plus the maximum of the cost of the depths of e_1 and e_2). The rule E:Par2 handles the case in which e_1 and e_2 are fully evaluated. It is similar to E:Let2 and the cost of the evaluation of the expression e is added to both the cost and the depth since e is evaluated after e_1 and e_2.

2.3 Properties of the Cost-Semantics

The main theorem of this section states that the resource cost of a partial evaluation is less than or equal to the cost of an evaluation of the same expression that terminates.

Theorem 1. *If* $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ *and* $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (w', d')$ *then* $w' \leqslant w$ *and* $d' \leqslant d$.

Theorem 1 can be proved by a straightforward induction on the derivation of the judgement $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$.

Provably Efficient Implementations. While work is a realistic cost-model for the sequential execution of programs, depth is not a realistic cost-model for parallel execution. The main reason is that it assumes that an infinite number of processors can be used for parallel evaluation. However, it has been shown [10] that work and depth are closely related to the evaluation time on more realistic abstract machines.

For example, *Brent's Theorem* [12] provides an asymptotic bound on the number of execution steps on the shared-memory multiprocessor (SMP) machine. It states that if $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ then e can be evaluated on a p-processor SMP machine in time $O(\max(w/p, d))$. An SMP machine has a fixed number p of processes and provides constant-time access to a shared memory. The proof of Brent's Theorem can be seen as the description of a so-called *provably efficient implementation*, that is, an implementation for which we can establish an asymptotic bound that depends on the number of processors.

Classically, we are especially interested in non-asymptotic bounds in resource analysis. It would thus be interesting to develop a non-asymptotic version of Brent's Theorem for a specific architecture using more refined models of concurrency [11]. However, such a development is not in the scope of this article.

Well-Formed Environments and Type Soundness. For each data type A we inductively define a set $[\![A]\!]$ of values of type A. Lists are interpreted as lists and pairs are interpreted as pairs.

$$[\![\text{int}]\!] = \mathbb{Z} \qquad [\![A * B]\!] = [\![A]\!] \times [\![B]\!]$$
$$[\![L(A)]\!] = \{[a_1, \ldots, a_n] \mid n \in \mathbb{N}, a_i \in [\![A]\!]\}$$

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

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 $H \vDash V : \Gamma$.

Simple Metrics and Progress. In the reminder of this section, we prove a property of the evaluation judgement under a simple metric. A *simple metric* M assigns the value 1 to every resource constant, that is, $M(x) = 1$ for every $x \in K$. With a simple metric, work counts the number of evaluation steps.

Theorem 2 states that, in a well-formed environment, well-typed expressions either evaluate to a value or the evaluation uses unbounded work and depth.

Theorem 2 (Progress). *Let M be a simple metric, $\Sigma; \Gamma \vdash e : B$, and $H \models V : \Gamma$. Then $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ for some $w, d \in \mathbb{N}$ or for every $n \in \mathbb{N}$ there exist $x, y \in \mathbb{N}$ such that $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (x, n)$ and $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (n, y)$.*

A direct consequence of Theorem 2 is that bounds on the depth of programs under a simple metric ensure termination.

3 Amortized Analysis and Parallel Programs

In this section, we give a short introduction into amortized resource analysis for sequential programs (for bounding the work) and then informally describe the main contribution of the article: a multivariate amortized resource analysis for parallel programs (for bounding the depth).

Amortized Resource Analysis. Amortized resource analysis is a type-based technique for deriving upper bounds on the resource cost of programs [3]. The advantages of amortized resource analysis are compositionality and efficient type inference that is based on linear programming. The idea is that types are decorated with resource annotations that describe a potential function. Such a potential function maps the sizes of typed data structures to a non-negative rational number. The typing rules ensure that the potential defined by a typing context is sufficient to pay for the evaluation cost of the expression that is typed under this context and for the potential of the result of the evaluation.

The basic idea of amortized analysis is best explained by example. Consider the function $\mathsf{mult} : \mathrm{int} * \mathsf{L}(\mathrm{int}) \to \mathsf{L}(\mathrm{int})$ that takes an integer and an integer list and multiplies each element of the list with the integer.

```
mult(x,ys) = match ys with | nil  → nil
                           | (y::ys') → x*y::mult(x,ys')
```

For simplicity, we assume a metric M^* that only counts the number of multiplications performed in an evaluation in this section. Then $V, H \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) \Downarrow (\ell, H') \mid (n, n)$ for a well-formed stack V and heap H in which ys points to a list of length n. In short, the work and depth of the evaluation of $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$ is $|\mathsf{ys}|$.

To obtain a bound on the work in type-based amortized resource analysis, we derive a type of the following form.

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) : (L(\mathrm{int}), Q')$$

Here Q and Q' are *coefficients* of multivariate resource polynomials $p_Q : [\![\mathrm{int} * L(\mathrm{int})]\!] \to \mathbb{Q}_0^+$ and $p_{Q'} : [\![L(\mathrm{int})]\!] \to \mathbb{Q}_0^+$ that map semantic values to non-negative rational numbers. The rules of the type system ensure that for every evaluation context (V, H) that maps x to a number m and ys to a list a, the potential $p_Q(m, a)$ is sufficient to cover the evaluation cost of $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$ and the potential $p_{Q'}(a')$ of the returned list a'. More formally, we have $p_Q(m, a) \geqslant w + p_{Q'}(a')$ if $V, H \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) \Downarrow (\ell, H') \mid (w, d)$ and ℓ points to the list a' in H'.

In our type system we can for instance derive coefficients Q and Q' that represent the potential functions

$$p_Q(n, a) = |a| \qquad \text{and} \qquad p_{Q'}(a) = 0\ .$$

The intuitive meaning is that we must have the potential $|\mathsf{ys}|$ available when evaluating $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$. During the evaluation, the potential is used to pay for the evaluation cost and we have no potential left after the evaluation.

To enable compositionality, we also have to be able to pass potential to the result of an evaluation. Another possible instantiation of Q and Q' would for example result in the following potential.

$$p_Q(n, a) = 2{\cdot}|a| \qquad \text{and} \qquad p_{Q'}(a) = |a|$$

The resulting typing can be read as follows. To evaluate $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$ we need the potential $2|\mathsf{ys}|$ to pay for the cost of the evaluation. After the evaluation there is the potential $|\mathsf{mult}(\mathsf{x}, \mathsf{ys})|$ left to pay for future cost in a surrounding program. Such an instantiation would be needed to type the inner function application in the expression $\mathsf{mult}(\mathsf{x}, \mathsf{mult}(\mathsf{z}, \mathsf{ys}))$.

Technically, the coefficients Q and Q' are families that are indexed by sets of base polynomials. The set of base polynomials is determined by the type of the corresponding data. For the type $\mathrm{int} * L(\mathrm{int})$, we have for example $Q = \{q_{(*,[])}, q_{(*,[*])}, q_{(*,[*,*])}, \ldots\}$ and $p_Q(n, a) = q_{(*,[])} + q_{(*,[*])}{\cdot}|a| + q_{(*,[*,*])}{\cdot}\binom{|a|}{2} + \ldots$. This allows us to express multivariate functions such as $m \cdot n$.

The rules of our type system show how to describe the valid instantiations of the coefficients Q and Q' with a set of linear inequalities. As a result, we can use linear programming to infer resource bounds efficiently.

A more in-depth discussion can be found in the literature [3, 19, 7].

