

Amortized Resource Analysis with Polynomial Potential

A Static Inference of Polynomial Bounds for Functional Programs (Extended Version)

Jan Hoffmann* and Martin Hofmann

Ludwig-Maximilians-Universität München
{jan.hoffmann,martin.hofmann}@ifi.lmu.de

Abstract. In 2003, Hofmann and Jost introduced a type system that uses a potential-based amortized analysis to infer bounds on the resource consumption of (first-order) functional programs. This analysis has been successfully applied to many standard algorithms but is limited to bounds that are linear in the size of the input.

Here we extend this system to polynomial resource bounds. An automatic amortized analysis is used to infer these bounds for functional programs without further annotations if a maximal degree for the bounding polynomials is given. The analysis is generic in the resource and can obtain good bounds on heap-space, stack-space and time usage.

Key words: Functional Programming, Static Analysis, Resource Consumption, Amortized Analysis

1 Introduction

In this paper we study the problem of statically determining an upper bound on the resource usage of a given first-order functional program as a function of the size of its input.

As in an earlier work by Hofmann and Jost [15] we rely on the potential method of amortized analysis to take into account the interaction between the steps of a computation and thus obtain tighter bounds than by a mere addition of the worst case resource bounds of the individual steps. Furthermore, the use of potentials relieves one of the burden of having to manipulate symbolic expressions during the analysis by a priori fixing their format.

The main limitation of the system of Hofmann and Jost [15] is its restriction to linear resource bounds. While this restriction is often acceptable when accounting heap space, it is rather limiting when accounting time and other resources. This raises the question whether it is possible to effectively utilize the potential method to compute super-linear resource bounds. We address the problem in this work by using a potential-based amortized analysis to infer *polynomial resource bounds*.

* Supported by the DFG Graduiertenkolleg 1480 (PUMA).

The analysis system we present applies to functional first-order programs with integers, lists and recursion. We also show how the analysis can be extended to programs with tree-like data structures.

Our analysis of the programs is *fully automatic* and does not require type annotations. It is furthermore *generic in the resource* and provides good bounds on heap space, stack space, clock cycles (time) or other resources that might be of interest to a user.

The linear system [15, 19] has been successfully applied in the domain of embedded systems [13]. We envisage that the present extension will also have applications there, in particular in situations where only a few functions exhibit super-linear resource consumption. For this, it is important that the system described here properly extends the linear one so that no expressive power is lost when moving to polynomials.

We give examples of typical programs with a polynomial resource behavior to which our extended system successfully applies. The examples have been implemented in a prototype of the system that is available online¹. It can be directly used in a web browser to analyze and to evaluate user generated programs. We experimented with a variety of example programs such as

- quicksort, mergesort, insertion sort
- multiplication and division for bit-vectors of arbitrary length
- longest common subsequence via dynamic programming
- breadth-first traversal of a tree using a functional queue
- sieve of Eratosthenes

A comparison of the computed bounds with the actual resource costs showed that many bounds exactly match the measured worst-case time and heap-space behaviors of the functions (this is for instance the case for quicksort, insertion sort, pairs and triples). Plots of our experiments are available online and in §8.

The main conceptual contribution of this paper lies in the transfer of the analysis method of Hofmann and Jost from linear to polynomial bounds. They used an *automatic amortized analysis* to infer first-order types that are annotated with information on the resource consumption. The analysis works basically like a standard type inference instrumented with linear constraints for the type annotations that can then be solved by linear programming. For this method to work it is essential that the occurring constraints are linear. Since one would expect an analysis for non-linear bounds to result in non-linear constraints it has been often assumed that amortized analysis is limited to linear bounds. That is maybe why the problem of an extension of amortized analysis to super-linear bounds has remained open for several years. The *amortized analysis with polynomial potential* we present is an elegant and powerful extension of the amortized analysis to polynomial bounds that naturally results in linear constraints.

Another substantial contribution is the use of amortized analysis to *infer sized types* in functional programs. Sized types provide a general framework to represent the size of the data in its type [17]. There has been work on the inference

¹ <http://raml.tcs.ifi.lmu.de>

of linear bounds for sized types [16, 5] and on the inference of size information for so-called shapely functions, i.e., functions such that the size of the function value is determined exactly by a polynomial in the sizes of the arguments. Our amortized analysis with polynomial potential provides an inference system for sized types which is not limited to linear bounds nor to shapely functions. We can rather infer sized types for the same wide range of functions whose resource consumption can be analyzed by our system.

The paper is organized as follows. In §2 we introduce the concept of amortized analysis and informally describe the novel technique that we introduce here. We then, in §3, define the functional programming language RAML (Resource Aware ML) that is used to describe our system and give, in §4, the operational big-step semantics that define the resource consumption of RAML programs. In the sections 5 and 6 we define the type system for the resource aware types that are used in the analysis. §7 shows the analysis of example functions. §8 contains a compilation of the experiments that we performed with the implementation. In §9 we describe how our system can be used to infer sized types and §10 shows how this size information can be used in turn to improve the amortized analysis. The inference algorithm is presented in §11 and in §12 we discuss how the analysis can be extended to trees. §13 gives an overview of the related work.

In the conference version of this article we omitted §9, §12, and §8. Furthermore, it contains only short versions of §7, §10 and §11.

2 Amortized Analysis: Examples and Intuition

Amortized analysis was initially introduced by Sleator and Tarjan [21] to analyze the efficiency of data structures. For a given data structure one is often interested in the costs of a sequence of operations whose costs vary depending on the state of the data structure. A method to analyze the cost of such a sequence is to introduce a non-negative potential of the data structure that can be used to pay (costly) operations. More precisely one defines the *amortized cost* of an operation as the sum of its actual cost and the (possibly negative) net gain of potential incurred by its invocation. The sum of the amortized costs taken over a sequence of operations plus the potential of the initial data structure then furnishes an upper bound on the actual cost of that sequence.

In 2003, Hofmann and Jost [15] applied amortized analysis to type systems in order to derive linear bounds on the heap-space usage of functional programs. The idea is to assign a linear potential to all data structures of variable length. This potential can then be used to “pay” for the resource consumption of functions that are applied to that data. Consider for example the function $attach:(int, L(int)) \rightarrow L(int, int)$ that takes an integer and a list of integers and returns a list of pairs of integers such that the first argument is attached to every element of the list. The expression $attach(1, [1, 2, 3, 4])$ thus evaluates to $[(1, 1), (1, 2), (1, 3), (1, 4)]$. The function $attach$ can be implemented as follows.

```
attach(x, l) = match l with | nil      → nil
                       | (y :: ys) → (x, y) :: (attach (x, ys))
```

To analyze the heap-space usage of *attach* we suppose that we need one memory cell for both creating a new list element, and creating a new pair. The heap-space usage of an execution of $attach(x,l)$ is then $2n$ memory cells if n is the length of l . This fact can be expressed by the resource-annotated type

$$attach: (int, L^{(2)}(int)) \xrightarrow{0/0} L^{(0)}(int, int).$$

The intuitive meaning of this typing is the following: To evaluate $attach(x,l)$ one needs 0 memory cells and 2 memory cells per element in the list. After the execution there are 0 memory cells and 0 cells per element of the returned list left. We say that the list l has the potential $\Phi(l, 2) = 2 \cdot |l|$ and that $attach(x,l)$ has the potential 0. Another possible typing of *attach* would be

$$attach: (int, L^{(4)}(int)) \xrightarrow{8/8} L^{(2)}(int, int).$$

This typing could be used for the inner occurrence of *attach* to type an expression like $attach(x, attach(z, ys))$.

Surprisingly, it turned out that such resource-annotated types can be automatically inferred without requiring any type annotations [15]. Essentially, the inference is done by a conventional type checking that produces linear inequalities which can be solved with linear programming. Furthermore, it has been shown [19] that the same potential-based approach can be similarly applied to a wide range of resources such as time and stack space [4] as well as to polymorphic, higher-order programs [18].

Now consider the function $pairs: L(int) \rightarrow L(int, int)$ that computes the two-element sets of a given set (if one views the input list as a set). The expression $pairs([1,2,3,4])$ thus evaluates to the list $[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]$. Below is an implementation of *pairs*.

```
pairs l = match l with | nil      → nil
                  | (x::xs) → append(attach(x,xs),pairs xs)
```

Since the size of $pairs(l)$ is quadratic in the size of l it is impossible to assign *pairs* a type with linear potential analogous to *attach*. In the next sections we show how to extend the linear potential annotation in a way that allows us to type functions with a polynomial resource consumption while still being able to perform automatic type inference. The function *pairs* could then be assigned the typing

$$pairs: L^{(0,4)}(int) \xrightarrow{0/0} L^{(1)}(int, int).$$

This means that a list l in an expression $pairs(l)$ has the potential $\Phi(l, (0, 4)) = 0 \cdot |l| + 4 \cdot \binom{|l|}{2}$ and thus the linear potential $4|l'|$ for every sub-list (suffix) l' of l . The function *append* could get the type

$$append: (L^{(2)}(int, int), L^{(1)}(int, int)) \xrightarrow{0/0} L^{(1)}(int, int)$$

since the function consumes one heap-cell for every element in the first argument. That is why $pairs(l)$ consumes 3 heap-cells per element of every sub-list of l and we can attach the potential 1 to every element of the list $pairs(l)$.

In a nutshell, our approach is as follows. We start from an as yet unknown potential-function of the form $\sum p_i(n_i)$ with polynomials p_i of a given maximal degree k and n_i referring to the sizes of the parameters. We then derive linear constraints on the coefficients of the p_i by type-checking the program. We choose, and this is an important contribution, a representation of polynomials of degree k as sums $\sum_{i=0,\dots,k} a_i \binom{n}{i}$ with $a_i \geq 0$. Compared with the traditional representation $\sum a_i \cdot n^i$, $a_i \geq 0$, this has the following advantages.

1. Some naturally arising resource bounds such as $\sum_{i=1,\dots,n} i$ cannot be expressed as a polynomial with non-negative coefficients in the traditional representation. On the other hand it is true that $\binom{n}{2} = \sum_{i=1,\dots,n} i$.
2. It is the largest class \mathcal{C} of non-negative, monotone polynomials such that $p \in \mathcal{C}$ implies $f(n) = p(n+1) - p(n) \in \mathcal{C}$ (see §5). All three properties are clearly desirable. The latter one, in particular, expresses that the “spill” arising upon shortening a list by one falls itself into \mathcal{C} .
3. The identity $\sum_{i=1,\dots,k} a_i \binom{n+1}{i} = a_1 + \sum_{i=1,\dots,k-1} a_{i+1} \binom{n}{i} + \sum_{i=1,\dots,k} a_i \binom{n}{i}$ gives rise to a local typing rule for *cons match* which very naturally allows the typing of both recursive calls and other calls to subordinate functions.
4. The linear constraints arising from the type inference have a very simple form due to the above equation. In particular each constraint involves at most three variables.

A key notion in the polynomial system is the additive shift \triangleleft of a type annotation which is defined through $\triangleleft(q_1, \dots, q_k) = (q_1 + q_2, \dots, q_{k-1} + q_k, q_k)$ to reflect the identity from item 3. It is for instance present in the typing $tail: L^{\vec{q}}(int) \xrightarrow{0/q_1} L^{\triangleleft(\vec{q})}(int)$ of the function *tail* that removes the first element from a list.

The idea behind the additive shift is that the potential resulting from the contraction $xs: L^{\triangleleft(\vec{q})}(int)$ of a list $(x::xs): L^{\vec{q}}(int)$ (usually in a pattern match) is used for three purposes: i) to pay the constant costs after and before the recursive calls (q_1), ii) to fund calls to auxiliary functions $((q_2, \dots, q_n))$, and iii) to pay for the recursive calls $((q_1, \dots, q_n))$. For instance, this pattern is present in the definition of the function *pairs*: In the pattern match, the type $xs: L^{(4,4)}(int)$ is assigned to the variable *xs*. The potential is then shared between the two occurrences of *xs* in the following expression by using $xs: L^{(4,0)}(int)$ to pay for *append* and *attach* (ii) and using $xs: L^{(0,4)}(int)$ to pay for the recursive call of *pairs* (iii); the constant costs (i) are zero in this example.

In this paper we restrict ourselves to bounds that are sums of univariate polynomials. Mixed bounds such as $m \cdot n$ must be over-approximated by polynomials like $m^2 + n^2$. This results in a particularly efficient inference algorithm since the number of constraints grows only linear in the maximal degree of the polynomials (see §11). We are nevertheless currently investigating an extension to arbitrary multivariate polynomials.

3 RAML – A Functional Programming Language

In this section we define the functional first-order language RAML (Resource Aware ML). RAML is similar to LF (linear functional language) from [15]. It enjoys an ML-style syntax, Booleans, integers, pairs, lists, recursion and pattern matching.

The differences between LF and RAML are irrelevant for the resource aware type analysis. On the one hand, we have added integers to formulate more realistic examples. On the other hand, we have abandoned the sum type since it is not used in the examples that are presented here. Additionally, for the sake of simplicity, we do not have a destructive match operation in RAML. The integration of both features into the system is straightforward and analogous to the method used in LF.

Below is the EBNF grammar for the *expressions of RAML*. We skip the standard definitions of integer constants $n \in \mathbb{Z}$ and variable identifiers $x \in \text{VID}$.

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

For the resource analysis it is unimportant which ground operations are used in the definition of *binop*. In fact one can use here every function that has a constant worst-case resource consumption. In our case we assume that we have integers of a fixed length, say 32 bits, in our system to ensure this property of the integer operations.

In the examples we often write $(x::y)$ instead of $\text{cons}(x,y)$.

We restrict our attention mainly to *list types* in this paper. However, we discuss extensions to other algebraic data types in §12.

The expressions of RAML are in *let normal form*. This means that term formers are applied to variables only whenever possible. This simplifies typing rules and semantics considerably without hampering expressivity in any way.

Below we define the well-typed expressions of RAML by assigning a *simple type*, i.e. a usual ML type without resource annotations, to every well-typed expression. Simple types are zero-order and first-order types as given by the following grammars.

$$A ::= \text{unit} \mid \text{bool} \mid \text{int} \mid L(A) \mid (A, A) \qquad F ::= (A, \dots, A) \rightarrow A$$

Let \mathcal{A}_S be the set of simple zero-order types (A in the grammar) and let \mathcal{F}_S be the set of simple first-order types (F in the grammar).

The typing rules for RAML expressions are given as an affine linear type system with a sharing rule that explicitly tracks multiple occurrences of variables. The type system thus imposes no linearity restrictions but gives finer information on occurrences of variables than a simple type system does.

A *typing context* is a partial, finite function $\Gamma : \text{VID} \rightarrow \mathcal{A}_S$ from variable identifiers to zero-order types. As usual Γ_1, Γ_2 denotes the union of the contexts Γ_1 and Γ_2 provided that $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$. We thus have the implicit side condition $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$ whenever Γ_1, Γ_2 occurs in a typing rule.

Let FID be a set of function identifiers. A *signature* $\Sigma : \text{FID} \rightarrow \mathcal{F}_S$ is a finite, partial mapping of function identifiers to first-order types.

The typing judgment $\Gamma \vdash_{\Sigma} e : A$ is defined by the following rules and states that the expression e has type A under the signature Σ in the context Γ .

$$\begin{array}{c}
\frac{}{\emptyset \vdash_{\Sigma} () : \text{unit}} \text{ (S:CONST-U)} \qquad \frac{b \in \{\text{True}, \text{False}\}}{\emptyset \vdash_{\Sigma} b : \text{bool}} \text{ (S:CONST-B)} \\
\\
\frac{n \in \mathbb{Z}}{\emptyset \vdash_{\Sigma} n : \text{int}} \text{ (S:CONST-I)} \qquad \frac{}{x : A \vdash_{\Sigma} x : A} \text{ (S:VAR)} \\
\\
\frac{op \in \{+, -, *, \text{mod}, \text{div}\}}{x_1 : \text{int}, x_2 : \text{int} \vdash_{\Sigma} x_1 \text{ op } x_2 : \text{int}} \text{ (S:BINOP-I)} \\
\\
\frac{op \in \{\text{or}, \text{and}\}}{x_1 : \text{bool}, x_2 : \text{bool} \vdash_{\Sigma} x_1 \text{ op } x_2 : \text{bool}} \text{ (S:BINOP-B)} \\
\\
\frac{\Sigma(f) = (A_1, \dots, A_k) \rightarrow A}{x_1 : A_1, \dots, x_k : A_k \vdash_{\Sigma} f(x_1, \dots, x_k) : A} \text{ (S:FUNAPP)} \\
\\
\frac{\Gamma_1 \vdash_{\Sigma} e_1 : A \quad \Gamma_2, x : A \vdash_{\Sigma} e_2 : B}{\Gamma_1, \Gamma_2 \vdash_{\Sigma} \text{let } x = e_1 \text{ in } e_2 : B} \text{ (S:LET)} \\
\\
\frac{\Gamma \vdash_{\Sigma} e_t : A \quad \Gamma \vdash_{\Sigma} e_f : A}{\Gamma, x : \text{bool} \vdash_{\Sigma} \text{if } x \text{ then } e_t \text{ else } e_f : A} \text{ (S:COND)} \\
\\
\frac{}{x_1 : A_1, x_2 : A_2 \vdash_{\Sigma} (x_1, x_2) : (A_1, A_2)} \text{ (S:PAIR)} \\
\\
\frac{\Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\Sigma} e : B}{\Gamma, x : (A_1, A_2) \vdash_{\Sigma} \text{match } x \text{ with } (x_1, x_2) \rightarrow e : B} \text{ (S:MATCH-P)} \\
\\
\frac{A \in \mathcal{A}_S}{\emptyset \vdash_{\Sigma} \text{nil} : L(A)} \text{ (S:NIL)} \qquad \frac{}{x_h : A, x_t : L(A) \vdash_{\Sigma} \text{cons}(x_h, x_t) : L(A)} \text{ (S:CONS)} \\
\\
\frac{\Gamma \vdash_{\Sigma} e_1 : B \quad \Gamma, x_h : A, x_t : L(A) \vdash_{\Sigma} e_2 : B}{\Gamma, x : L(A) \vdash_{\Sigma} \text{match } x \text{ with } \mid \text{nil} \rightarrow e_1 \mid \text{cons}(x_h, x_t) \rightarrow e_2 : B} \text{ (S:MATCH-L)}
\end{array}$$

$$\frac{\Gamma \vdash_{\Sigma} e : B \quad x \in \text{VID} \quad A \in \mathcal{A}_{\Sigma}}{\Gamma, x:A \vdash_{\Sigma} e : B} \text{ (S:AUGMENT)}$$

$$\frac{\Gamma, x:A, y:A \vdash_{\Sigma} e : B}{\Gamma, z:A \vdash_{\Sigma} e[z/x, z/y] : B} \text{ (S:SHARE)}$$

A *RAML program* is a tuple that consists of a signature Σ and a family of expressions with specified variable identifiers $(e_f, \vec{y}^f)_{f \in \text{dom}(\Sigma)}$ such that $y_1^f : A_1, \dots, y_k^f : A_k \vdash_{\Sigma} e_f : A$ if $\Sigma(f) = (A_1, \dots, A_k) \rightarrow A$.

In the example programs we write $f(y_1^f, \dots, y_k^f) = e_f$ to indicate that the expression e_f and the variables y_1^f, \dots, y_k^f are associated with the function f .

4 Operational Semantics for RAML

In this section we define a big-step operational semantics for RAML which is instrumented with resource counters. It is parametric in the particular resource of interest and can be instantiated for different resources including time, heap space and stack size.

The resource counter is only used to state and to prove the soundness of the upper bounds that follow from the resource aware type system and can be omitted in an implementation of the system. One can think of the counter as an abstraction of the actual resource consumption of the assembler code. In the more abstract view of this paper it simply defines the resource behavior of RAML.

Preliminaries: Let Loc be an infinite set of *locations* modeling memory addresses on a heap. The set of RAML *values* Val is given by

$$v ::= l \mid b \mid n \mid \text{NULL} \mid (v, v)$$

Thus a value $v \in \text{Val}$ is either a location $l \in \text{Loc}$, a Boolean constant b , an integer n , a null value NULL or a pair of values (v_1, v_2) .

A *heap* is a finite partial function $\mathcal{H} : \text{Loc} \rightarrow \text{Val}$ that maps locations to values. A *stack* is a finite partial mapping $\mathcal{V} : \text{VID} \rightarrow \text{Val}$ from variables to values.