Sequential Composition. In a sequential composition $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$, the initial potential, defined by a context and a corresponding annotation (Γ, Q), has to be used to pay for the work of the evaluation of e_1 and the work of the evaluation of e_2. Let us consider a concrete example again.

```
mult2(ys) = let xs = mult(496,ys) in
            let zs = mult(8128,ys) in (xs,zs)
```

The work (and depth) of the evaluation of the expression $\mathsf{mult2}(\mathsf{ys})$ is $2|\mathsf{ys}|$ in the metric M^*. In the type judgement, we express this bound as follows. First, we type the two function applications of mult as before using

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \vdash^{M^*} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) : (L(\mathrm{int}), Q')$$

where $p_Q(n, a) = |a|$ and $p_{Q'}(a) = 0$. In the type judgement

$$\mathsf{ys}{:}L(\mathrm{int}); R \vdash^{M^*} \mathsf{mult2}(\mathsf{ys}) : (L(\mathrm{int}) * L(\mathrm{int}), R')$$

we require that $p_R(a) \geqslant p_Q(a) + p_Q(a)$, that is, the initial potential (defined by the coefficients R) has to be shared in the two sequential branches. Such a sharing can still be expressed with linear constraints. such as $r_{[*]} \geqslant q_{(*,[*])} + q_{(*,[*])}$. A valid instantiation of R would thus correspond to the potential function $p_R(a) = 2|a|$. With this instantiation, the previous typing reflects the bound $2|\mathsf{ys}|$ for the evaluation of $\mathsf{mult2}(\mathsf{ys})$.

A slightly more involved example is the function $\mathsf{dyad} : L(\mathrm{int}) * L(\mathrm{int}) \to L(L(\mathrm{int}))$ which computes the dyadic product of two integer lists.

```
dyad (u,v) = match u with | nil  → nil
     | (x::xs)  → let x' = mult(x,v) in
                  let xs' = dyad(xs,v) in x'::xs';
```

Using the metric M^* that counts multiplications, multivariate resource analysis for sequential programs derives the bound $|\mathsf{u}|\cdot|\mathsf{v}|$. In the cons branch of the pattern match, we have the potential $|\mathsf{xs}|\cdot|\mathsf{v}| + |\mathsf{v}|$ which is shared to pay for the cost $|\mathsf{v}|$ of $\mathsf{mult}(\mathsf{x},\mathsf{v})$ and the cost $|\mathsf{xs}|\cdot|\mathsf{v}|$ of $\mathsf{dyad}(\mathsf{xs},\mathsf{v})$.

Moving multivariate potential through a program is not trivial; especially in the presence of nested data structures like trees of lists. To give an idea of the challenges, consider the expression e that is defined as follows.

```
let xs = mult(496,ys) in
let zs = append(ys,ys) in dyad(xs,zs)
```

The depth of evaluating e in the metric M^* is bounded by $|\mathsf{ys}| + 2|\mathsf{ys}|^2$. Like in the previous example, we express this in amortized resource analysis with the initial potential $|\mathsf{ys}| + 2|\mathsf{ys}|^2$. This potential has to be shared to pay for the cost of the evaluations of $\mathsf{mult}(\mathsf{496},\mathsf{ys})$ (namely $|\mathsf{ys}|$) and $\mathsf{dyad}(\mathsf{xs},\mathsf{zs})$ (namely $2|\mathsf{ys}|^2$). However, the type of dyad requires the quadratic potential $|\mathsf{xs}|\cdot|\mathsf{zs}|$. In this simple example, it is easy to see that $|\mathsf{xs}|\cdot|\mathsf{zs}| = 2|\mathsf{ys}|^2$. But in general, it is not straightforward to compute such a conversion of potential in an automatic analysis system, especially for nested data structures and super-linear size changes. The type inference for multivariate amortized resource analysis for sequential programs can analyze such programs efficiently [7].

Parallel Composition. The insight of this paper is that the potential method works also well to derive bounds on parallel evaluations. The main challenge in the development of an amortized resource analysis for parallel evaluations is to ensure the same compositionality as in sequential amortized resource analysis.

The basic idea of our new analysis system is to allow each branch in a parallel evaluation to use all the available potential without sharing. Consider for example the previously defined function mult2 in which we evaluate the two applications of mult in parallel.

```
mult2par(ys) = par xs = mult(496,ys)
               and zs = mult(8128,ys) in (xs,zs)
```

Since the depth of $\mathsf{mult}(\mathsf{n},\mathsf{ys})$ is $|\mathsf{ys}|$ for every n and the two applications of mult are evaluated in parallel, the depth of the evaluation of $\mathsf{mult2par}(\mathsf{ys})$ is $|\mathsf{ys}|$ in the metric M^*.

In the type judgement, we type the two function applications of mult as in the sequential case in which

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \mid\!\!\!\overset{M^*}{-}\; \mathsf{mult}(\mathsf{x},\mathsf{ys}) : (L(\mathrm{int}), Q')$$

such that $p_Q(n,a) = |a|$ and $p_{Q'}(a) = 0$. In the type judgement

$$\mathsf{ys}{:}L(\mathrm{int}); R \mid\!\!\!\overset{M^*}{-}\; \mathsf{mult2par}(\mathsf{ys}) : (L(\mathrm{int}) * L(\mathrm{int}), R')$$

for mult2par we require however only that $p_R(a) \geqslant p_Q(a)$. In this way, we express that the initial potential defined by the coefficients R has to be sufficient to

cover the cost of each parallel branch. Consequently, a possible instantiation of R corresponds to the potential function $p_R(a) = |a|$.

In the function dyad, we can replace the sequential computation of the inner lists of the result by a parallel computation in which we perform all calls to the function mult in parallel. The resulting function is dyad_par.

```
dyad_par (u,v) = match u with | nil  → nil
                 | (x::xs)  → par x' = mult(x,v)
                              and xs' = dyad_par(xs,v) in x'::xs';
```

The depth of dyad_par is |v|. In the type-based amortized analysis, we hence start with the initial potential |v|. In the cons branch of the pattern match, we can use the initial potential to pay for both, the cost |v| of mult(x, v) and the cost |v| of the recursive call dyad(xs, v) without sharing the initial potential.

Unfortunately, the compositionality of the sequential system is not preserved by this simple idea. The problem is that the naive reuse of potential that is passed through parallel branches would break the soundness of the system. To see why, consider the following function.

```
mult4(ys) = par xs = mult(496,ys)
  and zs = mult(8128,ys) in (mult(5,xs), mult(10,zs))
```

Recall, that a valid typing for xs = mult(496, ys) could take the initial potential 2|ys| and assign the potential |xs| to the result. If we would simply reuse the potential 2|ys| to type the second application of mult in the same way then we would have the potential |xs| + |zs| after the parallel branches. This potential could then be used to pay for the cost of the remaining two applications of mult. We have now verified the unsound bound 2|ys| on the depth of the evaluation of the expression mult4(ys) but the depth of the evaluation is 3|ys|.

The problem in the previous reasoning is that we doubled the part of the initial potential that we passed on for later use in the two parallel branches of the parallel composition. To fix this problem, we need a separate analysis of the sizes of data structures and the cost of parallel evaluations.

In this paper, we propose to use cost-free type judgements to reason about the size changes in parallel branches. Instead of simply using the initial potential in both parallel branches, we share the potential between the two branches but analyze the two branches twice. In the first analysis, we only pay for the resource consumption of the first branch. In the second, analysis we only pay for resource consumption of the second branch.

A cost-free type judgement is like any other type judgement in amortized resource analysis but uses the cost-free metric cf that assigns zero cost to every evaluation step. For example, a cost-free typing of the function mult(ys) would express that the initial potential can be passed to the result of the function. In the cost-free typing judgement

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \vdash^{\mathsf{cf}} \mathsf{mult}(\mathsf{x},\mathsf{ys}) : (L(\mathrm{int}), Q')$$

a valid instantiation of Q and Q' would correspond to the potential

$$p_Q(n,a) = |a| \qquad \text{and} \qquad p_{Q'}(a) = |a| \ .$$

The intuitive meaning is that in a call zs = mult(x, ys), the initial potential $|ys|$ can be transformed to the potential $|zs|$ of the result.