The rules below define an evaluation judgment of the form

$$\mathcal{V}, \mathcal{H} \stackrel{q}{\vdash}_{q'} e \rightsquigarrow v, \mathcal{H}'$$

expressing the following. If $q \in \mathbb{Q}^+$ is the value of the resource counter and if the stack \mathcal{V} and the initial heap \mathcal{H} are given then the expression e evaluates to the value v and the new heap \mathcal{H}' . Furthermore the resource counter is never negative during the evaluation and $q' \in \mathbb{Q}^+$ is the value of the resource counter after the evaluation. The actual resource consumption is then $\delta = q - q'$. Note that δ could be negative if resources become available during the execution of e .

There can exist two different evaluation judgments $\mathcal{V}, \mathcal{H} \stackrel{q}{\vdash}_{q'} e \rightsquigarrow v, \mathcal{H}'$ and $\mathcal{V}, \mathcal{H} \stackrel{p}{\vdash}_{p'} e \rightsquigarrow v, \mathcal{H}'$ for an expression e under the same heap \mathcal{H} and stack \mathcal{V} .

But then the resource consumption δ of e is identical in both cases and thus $\delta = q - q' = p - p'$. Since $q, q', p, p' \in \mathbb{Q}^+$ it follows also that $q, p \geq \delta$. Moreover it is an invariant of the rules that if $\mathcal{V}, \mathcal{H} \vdash_{q'}^q e \rightsquigarrow v, \mathcal{H}'$ then also $\mathcal{V}, \mathcal{H} \vdash_{q'+a}^{q+a} e \rightsquigarrow v, \mathcal{H}'$ for every $a \geq 0$.

The execution steps below are formulated with respect to constants $K \in \mathbb{Q}$ that depend on the resource the user is interested in. For example one could set $K^{\text{pair}} = K^{\text{cons}} = 1$ and $K = 0$ for all other constants K to analyze the number of heap-cells that are used during the execution. The constants might also be negative if resources are restituted during an execution step. This is the case for stack space and also heap space if one were to include destructive pattern matching as in LF [15] which is omitted here for simplicity.

$$\begin{array}{c}
\frac{}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{unit}}} () \rightsquigarrow \text{NULL}, \mathcal{H}} \text{(E:CONST-U)} \quad \frac{b \in \{\text{True}, \text{False}\}}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{bool}}} b \rightsquigarrow b, \mathcal{H}} \text{(E:CONST-B)} \\
\\
\frac{n \in \mathbb{Z}}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{int}}} n \rightsquigarrow n, \mathcal{H}} \text{(E:CONST-I)} \quad \frac{x \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{var}}} x \rightsquigarrow \mathcal{V}(x), \mathcal{H}} \text{(E:VAR)} \\
\\
\frac{\begin{array}{c} op \in \{+, -, *, \text{mod}, \text{div}, \text{and}, \text{or}\} \\ x_1, x_2 \in \text{dom}(\mathcal{V}) \quad v = op(\mathcal{V}(x_1), \mathcal{V}(x_2)) \end{array}}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{op}}} x_1 \text{ op } x_2 \rightsquigarrow v, \mathcal{H}} \text{(E:BINOP)} \\
\\
\frac{\begin{array}{c} \Sigma(f) = (A_1, \dots, A_k) \rightarrow A \quad \forall 1 \leq i \leq n : \mathcal{V}(x_i) = v_i \\ [y_1^f \mapsto v_1, \dots, y_k^f \mapsto v_k], \mathcal{H} \vdash_{q'+K_2^{\text{app}}}^{q-K_1^{\text{app}}} e_f \rightsquigarrow v, \mathcal{H}' \end{array}}{\mathcal{V}, \mathcal{H} \vdash_{q'}^q f(x_1, \dots, x_k) \rightsquigarrow v, \mathcal{H}'} \text{(E:FUNAPP)} \\
\\
\frac{\mathcal{V}, \mathcal{H} \vdash_{q_2}^{q_1 - K_1^{\text{let}}} e_1 \rightsquigarrow v_1, \mathcal{H}_1 \quad \mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash_{q_3 + K_3^{\text{let}}}^{q_2 - K_2^{\text{let}}} e_2 \rightsquigarrow v_2, \mathcal{H}_2}{\mathcal{V}, \mathcal{H} \vdash_{q_3}^{q_1} \text{let } x = e_1 \text{ in } e_2 \rightsquigarrow v_2, \mathcal{H}_2} \text{(E:LET)} \\
\\
\frac{\mathcal{V}(x) = \text{True} \quad \mathcal{V}, \mathcal{H} \vdash_{q'+K_2^{\text{condT}}}^{q-K_1^{\text{condT}}} e_t \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash_{q'}^q \text{if } x \text{ then } e_t \text{ else } e_f \rightsquigarrow v, \mathcal{H}'} \text{(E:COND-T)} \\
\\
\frac{\mathcal{V}(x) = \text{False} \quad \mathcal{V}, \mathcal{H} \vdash_{q'+K_2^{\text{condF}}}^{q-K_1^{\text{condF}}} e_f \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash_{q'}^q \text{if } x \text{ then } e_t \text{ else } e_f \rightsquigarrow v, \mathcal{H}'} \text{(E:COND-F)} \\
\\
\frac{x_1, x_2 \in \text{dom}(\mathcal{V}) \quad v = (\mathcal{V}(x_1), \mathcal{V}(x_2)) \quad l \notin \text{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\text{pair}}} (x_1, x_2) \rightsquigarrow l, \mathcal{H}[l \mapsto v]} \text{(E:PAIR)} \\
\\
\frac{\mathcal{V}(x) = l \quad \mathcal{H}(l) = (v_1, v_2)}{\mathcal{H}, \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash_{q'+K_2^{\text{matchP}}}^{q-K_1^{\text{matchP}}} e \rightsquigarrow v, \mathcal{H}'} \text{(E:MATCH-P)} \\
\\
\frac{}{\mathcal{V}, \mathcal{H} \vdash_{q'}^q \text{match } x \text{ with } (x_1, x_2) \rightarrow e \rightsquigarrow v, \mathcal{H}'} \text{(E:MATCH-P)}
\end{array}$$

$$\begin{array}{c}
\frac{}{\mathcal{V}, \mathcal{H} \vdash \frac{q+K^{\text{nil}}}{q} \text{ nil} \rightsquigarrow \text{NULL}, \mathcal{H}} \text{ (E:NIL)} \\
\\
\frac{x_h, x_t \in \text{dom}(\mathcal{V}) \quad v = (\mathcal{V}(x_1), \mathcal{V}(x_2)) \quad l \notin \text{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash \frac{q+K^{\text{cons}}}{q} \text{ cons}(x_h, x_t) \rightsquigarrow l, \mathcal{H}[l \mapsto v]} \text{ (E:CONS)} \\
\\
\frac{\mathcal{V}(x) = \text{NULL} \quad \mathcal{H}, \mathcal{V} \vdash \frac{q-K_1^{\text{matchN}}}{q'+K_2^{\text{matchN}}} e_1 \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash \frac{q}{q'} \text{ match } x \text{ with } \mid \text{ nil} \rightarrow e_1 \mid \text{ cons}(x_h, x_t) \rightarrow e_2 \rightsquigarrow v, \mathcal{H}'} \text{ (E:MATCH-N)} \\
\\
\frac{\mathcal{V}(x) = l \quad \mathcal{H}(l) = (v_h, v_t) \quad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H} \vdash \frac{q-K_1^{\text{matchC}}}{q'+K_2^{\text{matchC}}} e_2 \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash \frac{q}{q'} \text{ match } x \text{ with } \mid \text{ nil} \rightarrow e_1 \mid \text{ cons}(x_h, x_t) \rightarrow e_2 \rightsquigarrow v, \mathcal{H}'} \text{ (E:MATCH-C)}
\end{array}$$

Our analysis works for every resource whose consumption can be measured by a counter in the big-step semantics. This allows the analysis a wide range of resources including clock cycles, heap space or even very particular resource consumptions that might be of interest in special embedded systems.

Actual constants for stack-space, heap-space and clock-cycle consumption have been determined for the abstract machine of the language Hume [14] on the Renesas M32C/85U architecture. A list can be found in the literature [19].

5 Resource Annotations for Polynomial Bounds

Resource-annotated types are simple types where lists are annotated with non-negative vectors $\vec{p} \in \mathbb{Q}^n$. These vectors associate a potential with the list that can be used to pay for resource consumptions during an execution.

Recall the example functions *attach* and *pairs* that have been introduced in §2. We assigned the annotated type $\text{attach}: (int, L^{(2)}(int)) \xrightarrow{0/0} L^{(0)}(int, int)$ to the function to indicate that the evaluation of $\text{attach}(x, l)$ consumes $2 \cdot |l|$ resource units.

The function *pairs* calls the function *attach* for every sub-list (suffix) of the input which leads to a quadratic resource consumption. This corresponds to a general pattern in the sense that many typical quadratic functions consume a linear amount of resources for every sub-lists (suffix) of an input just like a typical linear function that consumes a constant amount of resources per element in its input list. We reflect this resource behavior by assigning a type like $\text{pairs}: L^{(0,5)}(int) \xrightarrow{0/0} L^{(2)}(int, int)$. Informally, this type says: To evaluate $\text{pairs}(l)$ one needs 0 resource units per element of l and 5 resource units per element of each sub-list of l . The result of the computation is a list of pairs of integers that has a potential of 2 resource units per element.

In general we define *resource-annotated zero-order types* A as follows.

$$A ::= \text{unit} \mid \text{bool} \mid \text{int} \mid L^{\vec{p}}(A) \mid (A, A)$$

Here \vec{p} is a *resource annotation* for a list type which is defined as a k -tuple $\vec{p} = (p_1, \dots, p_k) \in \mathbb{Q}^k$ with $p_i \geq 0$ and $k > 0$. Let \mathcal{A} be the set of resource-annotated zero-order types.

For two resource annotations $\vec{p} = (p_1, \dots, p_k)$ and $\vec{q} = (q_1, \dots, q_l)$ we write $\vec{p} \leq \vec{q}$ if $k \leq l$ and $p_i \leq q_i$ for all $1 \leq i \leq k$. If $l \geq k$ then we define $\vec{p} + \vec{q} = (p_1 + q_1, \dots, p_k + q_k, q_{k+1}, \dots, q_l)$.

Let $\vec{p} = (p_1, \dots, p_k)$ be an annotation for a list type. The *additive shift* of \vec{p} is $\triangleleft(\vec{p}) = (p_1 + p_2, p_2 + p_3, \dots, p_{k-1} + p_k, p_k)$.

Let \mathcal{H} be a heap and A be a resource-annotated type and let v be a value matching type A in \mathcal{H} . The *potential* $\Phi_{\mathcal{H}}(v:A)$ is then defined as follows.

- $\Phi_{\mathcal{H}}(\text{NULL}:A) = 0$
- If $A \in \{\text{unit}, \text{int}, \text{bool}\}$ then $\Phi_{\mathcal{H}}(v:A) = 0$.
- If $A = (A_1, A_2)$ and $v = (v_1, v_2)$ is a pair then $\Phi_{\mathcal{H}}(v:A) = \Phi_{\mathcal{H}}(v_1:A_1) + \Phi_{\mathcal{H}}(v_2:A_2)$.
- If $A = L^{(p_1, \dots, p_k)}(A')$ is a list type and $v = l$ is a location with $\mathcal{H}(l) = (v', l')$ then

$$\Phi_{\mathcal{H}}(l:A) = p_1 + \Phi_{\mathcal{H}}(v':A') + \Phi_{\mathcal{H}}(l':L^{\triangleleft(p_1, \dots, p_k)}(A')).$$

In the following sections we will sometimes explain an idea by talking about the potential $\Phi(x:A)$ of a variable x with respect to an annotated type A . In such a case we mean in fact the potential $\Phi_{\mathcal{H}}(\mathcal{V}(x):A)$ with respect to a stack \mathcal{V} and a heap \mathcal{H} that we do not want to specify precisely.

If l_1 is a location that points to a list then we write $\mathcal{H}(l_1) = [v_1, \dots, v_n]$ if $\mathcal{H}(l_i) = (v_i, l_{i+1})$ for $i = 1, \dots, n$ and $l_{n+1} = \text{NULL}$. If $l_1 = \text{NULL}$ then we write $\mathcal{H}(l_1) = []$.

Let for example \mathcal{H} be a heap such that $\mathcal{H}(v) = [v_1 \dots, v_n]$ is a list of integers. Then

- $\Phi_{\mathcal{H}}(l:L^{(p_1)}(\text{int})) = p_1 \cdot n$
- $\Phi_{\mathcal{H}}(l:L^{(0, p_2)}(\text{int})) = \sum_{i=1}^{n-1} p_2 \cdot i = p_2 \frac{n \cdot (n-1)}{2}$
- $\Phi_{\mathcal{H}}(l:L^{(0, 0, p_3)}(\text{int})) = \sum_{i=1}^{n-1} p_3 \frac{i \cdot (i-1)}{2} = p_3 \frac{n \cdot (n-1) \cdot (n-2)}{6}$

The next lemma shows how to express the potential $\Phi_{\mathcal{H}}(v:A)$ of a value v with respect the heap \mathcal{H} and a matching annotated type A in terms of polynomials in the lengths of the lists that are reachable from v . For a list annotation \vec{p} and an integer n we define

$$\phi(n, \vec{p}) = \sum_{i=1}^k \binom{n}{i} p_i.$$

Lemma 1. *Let \mathcal{H} be a heap such that $\mathcal{H}(l) = [v_1 \dots, v_n]$ is a list of length n and let $\vec{p} = (p_1, \dots, p_k)$ be an annotation for a list type. Then $\Phi_{\mathcal{H}}(l:L^{\vec{p}}(A)) = \phi(n, \vec{p}) + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A)$.*

Proof. We prove the statement by induction on n . If $n = 0$ then $l = \text{NULL}$ and we have $\Phi_{\mathcal{H}}(l:L^{\vec{p}}(A)) = 0 = \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \phi(0, \vec{p})$.

Let $n > 0$. It then follows by induction that

$$\begin{aligned}\Phi_{\mathcal{H}}(l:L^{\vec{p}}(A)) &= p_1 + \Phi_{\mathcal{H}}(v_1:A) + \Phi_{\mathcal{H}}(l':L^{\triangleleft(\vec{p})}(A)) \\ &= p_1 + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \phi(n-1, \triangleleft(\vec{p}))\end{aligned}$$

But since

$$\binom{n-1}{i} + \binom{n-1}{i+1} = \binom{n}{i+1} \quad (1)$$

it is true that

$$\begin{aligned}\phi(n-1, \triangleleft(\vec{p})) &= \sum_{i=1}^k \binom{n-1}{i} p_i + \sum_{i=1}^{k-1} \binom{n-1}{i} p_{i+1} \\ &= (n-1)p_1 + \sum_{i=1}^{k-1} \left(\binom{n-1}{i+1} + \binom{n-1}{i} \right) p_{i+1} \\ &= (n-1)p_1 + \sum_{i=1}^{k-1} \binom{n}{i+1} p_{i+1} \quad (\text{by (1)}) \\ &= \sum_{i=1}^k \binom{n}{i} p_i - p_1 = \phi(n, \vec{p}) - p_1\end{aligned}$$

It is essential for the type system that ϕ is linear in the sense of the following lemma that follows directly from the definition of ϕ .

Lemma 2. *Let $n \in \mathbb{N}$, $\alpha \in \mathbb{Q}$ and let \vec{p}, \vec{q} be resource annotations. Then $\phi(n, \vec{p}) + \phi(n, \vec{q}) = \phi(n, \vec{p} + \vec{q})$ and $\alpha \cdot \phi(n, \vec{p}) = \phi(n, \alpha \cdot \vec{p})$.*

As mentioned before it is a general pattern in functional programs to compute a task on a list recursively for the tail of the list and to use the result of the recursive call to compute the result of the function. In such a recursive function it is natural to assign a uniform potential to each sub-list (depending on its length) that occurs in a recursive call. In other words: one wants to use the potential of the input list to assign a uniform potential to every suffix of the list. With this view, the list potential $\alpha = \phi(n, (p_1, p_2, \dots, p_k))$ can be read as follows: a recursive function on a list l of length n that has the potential α can use the potential $\phi(i, (p_2, \dots, p_k))$ for the suffixes of l of length $1 \leq i < n$ that occurs in the recursion. This intuition is proved by the following lemma.

Lemma 3. *Let $\vec{p} = (p_1, \dots, p_k)$ be a resource annotation, let $n \in \mathbb{N}$ and define $\phi(n, ()) = 0$. Then $\phi(n, (p_1, \dots, p_k)) = n \cdot p_1 + \sum_{i=1}^{n-1} \phi(i, (p_2, \dots, p_k))$.*

Proof. The proof uses the following well-known equation.

$$\sum_{i=1}^{n-1} \binom{i}{k} = \binom{n}{k+1} \quad \text{for each } k \in \mathbb{N} \quad (2)$$

Let now $k \geq 0$. Then

$$\begin{aligned}
\phi(n, (p_1, \dots, p_{k+1})) &= \sum_{j=1}^{k+1} \binom{n}{j} p_j \\
&= n \cdot p_1 + \sum_{j=1}^k \binom{n}{j+1} p_{j+1} \\
&= n \cdot p_1 + \sum_{j=1}^k \left(\sum_{i=1}^{n-1} \binom{i}{j} p_{j+1} \right) && \text{(by (2))} \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} \left(\sum_{j=1}^k \binom{i}{j} p_{j+1} \right) \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} \phi(i, (p_2, \dots, p_{k+1})) && \text{(by definition)}
\end{aligned}$$

Note that the binomial coefficients are a basis of the vector space of the polynomials. Here, however, we are only interested in non-negative linear combinations of binomial coefficients. These admit a natural characterization in terms of growth: for $f : \mathbb{N} \rightarrow \mathbb{N}$ define $(\Delta f)(n) = f(n+1) - f(n)$. Call f *hereditarily non-negative* if $\Delta^i f \geq 0$ for all $i \geq 0$. One can show that a polynomial p is hereditarily non-negative if and only if it can be written as a non-negative linear combination of binomial coefficients. To wit, the coefficient of $\binom{n}{i}$ in the representation of p is $(\Delta^i p)(0)$. The hereditarily non-negative polynomials are scalar multiples of unary *resource polynomials* [9] and thus are closed under sum, product, and composition. Note that they include all non-negative linear combinations of the polynomials $(x^i)_{i \in \mathbb{N}}$.

6 Type system

This section presents typing rules for the resource-annotated zero-order types \mathcal{A} that have been defined in §5 and establishes their semantic soundness. Later in §10 we add another rule.

As in the case of the simple types, a *typing context* is a partial finite function $\Gamma : \text{VID} \rightarrow \mathcal{A}$ from variable identifiers to annotated zero-order types. The potential of a typing context Γ with respect to a heap \mathcal{H} and a stack \mathcal{V} is

$$\Phi_{\mathcal{V}, \mathcal{H}}(\Gamma) = \sum_{x \in \text{dom}(\Gamma)} \Phi_{\mathcal{H}}(\mathcal{V}(x) : \Gamma(x))$$

Sometimes we write just $\Phi(\Gamma)$ leaving stack and heap implicit.

The *resource-annotated first-order types* \mathcal{F} are defined by

$$F ::= (A, \dots, A) \xrightarrow{q/q'} A.$$

Here q, q' are rational numbers and A ranges over the resource-annotated zero-order types. Let \mathcal{F} denote the set of resource-annotated first-order types.

A *resource-annotated signature* $\Sigma : \text{FID} \rightarrow \mathcal{F}$ is a finite, partial mapping of function identifiers to resource-annotated first-order types. A *resource-annotated typing judgment* has the form $\Sigma; \Gamma \vdash_{\frac{q}{q'}} e : A$ where e is a RAML expression, $q, q' \in \mathbb{Q}^+$ are non-negative rational numbers, Σ is a resource-annotated signature, Γ is a resource-annotated context and A is a resource-annotated zero-order type. The intended meaning of this judgment is that if there are more than $q + \Phi(\Gamma)$ resource units available then this is sufficient to evaluate e and then there are more than $q' + \Phi(v:A)$ resource units left after the evaluation of e to a value v .