Using cost-free typings, we can now correctly reason about the depth of the evaluation of mult4. We start with the initial potential $3|\mathsf{ys}|$ and have to consider two cases in the parallel binding. In the first case, we have to pay only for resource cost of $\mathsf{mult}(\mathsf{496}, \mathsf{ys})$. So we share the initial potential and use $2|\mathsf{ys}|$: $|\mathsf{ys}|$ to pay the cost of $\mathsf{mult}(\mathsf{496}, \mathsf{ys})$ and $|\mathsf{ys}|$ to assign the potential $|\mathsf{xs}|$ to the result of the application. The reminder $|\mathsf{ys}|$ of the initial potential is used in a cost-free typing of $\mathsf{mult}(\mathsf{8128}, \mathsf{ys})$ where we assign the potential $|\mathsf{zs}|$ to the result of the function without paying any evaluation cost. In the second case, we derive a similar typing in which the roles of the two function calls are switched. In both cases, we start with the potential $3|\mathsf{ys}|$ and end with the potential $|\mathsf{xs}| + |\mathsf{zs}|$. We use it to pay for the two remaining calls of mult and have verified the correct bound.

In the univariate case, using the notation from [3, 19], we could formulate the type rule for parallel composition as follows. Here, the coefficients Q are not globally attached to a type or context but appear locally at list types such as $L^q(\mathrm{int})$. The sharing operator $\Gamma \curlyvee (\Gamma_1, \Gamma_2, \Gamma_3)$ requires the sharing of the potential in the context Γ in the contexts Γ_1,Γ_2 and Γ_3. For instance, we have $x{:}L^6(\mathrm{int}) \curlyvee (x{:}L^2(\mathrm{int}), x{:}L^3(\mathrm{int}), x{:}L^1(\mathrm{int}))$.

$$\frac{\begin{array}{c}\Gamma \curlyvee (\Delta_1, \Gamma_2, \Gamma') \qquad \Gamma \curlyvee (\Gamma_1, \Delta_2, \Gamma') \qquad \Gamma_1 \vdash^{M} e_1 : A_1 \qquad \Delta_2 \vdash^{\mathsf{cf}} e_2 : A_2 \\ \Delta_1 \vdash^{\mathsf{cf}} e_1 : A_1 \qquad \Gamma_2 \vdash^{M} e_2 : A_2 \qquad \Gamma', x_1{:}A_1, x_2{:}A_2 \vdash^{M} e : B\end{array}}{\Gamma \vdash^{M} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e : B}$$

In the rule, the initial potential Γ is shared twice using the sharing operator $\curlyvee$. First, to pay the cost of evaluating e_2 and e, and to pass potential to x_1 using the cost-free type judgement $\Delta_1 \vdash^{\mathsf{cf}} e_1 : A_1$. Second, to pay the cost of evaluation e_1 and e, and to pass potential to x_2 via the judgement $\Delta_2 \vdash^{\mathsf{cf}} e_2 : A_2$.

This work generalizes the idea to multivariate resource polynomials for which we also have to deal with mixed potential such as $|x_1|\cdot|x_2|$. The approach features the same compositionality as the sequential version of the analysis. As the experiments in Section 7 show, the analysis works well for many typical examples.

The use of cost-free typings to separate the reasoning about size changes of data structures and resource cost in amortized analysis has applications that go beyond parallel evaluations. Similar problems arise in sequential (and parallel) programs when deriving bounds for non-additive cost such as stack-space usage or recursion depth. We envision that the developed technique can be used to derive bounds for these cost measures too.

Other Forms of Parallelism. The binary parallel binding is a simple yet powerful form of parallelism. However, it is (for example) not possible to directly implement NESL's model of sequences that allows to perform an operation for every element in the sequence in constant depth. The reason is that the parallel binding would introduce a linear overhead.

Nevertheless it is possible to introduce another binary parallel binding that is semantically equivalent except that it has zero depth cost. We can then analyze more powerful parallelism primitives by translating them into code that uses this cost-free parallel binding. To demonstrate such a translation, we implemented NESL's [15] parallel sequence comprehensions in RAML (see Section 6).

4 Resource Polynomials and Annotated Types

In this section, we introduce multivariate resource polynomials and annotated types. Our goal is to systematically describe the potential functions that map data structures to non-negative rational numbers. Multivariate resource polynomials are a generalization of non-negative linear combinations of binomial coefficients. They have properties that make them ideal for the generation of succinct linear constraint systems in an automatic amortized analysis. The presentation might appear quite low level but this level of detail is necessary to describe the linear constraints in the type rules.

Two main advantages of resource polynomials are that they can express more precise bounds than non-negative linear-combinations of standard polynomials and that they can succinctly describe common size changes of data that appear in construction and destruction of data. More explanations can be found in the previous literature on multivariate amortized resource analysis [13, 7].

4.1 Resource Polynomials

A resource polynomial maps a value of some data type to a nonnegative rational number. Potential functions and thus resource bounds are always resource polynomials.

Base 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. These *base polynomials* form a basis (in the sense of linear algebra) of the resource polynomials for type A. The resource polynomials for type A are then given as nonnegative rational linear combinations of the base polynomials. We define $P(A)$ as follows.

$$P(\text{int}) = \{a \mapsto 1\} \qquad P(A_1 * A_2) = \{(a_1, a_2) \mapsto p_1(a_1) \cdot p_2(a_2) \mid p_i \in P(A_i)\}$$
$$P(L(A)) = \{\Sigma\Pi[p_1, \ldots, p_k] \mid k \in \mathbb{N}, p_i \in P(A)\}$$

We have $\Sigma\Pi[p_1, \ldots, p_k]([a_1, \ldots, a_n]) = \sum_{1 \leqslant j_1 < \cdots < j_k \leqslant n} \prod_{1 \leqslant i \leqslant k} p_i(a_{j_i})$. Every set $P(A)$ contains the constant function $v \mapsto 1$. For lists $L(A)$ this arises for $k = 0$ (one element sum, empty product).

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

Resource Polynomials. A *resource polynomial* $p : [\![A]\!] \to \mathbb{Q}_0^+$ for a data type A is a non-negative linear combination of base polynomials, i.e., $p = \sum_{i=1,\ldots,m} q_i \cdot p_i$ for $q_i \in \mathbb{Q}_0^+$ and $p_i \in P(A)$. $R(A)$ is the set of resource polynomials for A.

An instructive, but not exhaustive, example is given by $R_n = R(L(\text{int}) * \cdots * L(\text{int}))$. The set R_n is the set of linear combinations of products of binomial coefficients over variables $x_1, \ldots, x_n$, that is, $R_n = \{\sum_{i=1}^{m} q_i \prod_{j=1}^{n} \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$. Concrete examples that illustrate the definitions follow in the next subsection.

4.2 Annotated Types

To relate type annotations in the type system to resource polynomials, we introduce names (or indices) for base polynomials. These names are also helpful to intuitively explain the base polynomials of a given type.