Similarly as for simple types, a RAML program with resource-annotated types is a tuple that consists of a resource-annotated signature Σ and a family of expressions with specified variable identifiers $(e_f, \vec{y}^f)_{f \in \text{dom}(\Sigma)}$ such that for each e_f we have $\Sigma; y_1^f : A_1, \dots, y_k^f : A_k \vdash_{\frac{q}{q'}} e_f : A$ if $\Sigma(f) = (A_1, \dots, A_k) \xrightarrow{q/q'} A$.

The following type rules are used to derive a resource-annotated type judgment for RAML expressions. Therein, we write $e[z/x]$ to denote the expression e with all free occurrences of the variable x replaced with the variable z .

$$\begin{array}{c}
\frac{}{\Sigma; \emptyset \vdash_{\frac{K^{\text{unit}}}{0}} () : \text{unit}} \quad (\text{T:CONST-U}) \qquad \frac{n \in \mathbb{Z}}{\Sigma; \emptyset \vdash_{\frac{K^{\text{int}}}{0}} n : \text{int}} \quad (\text{T:CONST-I}) \\
\\
\frac{b \in \{\text{True}, \text{False}\}}{\Sigma; \emptyset \vdash_{\frac{K^{\text{bool}}}{0}} b : \text{bool}} \quad (\text{T:CONST-U}) \qquad \frac{}{\Sigma; x : A \vdash_{\frac{K^{\text{var}}}{0}} x : A} \quad (\text{T:VAR}) \\
\\
\frac{op \in \{+, -, *, \text{mod}, \text{div}\}}{\Sigma; x_1 : \text{int}, x_2 : \text{int} \vdash_{\frac{K^{\text{op}}}{0}} x_1 \text{ op } x_2 : \text{int}} \quad (\text{T:BINOP-I}) \\
\\
\frac{op \in \{\text{or}, \text{and}\}}{\Sigma; x_1 : \text{bool}, x_2 : \text{bool} \vdash_{\frac{K^{\text{op}}}{0}} x_1 \text{ op } x_2 : \text{bool}} \quad (\text{T:BINOP-B}) \\
\\
\frac{\Sigma(f) = (A_1, \dots, A_k) \xrightarrow{q/q'} A \quad \Gamma = x_1 : A_1, \dots, x_k : A_k}{\Sigma; \Gamma \vdash_{\frac{q + K_1^{\text{app}}}{q' - K_2^{\text{app}}}} f(x_1, \dots, x_k) : A} \quad (\text{T:FUNAPP}) \\
\\
\frac{\Sigma; \Gamma_1 \vdash_{\frac{q - K_1^{\text{let}}}{p}} e_1 : A \quad \Sigma; \Gamma_2, x : A \vdash_{\frac{p - K_2^{\text{let}}}{q' + K_3^{\text{let}}}} e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \vdash_{\frac{q}{q'}} \text{let } x = e_1 \text{ in } e_2 : B} \quad (\text{T:LET}) \\
\\
\frac{\Sigma; \Gamma \vdash_{\frac{q - K_1^{\text{conT}}}{q' + K_2^{\text{conT}}}} e_t : A \quad \Sigma; \Gamma \vdash_{\frac{q - K_1^{\text{conF}}}{q' + K_2^{\text{conF}}}} e_f : A}{\Sigma; \Gamma, x : \text{bool} \vdash_{\frac{q}{q'}} \text{if } x \text{ then } e_t \text{ else } e_f : A} \quad (\text{T:COND}) \\
\\
\frac{}{\Sigma; x_1 : A_1, x_2 : A_2 \vdash_{\frac{K^{\text{pair}}}{0}} (x_1, x_2) : (A_1, A_2)} \quad (\text{T:PAIR}) \\
\\
\frac{A = (A_1, A_2) \quad \Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\frac{q - K_1^{\text{matchP}}}{q' + K_2^{\text{matchP}}}} e : B}{\Sigma; \Gamma, x : A \vdash_{\frac{q}{q'}} \text{match } x \text{ with } (x_1, x_2) \rightarrow e : B} \quad (\text{T:MATCH-P})
\end{array}$$

$$\begin{array}{c}
\frac{A \in \mathcal{A}}{\Sigma; \emptyset \vdash_0^{K^{\text{nil}}} \text{nil}:L(A)} \text{ (T:NIL)} \\
\\
\frac{\vec{p} = (p_1 \dots p_k)}{\Sigma; x_h:A, x_t:L^{\triangleleft(\vec{p})}(A) \vdash_0^{p_1+K^{\text{cons}}} \text{cons}(x_h, x_t):L^{\vec{p}}(A)} \text{ (T:CONS)} \\
\\
\frac{\vec{p} = (p_1, \dots, p_k) \quad \Sigma; \Gamma \vdash_{q'+K_2^{\text{matchN}}}^{q-K_1^{\text{matchN}}} e_1 : B \quad \Sigma; \Gamma, x_h:A, x_t:L^{\triangleleft(\vec{p})}(A) \vdash_{q'+K_2^{\text{matchC}}}^{q+p_1-K_1^{\text{matchC}}} e_2 : B}{\Sigma; \Gamma, x:L^{\vec{p}}(A) \vdash_{q'}^{\text{match } x \text{ with } \mathbf{I} \text{ nil} \rightarrow e_1} e_1 : B \quad \mathbf{I} \text{ cons}(x_h, x_t) \rightarrow e_2} \text{ (T:MATCH-L)} \\
\\
\frac{\Sigma; \Gamma, x:A_1, y:A_2 \vdash_{q'}^q e : B \quad \forall (A \mid A_1, A_2)}{\Sigma; \Gamma, z:A \vdash_{q'}^q e[z/x, z/y] : B} \text{ (T:SHARE)} \\
\\
\frac{\Sigma; \Gamma, x:A \vdash_{q'}^q e : B \quad A' <: A}{\Sigma; \Gamma, x:A' \vdash_{q'}^q e : B} \text{ (T:SUPERTYPE)} \\
\\
\frac{\Sigma; \Gamma \vdash_{q'}^q e : B \quad B <: B'}{\Sigma; \Gamma \vdash_{q'}^q e : B'} \text{ (T:SUBTYPE)} \\
\\
\frac{\Sigma; \Gamma \vdash_{p'}^p e : B \quad q \geq p \quad q-p \geq q'-p'}{\Sigma; \Gamma \vdash_{q'}^q e : B} \text{ (T:RELAX)} \\
\\
\frac{\Sigma; \Gamma \vdash_{q'}^q e : B \quad x \in \text{VID} \quad A \in \mathcal{A}}{\Sigma; \Gamma, x:A \vdash_{q'}^q e : B} \text{ (T:AUGMENT)}
\end{array}$$

The definitions of the relations $\forall (\cdot \mid \cdot, \cdot)$ and $<:$ are given below.

We describe the idea behind the type rules exemplary for T:CONS and T:MATCH. The rule T:CONS formalizes the fact that one has to pay for the resource consumption of the evaluation of $\text{cons}(x_h, x_t)$, i.e., basically the allocation of a new heap-cell that points to x_h and x_t . This is represented by the constant K^{cons} that depends on the resource that is studied. In addition one has to pay for the potential that is assigned to the new list of type $L^{\vec{p}}(A)$. We do so by requiring x_t to have the type $L^{\triangleleft(\vec{p})}(A)$ and to have p_1 resource units available. It corresponds exactly to the recursive definition of the potential function Φ and ensures that potential is neither gained nor lost.

Complementarily, the rule T:MATCH-L defines how to use the potential of a list to pay for resource consumptions. First, it matches the corresponding rules from the operational semantics E:MATCH-* in terms of resource consumption. It incorporates the fact that either e_1 or e_2 is evaluated. More interestingly, the ‘‘cons case’’ is inverse to the rule T:CONS and allows one to use the potential associated with a list. For one thing, p_1 resource units become available directly, for another the tail of the list is annotated with $\triangleleft(\vec{p})$ rather than \vec{p} , permitting e.g. a recursive call (requiring annotation \vec{p}) and an additional use of the tail with annotation (p_2, p_3, \dots) .

It is important that all the numerical constraints that result from rules T:CONS, T:MATCH-L and the other rules are *linear*. This is the reason why it is easy to verify the constraints and why one can use linear programming to infer type annotations that match the constraints.

The Subtyping Relation Intuitively it is true that a zero-order type A is a subtype of a zero-order type B if and only if A and B have the same set of values, and for every value v the potential of $v:A$ is greater or equal than the potential of $v:B$. More formal, we define $<$: to be the smallest relation such that

$$\begin{aligned} C <: C & \text{ if } C \in \{\text{unit}, \text{bool}, \text{int}\} \\ (A_1, A_2) <: (B_1, B_2) & \text{ if } A_1 <: B_1 \text{ and } A_2 <: B_2 \\ L^{\vec{p}}(A) <: L^{\vec{q}}(B) & \text{ if } A <: B \text{ and } \vec{p} \geq \vec{q} \end{aligned}$$

The Sharing Relation The sharing relation $\Psi(\cdot \mid \cdot, \cdot)$ defines how the potential of a zero-order variable can be shared by multiple occurrences of that variable. We will have $\Psi(A \mid A_1, A_2)$ if and only if A , A_1 and A_2 are structural identical, i.e. have the same set of values, and for every value v the potential $\Phi(v:A)$ of $v:A$ is identical to the sum $\Phi(v:A_1) + \Phi(v:A_2)$ of the potentials of $v:A_1$ and $v:A_2$. So $\Psi(\cdot \mid \cdot, \cdot)$ is the smallest relation such that

$$\begin{aligned} \Psi(C \mid C, C) & \text{ if } C \in \{\text{unit}, \text{bool}, \text{int}\} \\ \Psi(L^{\vec{p}}(A) \mid L^{\vec{q}}(A_1), L^{\vec{r}}(A_2)) & \text{ if } \Psi(A \mid A_1, A_2) \text{ and } \vec{p} = \vec{q} + \vec{r} \\ \Psi((A, B) \mid (A_1, B_1), (A_2, B_2)) & \text{ if } \Psi(A \mid A_1, A_2) \text{ and } \Psi(B \mid B_1, B_2) \end{aligned}$$

Soundness of the Analysis The soundness theorem below states that a resource annotated type statement guarantees that an expression can be evaluated in the stated resource bounds and that at least the stated amount of resources is available after the evaluation.