Names For Base Polynomials. To assign a unique name to each base polynomial we define the *index set* $\mathcal{I}(A)$ to denote resource polynomials for a given data type A. Essentially, $\mathcal{I}(A)$ is the meaning of A with every atomic type replaced by the *unit index* $\circ$.

$$\mathcal{I}(\text{int}) = \{\circ\} \qquad \mathcal{I}(A_1 * A_2) = \{(i_1, i_2) \mid i_1 \in \mathcal{I}(A_1) \text{ and } i_2 \in \mathcal{I}(A_2)\}$$
$$\mathcal{I}(L(A)) = \{[i_1, \dots, i_k] \mid k \geqslant 0, i_j \in \mathcal{I}(A)\}$$

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

$$\deg(\circ) = 0 \qquad \deg(i_1, i_2) = \deg(i_1) + \deg(i_2)$$
$$\deg([i_1, \dots, i_k]) = k + \deg(i_1) + \dots + \deg(i_k)$$

Let $\mathcal{I}_k(A) = \{i \in \mathcal{I}(A) \mid \deg(i) \leqslant k\}$. The indices $i \in \mathcal{I}_k(A)$ are an enumeration of the base polyonomials $p_i \in P(A)$ of degree at most k. For each $i \in \mathcal{I}(A)$, we define a base polynomial $p_i \in P(A)$ as follows: If $A = \text{int}$ then $p_\circ(v) = 1$. If $A = (A_1 * A_2)$ is a pair type and $v = (v_1, v_2)$ then $p_{(i_1,i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2)$. If $A = L(B)$ is a list type and $v \in [\![L(B)]\!]$ then $p_{[i_1,\dots,i_m]}(v) = \Sigma\Pi[p_{i_1}, \dots, p_{i_m}](v)$. We use the notation 0_A (or just 0) for the index in $\mathcal{I}(A)$ such that $p_{0_A}(a) = 1$ for all a. We have $0_{\text{int}} = \circ$ and $0_{(A_1 * A_2)} = (0_{A_1}, 0_{A_2})$ and $0_{L(B)} = []$. If $A = L(B)$ for a data type B then the index $[0, \dots, 0] \in \mathcal{I}(A)$ of length n is denoted by just n. We identify the index (i_1, i_2, i_3, i_4) with the index $(i_1, (i_2, (i_3, i_4)))$.

Examples. First consider the type int. The index set $\mathcal{I}(\text{int}) = \{\circ\}$ only contains the unit element because the only base polynomial for the type int is the constant polynomial $p_\circ : \mathbb{Z} \to \mathbb{N}$ that maps every integer to 1, that is, $p_\circ(n) = 1$ for all $n \in \mathbb{Z}$. In terms of resource-cost analysis this implies that the resource polynomials can not represent cost that depends on the value of an integer.

Now consider the type $L(\text{int})$. The index set for lists of integers is $\mathcal{I}(L(\text{int})) = \{[], [\circ], [\circ, \circ], \dots\}$, the set of lists of unit indices $\circ$. The base polynomial $p_{[]} : [\![L(\text{int})]\!] \to \mathbb{N}$ is defined as $p_{[]}([a_1, \dots, a_n]) = 1$ (one element sum and empty product). More interestingly, we have $p_{[\circ]}([a_1, \dots, a_n]) = \sum_{1 \leqslant j \leqslant n} 1 = n$ and $p_{[\circ,\circ]}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < j_2 \leqslant n} 1 = \binom{n}{2}$. In general, if $i_k = [\circ, \dots, \circ]$ is as list with k unit indices then $p_{i_k}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < \dots < j_k \leqslant n} 1 = \binom{n}{k}$. The intuition is that the base polynomial $p_{i_k}([a_1, \dots, a_n])$ describes a constant resource cost that arises for every ordered k-tuple $(a_{j_1}, \dots, a_{j_n})$.

Finally, consider the type $L(L(\text{int}))$ of lists of lists of integers. The corresponding index set is $\mathcal{I}(L(L(\text{int}))) = \{[]\} \cup \{[i] \mid i \in \mathcal{I}(L(\text{int}))\} \cup \{[i_1, i_2] \mid i_1, i_2 \in \mathcal{I}(L(\text{int}))\} \cup \dots$. Again we have $p_{[]} : [\![L(L(\text{int}))]\!] \to \mathbb{N}$ and $p_{[]}([a_1, \dots, a_n]) = 1$. Moreover we also get the binomial coefficients again: If the index $i_k = [[], \dots, []]$ is as list of k empty lists then $p_{i_k}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < \dots < j_k \leqslant n} 1 = \binom{n}{k}$. This describes a cost that would arise in a program that computes something of constant cost for tuples of inner lists (e.g., sorting with respect to the smallest head elements). However, the base polynomials can also refer to the lengths of the inner

lists. For instance, we have $p[[\circ,\circ]]([a_1,\ldots,a_n]) = \sum_{1\leqslant i\leqslant n}\binom{|a_i|}{2}$, which represents a quadratic cost for every inner list (e.g, sorting the inner lists). This is not to be confused with the base polynomial $p_{[\circ,\circ]}([a_1,\ldots,a_n]) = \sum_{1\leqslant i<j\leqslant n}|a_i||a_j|$, which can be used to account for the cost of the comparisons in a lexicographic sorting of the outer list.

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

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

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

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

Let H be a heap and let ℓ be a location with $H \vDash \ell{\mapsto}a : A$ for a data type A. Then the type annotation Q_A defines the *potential* $\Phi_H(\ell{:}(A, Q_A)) = \sum_{i\in\mathcal{I}(A)} q_i \cdot p_i(a)$. If $a \in [\![A]\!]$ and Q is a type annotation for A then we also write $\Phi(a : (A, Q))$ for $\sum_i q_i p_i(a)$.

Let for example, $Q = (q_i)_{i\in L(\mathrm{int})}$ be an annotation for the type $L(\mathrm{int})$ and let $q_{[]} = 2$, $q_{[\circ]} = 2.5$, $q_{[\circ,\circ,\circ]} = 8$, and $q_i = 0$ for all other $i \in \mathcal{I}(L(\mathrm{int}))$. The we have $\Phi([a_1,\ldots,a_n] : (L(\mathrm{int}), Q)) = 2 + 2.5n + 8\binom{n}{3}$.

The Potential of a Context. For use in the type system we need to extend the definition of resource polynomials to typing contexts. We treat a context like a tuple type. Let $\Gamma = x_1{:}A_1,\ldots,x_n{:}A_n$ be a typing context and let $k \in \mathbb{N}$. The index set $\mathcal{I}(\Gamma)$ is defined through $\mathcal{I}(\Gamma) = \{(i_1,\ldots,i_n) \mid i_j \in \mathcal{I}(A_j)\}$.

The degree of $i = (i_1,\ldots,i_n) \in \mathcal{I}(\Gamma)$ is defined through $\deg(i) = \deg(i_1) + \cdots + \deg(i_n)$. As for data types, we define $I_k(\Gamma) = \{i \in I(\Gamma) \mid \deg(i) \leqslant k\}$. A *type annotation* Q for Γ is a family $Q = (q_i)_{i\in\mathcal{I}_k(\Gamma)}$ with $q_i \in \mathbb{Q}_0^+$. We denote a *resource-annotated context* with $\Gamma; Q$. Let H be a heap and V be a stack with $H \vDash V : \Gamma$ where $H \vDash V(x_j){\mapsto}a_{x_j} : \Gamma(x_j)$.

The potential of an annotated context $\Gamma; Q$ with respect to then environment H and V is $\Phi_{V,H}(\Gamma; Q) = \sum_{(i_1,\ldots,i_n)\in\mathcal{I}_k(\Gamma)} q_{\vec{i}} \prod_{j=1}^n p_{i_j}(a_{x_j})$. In particular, if $\Gamma = \varnothing$ then $\mathcal{I}_k(\Gamma) = \{()\}$ and $\Phi_{V,H}(\Gamma; q_{()}) = q_{()}$. We sometimes also write q_0 for $q_{()}$.

5 Type System for Bounds on the Depth

In this section, we formally describe the novel resource-aware type system. We focus on the type judgement and explain the rules that are most important for handling parallel evaluation. The full type system is given in the extended version of this article [17].

The main theorem of this section proves the soundness of the type system with respect to the depths of evaluations as defined by the operational big-step semantics. The soundness holds for terminating and non-terminating evaluations.