Such a statement is only meaningful with respect to a well-formed stack and a well-formed heap. A stack \mathcal{V} and a heap \mathcal{H} are *well-formed* with respect to a context Γ if $\mathcal{V}(x)$ is a value matching the type $\Gamma(x)$ for every $x \in \text{dom}(\Gamma)$. We then write $\mathcal{H} \models \mathcal{V} : \Gamma$. It is not hard to show that if $\mathcal{H} \models \mathcal{V} : \Gamma$ and $\mathcal{V}, \mathcal{H} \stackrel{q}{\vdash} e \rightsquigarrow v, \mathcal{H}'$ then also $\mathcal{H}' \models \mathcal{V} : \Gamma$.

Theorem 1 (Soundness). *Let Σ be the signature of a given RAML program and let e be an expression. Let $\mathcal{H} \models \mathcal{V} : \Gamma$ and let there exist some $u, u' \in \mathbb{Q}^+$ such that $\mathcal{V}, \mathcal{H} \stackrel{u}{\vdash} e \rightsquigarrow v, \mathcal{H}'$. If*

$$\Sigma; \Gamma \stackrel{p}{\vdash} e : A \quad \text{and} \quad q \geq \Phi_{\mathcal{V}, \mathcal{H}}(\Gamma) + p + r \text{ for a } r \in \mathbb{Q}^+$$

then there is a $q' \geq \Phi_{\mathcal{H}'}(v:A) + p' + r$ such that $\mathcal{V}, \mathcal{H} \stackrel{q'}{\vdash} e \rightsquigarrow v, \mathcal{H}'$.

Theorem 1 is proved in the same way as the corresponding theorem in the system of [15]. The key ingredients that are used are the lemmas from §5.

7 Examples

For simplicity we only provide examples for heap-space consumption in this section. We assume that one heap-cell is allocated whenever new data is created. Thus we set $K^{\text{pair}}=K^{\text{cons}}=1$ and $K=0$ for all other constants K .

For each function we give its annotated type and the type of the potential-carrying variables that appear in its definition. We distinguish different occurrences of the same variable by adding superscripts. To save space we omit some less interesting types and sometimes waive the let-normal form.

The types contain meta-variables p_1, c, d, q_3 ranging over non-negative rational numbers. Any instantiation of the former yields a correct typing.

7.1 Subsets of Size k

Our canonical example for polynomial heap-space consumption is the following problem: view a given list as a set and compute the subsets of size k for a given k . The size of the output is a polynomial of degree k .

Below we define the subset functions for $k = 2$ and $k = 3$ but one can also see how it works for $k > 3$. The function $\text{attach}(x,l)$ computes a list of pairs so that x is paired with every element in the list l . The function $\text{pairs}(l)$ computes a list of all (unordered) pairs that can be built from the elements of l and similarly the function $\text{triples}(l)$ computes a list of all (unordered) triples.

$$\begin{aligned} \text{attach}(x,l) &= \mathbf{match\ l\ with} \mid \text{nil} \quad \rightarrow \text{nil} \\ &\quad \mid (y::ys) \rightarrow \mathbf{let\ } l' = \text{attach}(x,ys) \mathbf{\ in\ } (x,y)::l' \end{aligned}$$

$$\begin{aligned} \text{attach}: (int, L^{(p+2)}(int)) \xrightarrow{c/c} L^{(p)}(int, int) \\ l:L^{(p+2)}(int) \ ys:L^{(p+2)}(int) \ l':L^{(p)}(int, int) \end{aligned}$$

$$\begin{aligned} \text{append}(l,ys) &= \mathbf{match\ l\ with} \mid \text{nil} \quad \rightarrow ys \\ &\quad \mid (x::xs) \rightarrow \mathbf{let\ } l' = \text{append}(xs,ys) \mathbf{\ in\ } x::l' \end{aligned}$$

$$\begin{aligned} \text{append}: (L^{(p+1)}(A), L^{(p)}(A)) \xrightarrow{c/c} L^{(p)}(A) \\ l:L^{(p+1)}(A) \ ys:L^{(p)}(A) \ xs:L^{(p+1)}(A) \quad l':L^{(p)}(A) \end{aligned}$$

$$\begin{aligned} \text{pairs}(l) &= \mathbf{match\ l\ with} \mid \text{nil} \quad \rightarrow \text{nil} \\ &\quad \mid (x::xs) \rightarrow \mathbf{let\ nps} = \text{attach}(x,xs^1) \mathbf{\ in} \\ &\quad \quad \quad \mathbf{let\ rps} = \text{pairs}(xs^2) \mathbf{\ in\ } \text{append}(nps,rps) \end{aligned}$$

$$\begin{aligned} \text{pairs}: L^{(0,p_2+3)}(int) \xrightarrow{c/c} L^{(p_2)}(int, int) \\ l :L^{(0,p_2+3)}(int) \quad xs^1:L^{(p_2+3)}(int) \quad rps:L^{(p_2)}(int, int) \\ xs:L^{(p_2+3,p_2+3)}(int) \quad xs^2:L^{(0,p_2+3)}(int) \quad nps:L^{(p_2+1)}(int, int) \end{aligned}$$

$$\begin{aligned} \text{triples}(l) &= \mathbf{match\ l\ with} \mid \text{nil} \quad \rightarrow \text{nil} \\ &\quad \mid (x::xs) \rightarrow \mathbf{let\ tps} = \text{pairs}(xs^1) \mathbf{\ in} \\ &\quad \quad \quad \mathbf{let\ nts} = \text{attach}(x,tps) \mathbf{\ in} \\ &\quad \quad \quad \mathbf{let\ rts} = \text{triples}(xs^2) \mathbf{\ in\ } \text{append}(nts,rts) \end{aligned}$$

$$\begin{aligned} \text{triples}: L^{(0,0,p_3+6)}(int) \xrightarrow{c/c} L^{(p_3)}(int, int, int) \\ xs :L^{(0,p_3+6,p_3+6)}(int) \quad xs^2:L^{(0,0,p_3+6)}(int) \quad nts:L^{(p_3+1)}(int, int, int) \\ xs^1:L^{(0,p_3+6)}(int) \quad rts:L^{(p_3)}(int, int, int) \quad tps:L^{(p_3+3)}(int, int) \end{aligned}$$

If defined as above the function *dyade* is an example of a function that cannot be typed in our system. The reason is that the heap space that is used by *dyade* is quadratic in both arguments but the quadratic potential in *v2* can only be accessed by a pattern matching on *v2*.

A closer look at the definition of *dyade* shows that there are $|v1|$ calls of $mult(x,v2)$ which can only be paid from the potential in *v2*. But the potential of *v2* does not depend on the length of *v1*.

Luckily there is a uniform approach to such a problem. The idea is basically to do a recursive call $dyade(xs,ys)$ that involves the tails of both input lists. Below we present a different version of *dyade* that can be analyzed by our system.

```
merge(col,mat) = match col with | nil → nil
                | (c::cs) → match mat with | nil → nil
                | (m::ms) → (c::m)::(merge(cs,ms))
```

```
merge:  $(L^{p+2}(int), L^q(L^p(int))) \xrightarrow{c/c} L^q(L^p(int))$ 
```

```
dyade(v1,v2) = match v1 with | nil → nil
                | (x::xs) → match v2a with | nil → nil::dyade(xs,nil)
                | (y::ys) → let mat' = dyade(xs1,ys) in
                    let line = mult(x,v2b) in
                    let col = mult(y,xs2) in
                    let mat = merge(col,mat') in line::mat
```

```
dyade:  $(L^{(q+1,p+4)}(int), L^{(p+2,p+2)}(int)) \xrightarrow{c/c} L^{q+1}(L^p(int))$ 
xs :  $L^{(p+q+5,p+4)}(int)$  v2a:  $L^{(0,p+2)}(int)$  line :  $L^{(p)}(int)$  v1 :  $L^{(q+1,p+4)}(int)$ 
xs1:  $L^{(q+1,p+4)}(int)$  v2b:  $L^{(p+2)}(int)$  mat':  $L^q(L^p(int))$  ys :  $L^{(p+2,p+2)}(int)$ 
xs2:  $L^{(p+4)}(int)$  v2 :  $L^{(p+2,p+2)}(int)$  mat :  $L^q(L^p(int))$  col:  $L^{(p+2)}(int)$ 
```

The idea of the above version of *dyade* is as follows. In order to compute $dyade(x::xs,y::ys)$ compute $dyade(xs,ys)$ first. Then use the linear potential of the first list to create a new column and use the linear potential of the second list to create new line.

As mentioned before, this approach works for a wide range of functions. To see which functions can be typed in this way note that *dyade* can be implemented in ML with two nested folds as follows.

```
fun dyade(v1,v2) = foldr (fn (x,m) => (foldr (fn (y,l) => (x*y)::l) [] v2) :: m)
                        [] v1
```

Assume now that a general function of the following form is implemented in ML.

```
fun h(l1,l2,b0,c0) = foldr (fn (x,c) => g(foldr (fn (y,b) => f(x,y,b)) b0 l2, c))
                          c0 l1
```

Assume furthermore that we have $f: (A_1, A_2, B) \xrightarrow{r/0} B$ and $g: (B, C) \xrightarrow{s/0} C$ already implemented in our system. Then we can implement the function *h* as follows.

```
innerfold(x,l,b0) = match l with | nil → b0
                    | (y::ys) → let b = innerfold(x,ys,b0) in f(x,y,b)
```

```

merge(y,xlist , blist ) = match xlist with | nil → nil
  | (x::xs) → match blist with | nil → nil
  | (b::bs) → f(x,y,b)::merge(x,bs,as)

infolds (l1 , l2 , b0) = match l1 with | nil → nil
  | (x::xs) → match l2 with | nil → b0::infolds(xs,nil,b0)
  | (y::ys) → let folds = infolds(xs,ys,b0) in
  let newfold = innerfold(x,l2,b0) in
  let folds' = merge(y,xs,folds) in newfold::folds'

outerfold (l , c0) = match l with | nil → c0
  | (b::bs) → let c = outerfold(bs) in g(b,c)

h(l1 , l2 , b0 , c0) = let l = innerfolds(l1 , l2 , b0) in outerfold (l , c0)

innerfold: (A1 , Lr(A2) , B)  $\xrightarrow{q/q}$  B
merge : (A2 , Lr+2(A1) , Lp(B))  $\xrightarrow{q/q}$  B
infolds : (L(p+1,r+2)(A1) , L(r,r)(A2) , B)  $\xrightarrow{q/q}$  Lp(B)
outerfold: (Ls(B) , C)  $\xrightarrow{q/q}$  C
h : (L(s+1,r+2)(A1) , L(r,r)(A2) , B , C)  $\xrightarrow{q/q}$  C

```

7.3 Longest Common Subsequence

A standard example of dynamic programming that can be found in many textbooks is the computation of the longest common subsequence (LCS) of two given lists (sequences). Given two sequences a_1, \dots, a_n and b_1, \dots, b_m , one successively fills an $n \times m$ matrix (here a list of lists) A such that $A(i, j)$ contains the length of the LCS of a_1, \dots, a_i and b_1, \dots, b_j . It is the case that

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

This algorithm can be analyzed in our system and is exemplary for similar algorithms that use dynamic programming.

```

tail'(l) = match l with | nil → nil | (x::xs) → xs

firstline(m) = match m with | nil → nil | (l::_) → l

lastvals (l) = match l with | nil → (0,0)
  | (a1::l') → match l' with | nil → (a1,0)
  | (a2::_) → (a1,a2)

tail' : Lp(int)  $\xrightarrow{c/c}$  Lp(int) firstline: Lp(Lq(int))  $\xrightarrow{c/c}$  Lq(int)
lastvals: Lp(int)  $\xrightarrow{c/c}$  (int, int)

addcolumn(m,x,c) = match c with | nil → nil
  | (y::ys) → let m' = addcolumn(tail'(m),x,ys) in

```

```

let (above,updiag) = lastvals( firstline (m')) in
let l1 = firstline (m) in
let (left , -) = lastvals(l1) in
let elem = if x = y then updiag+1
                else max(above,left)
in ((elem::l1)::m')

addcolumn:  $(L^p(L^q(int)), int, L^{p+q+2}(int)) \xrightarrow{c/c} L^p(L^q(int))$ 

newline (y, lastline , l) = match l with | nil  $\rightarrow$  nil
    | (x::xs)  $\rightarrow$  let nl = newline(y,tail'( lastline ),xs) in
        let (left , -) = lastvals(nl) in
        let (above,updiag) = lastvals( lastline ) in
        let elem = if x = y then updiag+1
                    else max(above,left)
        in elem::nl

newline:  $(int, L^q(int), L^{q+1}(int)) \xrightarrow{c/c} L^q(int)$ 

addline(m,y,xs) = let nl = newline(y, firstline (m),xs) in nl::m

addline:  $(L^p(L^q(int)), int, L^{q+1}(int)) \xrightarrow{c+p+1/c} L^p(L^q(int))$ 

lcstable(l1, l2) = match l1 with | nil  $\rightarrow$  nil
    | (x::xs1)  $\rightarrow$  match l21 with | nil  $\rightarrow$  nil
        | (y::ys)  $\rightarrow$  let m = lcstable(xs2,ys) in
            let m' = addline(m,y,xs3) in
            addcolumn(m',x,l22)

lcstable:  $(L^{(0,q+1)}(int), L^{(2p+q+3,p+q+2)}(int)) \xrightarrow{c/c} L^p(L^q(int))$ 
ys :  $L^{(2p+q+3,p+q+2)}(int)$  xs3:  $L^{(q+1,0)}(int)$  m':  $L^p(L^q(int))$  l21:  $L^{(p+1,p+q+2)}(int)$ 
xs1:  $L^{(q+1,q+1)}(int)$  xs2:  $L^{(0,q+1)}(int)$  m :  $L^p(L^q(int))$  l22:  $L^{(p+q+2)}(int)$ 

lcs(l1, l2) = let m = lcstable(l1, l2) in
    match m with | nil  $\rightarrow$  0
        | ((length :: _):: _)  $\rightarrow$  length

lcs:  $(L^{(0,1)}(int), L^{(3,2)}(int)) \xrightarrow{c/c} int$ 

```

8 Experiments

We developed a prototype implementation and implemented a number of well-known, non-trivial algorithms that exhibit a super-linear resource consumption. These examples, as well as the prototype itself, are available online² and can be directly tested and modified in a web-browser. The prototype implementation can analyze the heap-space consumption and the number of evaluation steps. It is adequately documented easy to use. One can use it not only compute resource bounds but also to measure the actual resource consumption of a program. We invite everybody to experiment with it to explore the frontiers of our system.

The algorithms that we implemented include

² <http://raml.tcs.ifi.lmu.de>

- quicksort, mergesort, insertionsort
- multiplication and division for bit-vectors of arbitrary length
- longest common subsequence via dynamic programming
- breadth-first traversal of a tree using a functional queue
- sieve of Eratosthenes

The bounds inferred by our prototype are asymptotically tight in most cases (*quicksort*, *insertionsort*, *pairs*, *triples*, *quadruples*, *eratos*, *startBreadth*). If the bounds are not asymptotically tight then it is the case that the tight bound is not a polynomial (*mergesort*) or that it contains mixed terms like $n \cdot m$ (*mult*, *div*, *dyade*, *lcs*).

The computation of the bounds scales well and does only take a few seconds in most cases.

Table 1 and 2 contain a compilation of our experiments with the prototype. They show a list of functions that we implemented and for each function the computed bound, the asymptotic worst-case behavior, and the run time of the prototype on a MacBook Pro with a 2.16 GHz Intel Core 2 Duo, 2 GB RAM and Mac OS 10.6. The implementation of the functions is available on the website.

Table 1. The computed evaluation-step bounds, the actual worst-case time behavior, and the run time of the analysis in seconds.

	Computed Time Bound	Act. Behavior	Run Time
quicksortD	$3 + 14a + 12a^2$	$O(a^2)$	0.1438 s
insertionsortD	$3 + 6a + 6a^2$	$O(a^2)$	0.0542 s
mergesort	$5 - 46a + 46a^2$	$O(a \log a)$	0.3059 s
pairs	$3 + 7a + 9a^2$	$O(a^2)$	0.0507 s
triples	$3 + 24a - 10a^2 + 6a^3$	$O(a^3)$	0.3043 s
quadruples	$3 + 1.8\bar{3}a + 26.75a^2 - 10.8\bar{3}a^3 + 2.25a^4$	$O(a^4)$	2.0892 s
dyade	$3 - a + 11a^2 + 29b + 5b^2$	$O(a \cdot b)$	0.1673 s
lcs	$13 - 22.5a + 24.5a^2 + 67.5b + 30.5b^2$	$O(a \cdot b)$	2.0146 s
mult	$7 + 143a + 29a^2 - 81b + 87b^2$	$O(a \cdot b)$	0.3173 s
div	$33 + 98a + 34a^2 + 16b$	$O(a \cdot b)$	1.7259 s
eratos	$3 + 4a + 8a^2$	$O(a^2)$	0.0447 s
startBreadth	$17 + 45a + 45a^2$	$O(a^2)$	1.0464 s

To verify the quality of the computed bounds we measured the actual resource consumption of the analyzed functions with a counter and compared it to the bound. In order to measure the worst-case costs for larger inputs we manually identified inputs for which the worst-case behavior emerges (for example a backward ordered list for *quicksort*).

The comparison of the measured resource costs with the computed bounds showed that the bounds match exactly the measured worst-case costs for many functions (e.g. quicksort, insertionsort, pairs and triples). Figures 1, 2, 3 and 4 exemplarily show the plots of our experiments for the functions *quicksort* and

Table 2. The computed heap-space bounds, the actual worst-case heap-space behavior, and the run time of the analysis in seconds.

	Computed Heap Bound	Act. Behavior	Run Time
quicksortD	$2a$	$O(a)$	0.1002 s
insertionsortD	$2a$	$O(a)$	0.0449 s
mergesort	0	$O(1)$	0.1626 s
pairs	$-3a + 3a^2$	$O(a^2)$	0.0461 s
triples	$4.6a + -7a^2 + 2.3a^3$	$O(a^3)$	0.2588 s
quadruples	$-6a + 11a^2 - 6a^3 + 1a^4$	$O(a^4)$	1.6634 s
dyade	$-a + 3a^2 + b + b^2$	$O(a \cdot b)$	0.1609 s
lcs	$-a + a^2 + 4b + 2b^2$	$O(a \cdot b)$	1.4311 s
mult	$8a + 2b$	$O(a + b)$	0.2058 s
div	$6a + 2b$	$O(a)$	0.9720 s
eratos	$2a$	$O(a)$	0.0398 s
startBreadth	$14a + 14a^2$	$O(a^2)$	1.0148 s

triples. The blue lines are plots of the computed bounds and the red crosses show the measured worst-case resource consumption for several input lengths. The plots for the other functions are available on the website.

Many of the tight resource bounds contain negative and rational coefficients. This shows how both resource polynomials, i.e. binomial coefficients, and rational coefficients are used to derive tight resource bounds.

9 Resource-Annotated Types and Sized Types

An application of the inference algorithm for resource-annotated types in RAML is the inference of bounds on the size of the output of a function in terms of the sizes of its arguments. This size information can be used in turn to infer the resource consumptions of a larger class of functions.

Hughes, Pareto and Sabry [17] have introduced the notion of *sized types* which are a framework for type systems that provides information on the size of the data. The basic idea of sized types is to enrich the recursive data types with a number that represents the size of data structures. This information can be used to prove termination [1], to find errors [17] or to state bounds on the resource consumption of functional programs [16, 10].

In this section we provide an inference method that computes polynomial relations between the sizes of the input and the size of the output of a function. This method is not limited to shapely functions (see §1) but can infer sized types for every function that can be analyzed with our method.

Sized types are usually defined in terms of denotational semantics. Here, we rely on an intuitive, informal semantics of sized types. A formalization along the lines of [17] would be possible. Below we give the grammar of sized types for RAML and an informal explanation of what a sized type statement means. The size types that we will infer are *existential sized types* as defined by Xi [22].