Type Judgments. The typing rules in Figure 2 define a *resource-annotated typing judgment* of the form

$$\Sigma; \Gamma; \{Q_1,\ldots,Q_n\} \vdash^M e : (A, Q')$$

where M is a metric, $n \in \{1, 2\}$, e is an expression, Σ is a resource-annotated signature (see below), $(\Gamma; Q_i)$ is a resource-annotated context for every $i \in \{1, \ldots, n\}$, and (A, Q') is a resource-annotated data type. The intended meaning of this judgment is the following. If there are more than $\Phi(\Gamma; Q_i)$ resource units available for every $i \in \{1, \ldots, n\}$ then this is sufficient to pay for the depth of the evaluation of e under the metric M. In addition, there are more than $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

In outermost judgements, we are only interested in the case where $n = 1$ and the judgement is equivalent to the similar judgement for sequential programs [7]. The form in which $n = 2$ is introduced in the type rule E:Par for parallel bindings and eliminated by multiple applications of the sharing rule E:Share (more explanations follow).

The type judgement is affine in the sense that every variable in a context Γ can be used at most once in the expression e. Of course, we have to also deal with expressions in which a variable occurs more than once. To account for multiple variable uses we use the sharing rule T:Share that doubles a variable in a context without increasing the potential of the context.

As usual Γ_1, Γ_2 denotes the union of the contexts Γ_1 and Γ_2 provided that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$. We thus have the implicit side condition $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$ whenever Γ_1, Γ_2 occurs in a typing rule. Especially, writing $\Gamma = x_1{:}A_1, \ldots, x_k{:}A_k$ means that the variables x_i are pairwise distinct.

Programs with Annotated Types. *Resource-annotated first-order types* have the form $(A, Q) \to (B, Q')$ for annotated data types (A, Q) and (B, Q'). A *resource-annotated signature* Σ is a finite, partial mapping of function identifiers to *sets of* resource-annotated first-order types. A program with resource-annotated types for the metric M consists of a resource-annotated signature Σ and a family of expressions with variables identifiers $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ such that $\Sigma; y_f{:}A; Q \overset{M}{\vdash} e_f : (B, Q')$ for every function type $(A, Q) \to (B, Q') \in \Sigma(f)$.

Sharing. Let $\Gamma, x_1{:}A, x_2{:}A; Q$ be an annotated context. The *sharing operation* $\curlyvee Q$ defines an annotation for a context of the form $\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable. Details can be found in the full version of the article.

Typing Rules. Figure 2 shows the annotated typing rules that are most relevant for parallel evaluation. Most of the other rules are similar to the rules for multivariate amortized analysis for sequential programs [13, 20]. The main difference it that the rules here operate on annotations that are singleton sets $\{Q\}$ instead of the usual context annotations Q.

In the rules T:Let and T:Par, the result of the evaluation of an expression e is bound 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 e.

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

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

$$\frac{\Sigma;\Gamma_1,\Gamma_2;R \mid\!\!\!\frac{M}{}\ e_1 \rightsquigarrow \Gamma_2, x{:}A; R' \qquad \Sigma;,\Gamma_2,x{:}A;\{R'\} \mid\!\!\!\frac{M}{}\ e_2 : (B,Q') \qquad Q = R + M^{\mathsf{let}}}{\Sigma;\Gamma_1,\Gamma_2;\{Q\} \mid\!\!\!\frac{M}{}\ \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : (B,Q')}\ (\text{T:Let})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma_1,\Gamma_2,\Delta;P \mid\!\!\!\frac{\mathsf{cf}}{}\ e_1 \rightsquigarrow \Gamma_2,\Delta,x_1{:}A_1;P' \\ \Sigma;\Gamma_2,\Delta,x_1{:}A_1;P' \mid\!\!\!\frac{M}{}\ e_2 \rightsquigarrow \Delta,x_1{:}A_1,x_2{:}A_2;R \\ \Sigma;\Gamma_2,\Delta,x_1{:}A_1;Q' \mid\!\!\!\frac{\mathsf{cf}}{}\ e_2 \rightsquigarrow \Delta,x_1{:}A_1,x_2{:}A_2;R \\ \Sigma;\Gamma_1,\Gamma_2,\Delta;Q \mid\!\!\!\frac{M}{} e_1 \rightsquigarrow \Gamma_2,\Delta,x_1{:}A_1;Q' \qquad \Sigma;\Delta,x_1{:}A_1,x_2{:}A_2;R \mid\!\!\!\frac{M}{} e : (B,R')\end{array}}{\Sigma;\Gamma_1,\Gamma_2,\Delta;\{Q+M^{\mathsf{Par}},P+M^{\mathsf{Par}}\} \mid\!\!\!\frac{M}{}\ \mathsf{par}\, x_1 = e_1\, \mathsf{and}\, x_2 = e_2\, \mathsf{in}\, e : (B,R')}\ (\text{T:Par})$$

$$\frac{\Sigma;\Gamma,x_1{:}A,x_2{:}A;\{P_1,\ldots,P_m\} \mid\!\!\!\frac{M}{}\ e : (B,Q') \qquad \forall i\, \exists j : Q_j = \curlyvee P_i}{\Sigma;\Gamma,x{:}A;\{Q_1,\ldots,Q_n\} \mid\!\!\!\frac{M}{}\ e[x/x_1,x/x_2] : (B,Q')}\ (\text{T:Share})$$

◆ ◆ ◆

$$\frac{\begin{array}{c}\forall j \in \mathcal{I}(\Delta): \quad j{=}\vec{0} \implies \Sigma;\Gamma;\pi_j^{\Gamma}(Q) \mid\!\!\!\frac{M}{}\ e : (A,\pi_j^{x:A}(Q')) \\ j{\neq}\vec{0} \implies \Sigma_j;\Gamma;\pi_j^{\Gamma}(Q) \mid\!\!\!\frac{\mathsf{cf}}{}\ e : (A,\pi_j^{x:A}(Q'))\end{array}}{\Sigma;\Gamma,\Delta;Q \mid\!\!\!\frac{M}{}\ e \rightsquigarrow \Delta,x{:}A;Q'}\ (\text{B:Bind})$$

Fig. 2. Selected novel typing rules for annotated types and the binding rule for multivariate variable binding.

in the rule B:Bind. The intuitive meaning of the judgement is the following. Assume that e is evaluated in the context Γ,Δ, that $\mathrm{FV}(e) \in \mathrm{dom}(\Gamma)$, 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 $\Phi(\Delta,x{:}A;Q')$.

The rule T:Par for parallel bindings $\mathsf{par}\, x_1 = e_1\, \mathsf{and}\, x_2 = e_2\, \mathsf{in}\, e$ is the main novelty in the type system. The idea is that we type the expressions e_1 and e_2 twice using the new binding judgement. In the first group of bindings, we account for the cost of e_1 and derive a context $\Gamma_2,\Delta,x_1{:}A_1;P_1'$ in which the result of the evaluation of e_1 is bound to x_1. This context is then used to bind the result of evaluating e_2 in the context $\Delta,x_1{:}A_1,x_2{:}A_2;R$ without paying for the resource consumption. In the second group of bindings, we also derive the context $\Delta,x_1{:}A_1,x_2{:}A_2;R$ but pay for the cost of evaluating e_2 instead of e_1. The type annotations Q_1 and Q_2 for the initial context $\Gamma = \Gamma_1,\Gamma_2,\Delta$ establish a bound on the depth d of evaluating the whole parallel binding: If the depth of evaluating e_1 is larger than the depth of evaluating e_2 then $\Phi(\Gamma;Q_1) \geqslant d$. Otherwise we have $\Phi(\Gamma;Q_2) \geqslant d$. If the parallel binding evaluates to a value v then we have additionally that $\max(\Phi(\Gamma;Q_1),\Phi(\Gamma;Q_2)) \geqslant d + \Phi(v{:}(B,Q'))$.

It is important that the annotations Q_1 and Q_2 of the initial context Γ_1,Γ_2,Δ can defer. The reason is that we have to allow a different sharing of potential in the two groups of bindings. If we would require $Q_1 = Q_2$ then the system would be too restrictive. However, each type derivation has to establish the equality of the two annotations directly after the use of T:Par by multiple uses of the

sharing rule T:Share. Note that T:Par is the only rule that can introduce a non-singleton set $\{Q_1, Q_n\}$ of context annotations.

T:Share has to be applied to expressions that contain a variable twice (x in the rule). The sharing operation $\curlyvee P$ transfers the annotation P for the context $\Gamma, x_1{:}A, x_2{:}A$ into an annotation Q for the context $\Gamma, x{:}A$ without loss of potential . This is crucial for the accuracy of the analysis since instances of T:Share are quite frequent in typical examples. The remaining rules are affine in the sense that they assume that every variable occurs at most once in the typed expression.

T:Share is the only rule whose premiss allows judgements that contain a non-singleton set $\{P_1, \dots, P_m\}$ of context annotations. It has to be applied to produce a judgement with singleton set $\{Q\}$ before any of the other rules can be applied. The idea is that we always have $n \leqslant m$ for the set $\{Q_1, \dots, Q_n\}$ and the sharing operation $\curlyvee_i$ is used to unify the different P_i.

Soundness. The operational big-step semantics with partial evaluations makes it possible to state and prove a strong soundness result. An annotated type judgment for an expression e establishes a bound on the depth of all evaluations of e in a well-formed environment; regardless of whether these evaluations diverge or fail.Moreover, the soundness theorem states also 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 depth of the evaluation.

Theorem 3 (Soundness). *If $H \vDash V{:}\Gamma$ and $\Sigma; \Gamma; \mathcal{Q} \vdash e{:}(B, Q')$ then there exists a $Q \in \mathcal{Q}$ such that the following holds.*

1. *If $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ then $d \leqslant \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(\ell{:}(B, Q'))$.*
2. *If $V, H \overset{M}{\vdash} e \Downarrow \rho \mid (w, d)$ then $d \leqslant \Phi_{V,H}(\Gamma; Q)$.*

Theorem 3 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 very similar to the proof of the rules for multivariate resource analysis for sequential programs [7]. The main novelty is the treatment of parallel evaluation in the rule T:Par which we described previously.

If the metric M is simple (all constants are 1) then it follows from Theorem 3 that the bounds on the depth also prove the termination of programs.

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

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

the type rules. Forth, we solve the inequalities with an LP solver such as CLP. A solution of the linear program corresponds to a type derivation in which the variables in the type annotations are instantiated according to the solution.

In practice, the type inference is slightly more complex. Most importantly, we have to deal with resource-polymorphic recursion in many examples. This means that we need a type annotation in the recursive call that differs from the annotation in the argument and result types of the function. To infer such types we successively infer type annotations of higher and higher degree. Details can be found in previous work [21]. 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 [7]. Finally, we use several optimizations to reduce the number of generated constraints. See [7] for an example type derivation.

6 Nested Data Parallelism

The techniques that we describe in this work for a minimal function language scale to more advanced parallel languages such as Blelloch's NESL [15].

To describe the novel type analysis in this paper, we use a binary binding construct to introduce parallelism. In NESL, parallelism is introduced via built-in functions on sequences as well as parallel sequence comprehension that is similar to Haskell's list comprehension. The depth of all built-in sequence functions such as *append* and *sum* is constant in NESL. Similarly, the depth overhead of the parallel sequence comprehension is constant too. Of course, it is possible to define equivalent functions in RAML. However, the depth would often be linear since we, for instance, have to sequentially form the resulting list.

Nevertheless, the user definable resource metrics in RAML make it easy to introduce built-in functions and language constructs with customized work and depth. For instance we could implement NESL's *append* like the recursive append in RAML but use a metric inside the function body in which all evaluation steps have depth zero. Then the depth of the evaluation of $\mathsf{append}(\mathsf{x},\mathsf{y})$ is constant and the work is linear in $|x|$.

To demonstrate this ability of our approach, we implemented parallel list comprehensions, NESL's most powerful construct for parallel computations. A list comprehension has the form $\{\ e\ :\ x_1 \,\mathsf{in}\, e_1\,;\ldots;\, x_n \,\mathsf{in}\, e_n\ \mid\ e_b\ \}$. where e is an expression, $e_1,\ldots,e_n$ are expressions of some list type, and e_b is a boolean expression. The semantics is that we bind $x_1,\ldots,x_n$ successively to the elements of the lists $e_1,\ldots,e_n$ and evaluate e_b and e under these bindings. If e_b evaluates to true under a binding then we include the result of e under that binding in the resulting list. In other words, the above list comprehension is equivalent to the Haskell expression $[\ e \mid (x_1,\ldots,x_n) \leftarrow \mathsf{zip}_n\, e_1\ \ldots\ e_n\ ,\ e_b\ ]$.

The *work* of evaluating $\{\ e\ :\ x_1 \,\mathsf{in}\, e_1\,;\ldots;\, x_n \,\mathsf{in}\, e_n\ \mid\ e_b\ \}$ is sum of the cost of evaluating $e_1,\ldots,e_{n-1}$ and e_n plus the sum of the cost of evaluating e_b and e with the successive bindings to the elements of the results of the evaluation of $e_1,\ldots,e_n$. The *depth* of the evaluation is sum of the cost of evaluating $e_1,\ldots,e_{n-1}$ and e_n plus the maximum of the cost of evaluating e_b and e with the successive bindings to the elements of the results of the e_i.

Function Name / Function Type	Computed Depth Bound / Computed Work Bound	Run Time	Asym. Behav.
dyad	$10m + 10n + 3$	0.19 s	$O(n+m)$
$L(\text{int}) * L(\text{int}) \to L(L(\text{int}))$	$10mn + 17n + 3$	0.20 s	$O(nm)$
dyad_all	$1.\bar{6}n^3 - 4n^2 + 10nm + 14.\bar{6}n + 5$	1.66 s	$O(n^2+m)$
$L(L(\text{int})) \to L(L(L(\text{int})))$	$1.\bar{3}n^3 + 5n^2m^2 + 8.5n^2m + \ldots$	0.96 s	$O(n^3+n^2m^2)$
m_mult1	$15xy + 16x + 10n + 6$	0.37 s	$O(xy)$
$L(L(\text{int})) * L(L(\text{int})) \to L(L(\text{int}))$	$15xyn + 16nm + 18n + 3$	0.36 s	$O(xyn)$
m_mult_pairs $[M := L(L(\text{int}))]$	$4n^2 + 15nmx + 10nm + 10n + 3$	3.90 s	O(nm + mx)
$L(M) * L(M) \to L(M)$	$7.5n^2m^2x + 7n^2m^2 + n^2mx \ldots$	6.35 s	$O(n^2m^2x)$
m_mult2 $[M := L(L(\text{int}))]$	$35u + 10y + 15x + 11n + 40$	2.75 s	$O(z+x+n)$
$(M * \text{nat}) * (M * \text{nat}) \to M$	$3.5u^2y + uyz + 14.5uy + \ldots$	2.99 s	$O(nx(z+y))$
quicksort_list	$12n^2 + 16nm + 12n + 3$	0.67 s	$O(n^2+m)$
$L(L(\text{int})) \to L(L(\text{int}))$	$8n^2m + 15.5n^2 - 8nm + 13.5n + 3$	0.51 s	$O(n^2m)$
intersection	$10m + 12n + 3$	0.49 s	$O(n+m)$
$L(\text{int}) * L(\text{int}) \to L(\text{int})$	$10mn + 19n + 3$	0.28 s	$O(nm)$
product	$8mn + 10m + 14n + 3$	1.05 s	$O(nm)$
$L(\text{int}) * L(\text{int}) \to L(\text{int} * \text{int})$	$18mn + 21n + 3$	0.71 s	$O(nm)$
max_weight	$46n + 44$	0.39 s	$O(n)$
$L(\text{int}) \to \text{int} * L(\text{int})$	$13.5n^2 + 65.5n + 19$	0.30 s	$O(n^2)$
fib	$13n + 4$	0.09 s	$O(n)$
$\text{nat} * \text{nat} \to \text{nat}$	$- - -$	0.12 s	$O(2^n)$
dyad_comp	13	0.28 s	$O(1)$
$L(\text{int}) * L(\text{int}) \to L(L(\text{int}))$	$6mn + 5n + 2$	0.13 s	$O(nm)$
find	$12m + 29n + 22$	0.38 s	$O(m+n)$
$L(\text{int}) * L(\text{int}) \to L(L(\text{int}))$	$20mn + 18m + 9n + 16$	0.41 s	$O(nm)$

Table 1. Compilation of Computed Depth and Work Bounds.

7 Experimental Evaluation

We implemented the developed automatic depth analysis in Resource Aware ML (RAML). The implementation consists mainly of adding the syntactic form for the parallel binding and the parallel list comprehensions together with the treatment in the parser, the interpreter, and the resource-aware type system. RAML is publically available for download and through a user-friendly online interface [16]. On the project web page you also find the source code of all example programs and of RAML itself.

We used the implementation to perform an experimental evaluation of the analysis on typical examples from functional programming. In the compilation of our results we focus on examples that have a different asymptotic worst-case behavior in parallel and sequential evaluation. In many other cases, the worst-case behavior only differs in the constant factors. Also note that many of the classic examples of Blelloch [10]—like quick sort—have a better asymptotic average behavior in parallel evaluation but the same asymptotic worst-case behavior in parallel and sequential cost.

Table 1 contains a representative compilation of our experimental results. For each analyzed function, it shows the function type, the computed bounds on the work and the depth, the run time of the analysis in seconds and the actual asymptotic behavior of the function. The experiments were performed on an iMac with a 3.4 GHz Intel Core i7 and 8 GB memory. As LP solver we used IBM's

CPLEX and the constraint solving takes about 60% of the overall run time of the prototype on average. The computed bounds are simplified multivariate resource polynomials that are presented to the user by RAML. Note that RAML also outputs the (unsimplified) multivariate resource polynomials. The variables in the computed bounds correspond to the sizes of different parts of the input. As naming convention we use the order n, m, x, y, z, u of 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.

All bounds are asymptotically tight if the tight bound is representable by a multivariate resource polynomial. For example, the exponential work bound for fib and the logarithmic bounds for bitonic_sort are not representable as a resource polynomial. Another example is the loose depth bound for dyad_all where we would need the base function $\max_{1 \leqslant i \leqslant n} m_i$ but only have $\sum_{1 \leqslant i \leqslant n} m_i$.

Matrix Operations. To study programs that use nested data structures we implemented several matrix operations for matrices that are represented by lists of lists of integers. The implemented operations include, the dyadic product from Section 3 (dyad), transposition of matrices (transpose, see [16]), addition of matrices (m_add, see [16]), and multiplication of matrices (m_mult1 and m_mult2).

To demonstrate the compositionality of the analysis, we have implemented two more involved functions for matrices. The function dyad_all computes the dyadic product (using dyad) of all ordered pairs of the inner lists in the argument. The function m_mult_pairs computes the products $M_1 \cdot M_2$ (using m_mult1) of all pairs of matrices such that M_1 is in the first list of the argument and M_2 is in the second list of the argument.

Sorting Algorithms. The sorting algorithms that we implemented include quick sort and bitonic sort for lists of integers (quicksort and bitonic_sort, see [16]).

The analysis computes asymptotically tight quadratic bounds for the work and depth of quick sort. The asymptotically tight bounds for the work and depth of bitonic sort are $O(n \log n)$ and $O(n \log^2 n)$, respectively, and can thus not be expressed by polynomials. However, the analysis computes quadratic and cubic bounds that are asymptotically optimal if we only consider polynomial bounds.

More interesting are sorting algorithms for lists of lists, where the comparisons need linear instead of constant time. In these algorithms we can often perform the comparisons in parallel. For instance, the analysis computes asymptotically tight bounds for quick sort for lists of lists of integers (quicksort_list, see Table 1).

Set Operations. We implemented sets as unsorted lists without duplicates. Most list operations such as intersection (Table 1), difference (see [16]), and union (see [16]) have linear depth and quadratic work. The analysis finds these asymptotically tight bounds.

The function product computes the Cartesian product of two sets. Work and depth of product are both linear and the analysis finds asymptotically tight bounds. However, the constant factors in the parallel evaluation are much smaller.

Miscellaneous. The function max_weight (Table 1) computes the maximal weight of a (connected) sublist of an integer list. The weight of a list is simply the sum of its elements. The work of the algorithm is quadratic but the depth is linear.

Finally, there is a large class of programs that have non-polynomial work but polynomial depth. Since the analysis can only compute polynomial bounds we can only derive bounds on the depth for such programs. A simple example in Table 1 is the function fib that computes the Fibonacci numbers without memoization.

Parallel List Comprehensions. The aforementioned examples are all implemented without using parallel list comprehensions. Parallel list comprehensions have a better asymptotic behavior than semantically-equivalent recursive functions in RAML's current resource metric for evaluation steps.

A simple example is the function dyad_comp which is equivalent to dyad and which is implemented with the expression $\{\{x * y : y \text{ in } ys\} : x \text{ in } xs\}$. As listed in Table 1, the depth of dyad_comp is constant while the depth of dyad is linear. RAML computes tight bounds.

A more involved example is the function find that finds a given integer list (needle) in another list (haystack). It returns the starting indices of each occurrence of the needle in the haystack. The algorithm is described by Blelloch [15] and cleverly uses parallel list comprehensions to perform the search in parallel. RAML computes asymptotically tight bounds on the work and depth.

Discussion. Our experiments show that the range of the analysis is not reduced when deriving bounds on the depth: The prototype implementation can always infer bounds on the depth of a program if it can infer bounds on the sequential version of the program. The derivation of bounds for parallel programs is also almost as efficient as the derivation of bounds for sequential programs.

We experimentally compared the derived worst-case bounds with the measured work and depth of evaluations with different inputs. In most cases, the derived bounds on the depth are asymptotically tight and the constant factors are close or equal to the optimal ones. As a representative example, the full version of the article contains plots of our experiments for quick sort for lists of lists.

8 Related Work

Automatic amortized resource analysis was introduced by Hofmann and Jost for a strict first-order functional language [3]. The technique has been applied to higher-order functional programs [22], to derive stack-space bounds for functional programs [23], to functional programs with lazy evaluation [4], to object-oriented programs [24, 25], and to low-level code by integrating it with separation logic [26]. All the aforementioned amortized-analysis–based systems are limited to linear bounds. The polynomial potential functions that we use in this paper were introduced by Hoffmann et al. [19, 13, 7]. In contrast to this work, none of the previous works on amortized analysis considered parallel evaluation. The main technical innovation of this work is the new rule for parallel composition that is not straightforward. The smooth integration of this rule in the existing framework of multivariate amortized resource analysis is a main advantages of our work.

Type systems for inferring and verifying cost bounds for sequential programs have been extensively studied. Vasconcelos et al. [27, 1] described an automatic analysis system that is based on sized-types [28] and derives linear bounds for

higher-order sequential functional programs. Dal Lago et al. [29, 30] introduced linear dependent types to obtain a complete analysis system for the time complexity of the call-by-name and call-by-value lambda calculus. Crary and Weirich [31] presented a type system for specifying and certifying resource consumption. Danielsson [32] developed a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of functional programs. We are not aware of any type-based analysis systems for parallel evaluation.

Classically, cost analyses are often based on deriving and solving recurrence relations. This approach was pioneered by Wegbreit [33] and has been extensively studied for sequential programs written in imperative languages [6, 34] and functional languages [35, 2].

In comparison, there has been little work done on the analysis of parallel programs. Albert et al. [36] use recurrence relations to derive cost bounds for concurrent object-oriented programs. Their model of concurrent imperative programs that communicate over a shared memory and the used cost measure is however quite different from the depth of functional programs that we study.

The only article on using recurrence relations for deriving bounds on parallel functional programs that we are aware of is a technical report by Zimmermann [37]. The programs that were analyzed in this work are fairly simple and more involved programs such as sorting algorithms seem to be beyond its scope. Additionally, the technique does not provide the compositionality of amortized resource analysis.

Trinder et al. [38] give a survey of resource analysis techniques for parallel and distributed systems. However, they focus on the usage of analyses for sequential programs to improve the coordination in parallel systems. Abstract interpretation based approaches to resource analysis [5, 39] are limited to sequential programs.

Finally, there exists research that studies cost models to formally analyze parallel programs. Blelloch and Greiner [10] pioneered the cost measures work and depth that we use in this work. There are more advanced cost models that take into account caches and IO (see, e.g., Blelloch and Harper [11]), However, these works do not provide machine support for deriving static cost bounds.

9 Conclusion

We have introduced the first type-based cost analysis for deriving bounds on the depth of evaluations of parallel function programs. The derived bounds are multivariate resource polynomials that can express a wide range of relations between different parts of the input. As any type system, the analysis is naturally compositional. The new analysis system has been implemented in Resource Aware ML (RAML) [14]. We have performed a thorough and reproducible experimental evaluation with typical examples from functional programming that shows the practicability of the approach.

An extension of amortized resource analysis to handle non-polynomial bounds such as max and log in a compositional way is an orthogonal research question that we plan to address in the future. A promising direction that we are currently studying is the use of numerical logical variables to guide the analysis to derive non-polynomial bounds. The logical variables would be treated like regular

variables in the analysis. However, the user would be responsible for maintaining and proving relations such as $a = \log n$ where a is a logical variable an n is the size of a regular data structure. In this way, we would gain flexibility while maintaining the compositionality of the analysis.

Another orthogonal question is the extension of the analysis to additional language features such as higher-order functions, references, and user-defined data structures. These extensions have already been implemented in a prototype and pose interesting research challenges in there own right. We plan to report on them in a forthcoming article.

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

References

1. Vasconcelos, P.: Space Cost Analysis Using Sized Types. PhD thesis, School of Computer Science, University of St Andrews (2008)
2. Danner, N., Paykin, J., Royer, J.S.: A Static Cost Analysis for a Higher-Order Language. In: 7th Workshop on Prog. Languages Meets Prog. Verification (PLPV'13). (2013) 25–34
3. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). (2003) 185–197
4. Simões, H.R., Vasconcelos, P.B., Florido, M., Jost, S., Hammond, K.: Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In: 17th Int. Conf. on Funct. Prog. (ICFP'12). (2012) 165–176
5. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
6. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07). (2007) 157–172
7. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. (2012)
8. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating Runtime and Size Complexity Analysis of Integer Programs. In: Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14). (2014) 140–155
9. Sinn, M., Zuleger, F., Veith, H.: A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In: Computer Aided Verification - 26th Int. Conf. (CAV'14). (2014) 743–759
10. Blelloch, G.E., Greiner, J.: A Provable Time and Space Efficient Implementation of NESL. In: 1st Int. Conf. on Funct. Prog. (ICFP'96). (1996) 213–225
11. Blelloch, G.E., Harper, R.: Cache and I/O Effcent Functional Algorithms. In: 40th ACM Symp. on Principles Prog. Langs. (POPL'13). (2013) 39–50
12. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press (2012)
13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th ACM Symp. on Principles of Prog. Langs. (POPL'11). (2011)

14. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource Aware ML. In: 24rd Int. Conf. on Computer Aided Verification (CAV'12). (2012)
15. Blelloch, G.E.: Nesl: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, CMU (1995)
16. Aehlig, K., Hofmann, M., Hoffmann, J.: RAML Web Site. `http://raml.tcs.ifi.lmu.de` (2010-2014)
17. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. `http://cs.yale.edu/~hoffmann/papers/parallelcost2014.pdf` (2014) Full Version.
18. Charguéraud, A.: Pretty-Big-Step Semantics. In: 22nd Euro. Symp. on Prog. (ESOP'13). (2013) 41–60
19. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th Euro, Symp. on Prog. (ESOP'10). (2010)
20. Hoffmann, J., Shao, Z.: Type-Based Amortized Resource Analysis with Integers and Arrays. In: 12th International Symposium on Functional and Logic Programming (FLOPS'14). (2014)
21. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Prog. Langs. and Systems - 8th Asian Symposium (APLAS'10). (2010)
22. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th ACM Symp. on Principles of Prog. Langs. (POPL'10). (2010) 223–236
23. Campbell, B.: Amortised Memory Analysis using the Depth of Data Structures. In: 18th Euro. Symp. on Prog. (ESOP'09). (2009) 190–204
24. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: 15th Euro. Symp. on Prog. (ESOP'06). (2006) 22–37
25. Hofmann, M., Rodriguez, D.: Automatic Type Inference for Amortised Heap-Space Analysis. In: 22nd Euro. Symp. on Prog. (ESOP'13). (2013) 593–613
26. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010) 85–103
27. Vasconcelos, P.B., Hammond, K.: Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In: Int. Workshop on Impl. of Funct. Langs. (IFL'03), Springer-Verlag LNCS (2003) 86–101
28. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: 23th ACM Symp. on Principles of Prog. Langs. (POPL'96). (1996) 410–423
29. Lago, U.D., Gaboardi, M.: Linear Dependent Types and Relative Completeness. In: 26th IEEE Symp. on Logic in Computer Science (LICS'11). (2011) 133–142
30. Lago, U.D., Petit, B.: The Geometry of Types. In: 40th ACM Symp. on Principles Prog. Langs. (POPL'13). (2013) 167–178
31. Crary, K., Weirich, S.: Resource Bound Certification. In: 27th ACM Symp. on Principles of Prog. Langs. (POPL'00). (2000) 184–198
32. Danielsson, N.A.: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In: 35th ACM Symp. on Principles Prog. Langs. (POPL'08). (2008) 133–144
33. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM **18**(9) (1975) 528–539
34. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: 19th Int. Static Analysis Symposium (SAS'12). (2012) 405–421
35. Grobauer, B.: Cost Recurrences for DML Programs. In: 6th Int. Conf. on Funct. Prog. (ICFP'01). (2001) 253–264

36. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Prog. Langs. and Systems - 9th Asian Symposium (APLAS'11). (2011) 238–254
37. Zimmermann, W.: Automatic Worst Case Complexity Analysis of Parallel Programs. Technical Report TR-90-066, University of California, Berkeley (1990)
38. Trinder, P.W., Cole, M.I., Hammond, K., Loidl, H.W., Michaelson, G.: Resource Analyses for Parallel and Distributed Coordination. Concurrency and Computation: Practice and Experience **25**(3) (2013) 309–348
39. Zuleger, F., Sinn, M., Gulwani, S., Veith, H.: Bound Analysis of Imperative Programs with the Size-change Abstraction. In: 18th Int. Static Analysis Symposium (SAS'11). (2011)