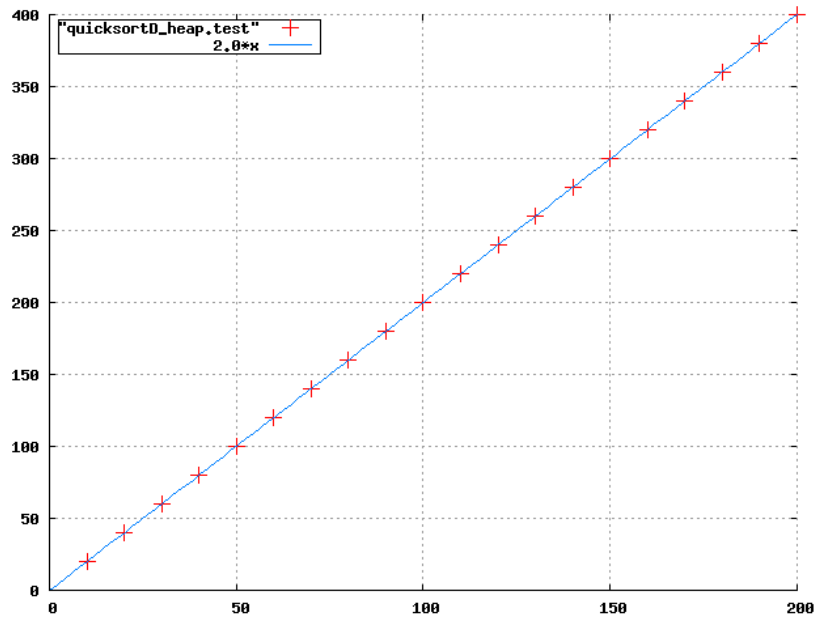


Fig. 1. The computed heap-space bound (blue line) compared to the measured worst-case heap-space consumption (red crosses) of *quicksort*.

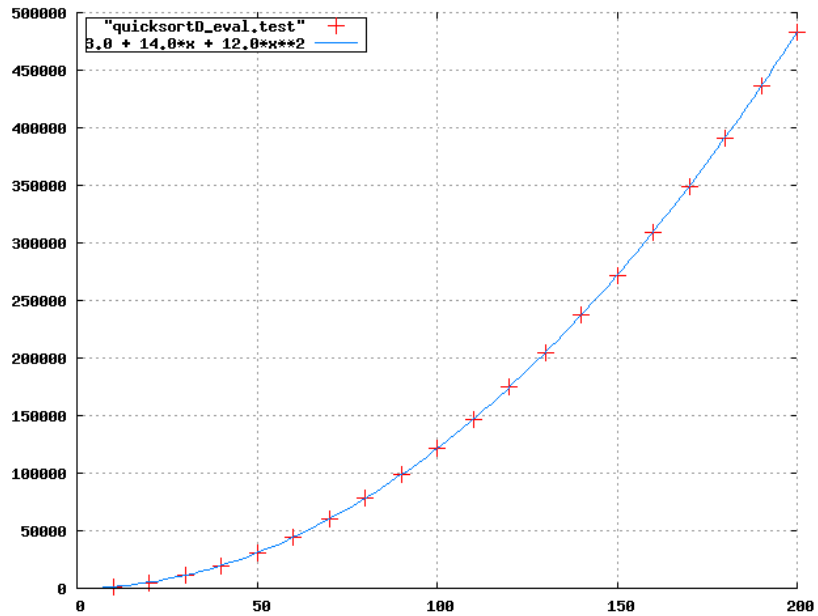


Fig. 2. The computed evaluation-step bound (blue line) compared to the actual worst-case number of evaluation-steps (red crosses) used by *quicksort*.

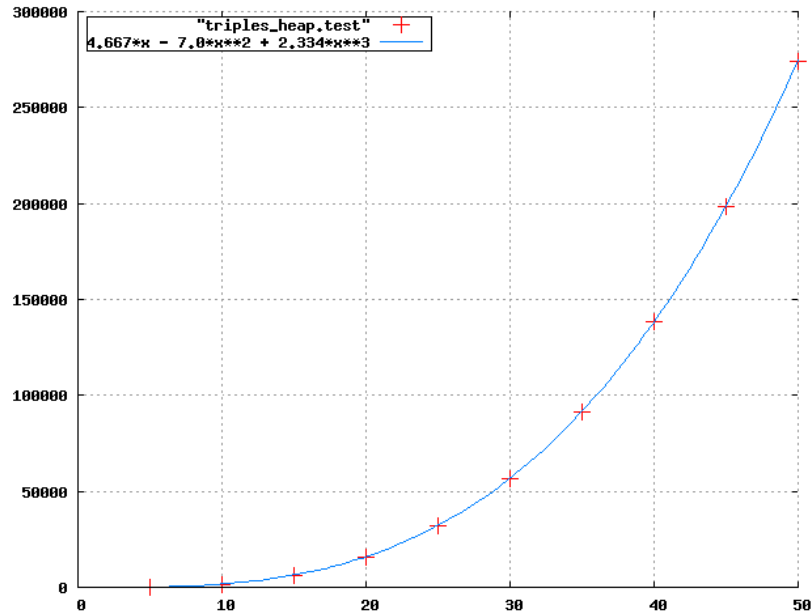


Fig. 3. The computed heap-space bound (blue line) compared to the measured worst-case heap-space consumption (red crosses) of *triples*.

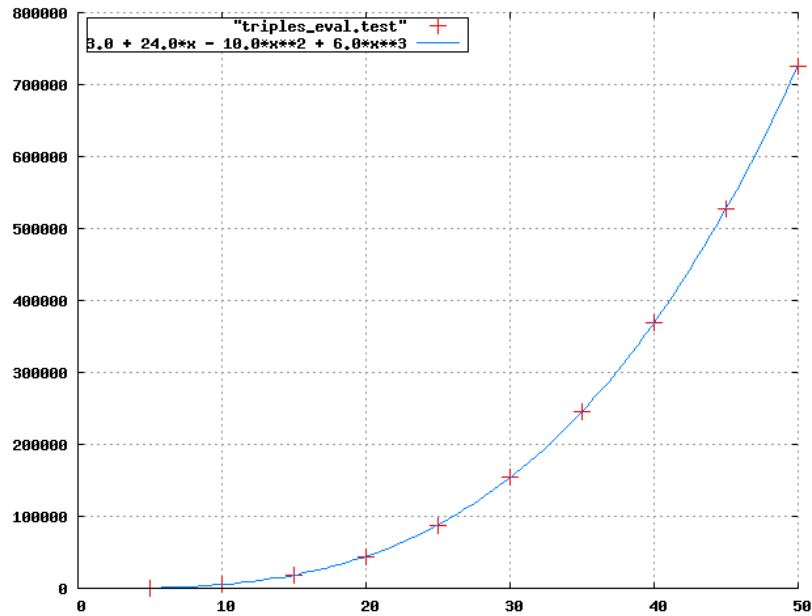


Fig. 4. The computed evaluation-step bound (blue line) compared to the actual worst-case number of evaluation-steps (red crosses) used by *triples*.

Here we present our results only for lists over non-inductive data types (simple lists) in this extended abstract. All results can be extended to arbitrary RAML types but this would require more involved notations without introducing additional ideas.

The *sized zero-order types with simple lists* for RAML are defined by

$$C ::= L_x(D) \mid (C, C) \quad D ::= \text{unit} \mid \text{bool} \mid \text{int} \mid (D, D)$$

Herein x is a size variable. Let $\text{Var}(C)$ be the set of size variables that occur in a sized zero-order type C .

Let C, C_1, \dots, C_k be sized zero-order types with $\bigcup \text{Var}(C_i) = \{x_1, \dots, x_s\}$, $\text{Var}(C) = \{y_1, \dots, y_t\}$, $\text{Var}(C_i) \cap \text{Var}(C_j) = \emptyset$ for $i \neq j$ and $\bigcup \text{Var}(C_i) \cap \text{Var}(C) = \emptyset$. Let furthermore f_i be polynomials, $c_{ij} \in \{0, 1\}$ and let ψ be a constraint of the form $\psi = \bigwedge_{i=1, \dots, r} \sum_{j=1, \dots, t} c_{ij} y_j \leq f_i(x_1, \dots, x_s)$. Then

$$[\forall \vec{x}](C_1, \dots, C_k) \rightarrow [\exists \vec{y}. \psi] C$$

is a *sized first-order type*.

Signatures Σ and typing contexts Γ with respect to sized types are defined similar as in the case of simple types. If Γ is a typing context for sized types then $\text{Var}(\Gamma) = \bigcup \{\text{Var}(C) \mid C \in \text{dom}(\Gamma)\}$ is the set of size variables that occur in Γ .

We say that a value v *respects* a sized type C under a valuation $\alpha : \text{Var}(C) \rightarrow \mathbb{N}$ if it has the size that is stated by the type with respect to α . That is the case if one of the following statements applies.

- $C \in \{\text{int}, \text{bool}, \text{unit}\}$ and v is a value of type C
- $C = (C_1, C_2)$, $v = (v_1, v_2)$ and v_i respects C_i under α for $i=1, 2$
- $C = L_n(D)$, $v = [v_1, \dots, v_{n'}]$ is a list of length $n' \leq n$ and v_i respects D under α for $i \in \{1, \dots, n'\}$

Let Σ be a signature, Γ a typing context with $\text{Var}(\Gamma) = \{x_1, \dots, x_s\}$, let e a RAML expression and let A be a sized zero-order type with $\text{Var}(A) = \{y_1, \dots, y_t\}$. A type statement with respect to size types has then the form

$$[\forall \vec{x}] \Gamma \Vdash_{\Sigma} e : [\exists \vec{y}. \bigwedge_{i=1}^r \sum_{j=1}^t c_{ij} y_j \leq f_i(x_1, \dots, x_s)] A$$

for polynomials f_1, \dots, f_r and $c_{ij} \in \{0, 1\}$. The informal meaning of this statement is the following. In an environment in which the variables in the context Γ respect their corresponding sized types for some $x_1, \dots, x_s \in \mathbb{N}$ there are $y_1, \dots, y_t \in \mathbb{N}$ such that $\bigwedge_{i=1, \dots, r} \sum_{j=1, \dots, t} c_{ij} y_j \leq f_i(x_1, \dots, x_s)$ and e evaluates to a value that respects the sized type A under this valuation.

To see how the resource types of RAML can be used to derive sized types consider the function *pairs* that has been defined in §2. We argued that we can derive the annotated type

$$\text{pairs}: L^{(0,4)}(\text{int}) \xrightarrow{0/0} L^{(1)}(\text{int}, \text{int})$$

that bounds the heap-space consumption of pairs . Let n be the length of the list that is the result of an evaluation of $y = \text{pairs}(x)$. The potential of y is then $\Phi(y:L^{(1)}(\text{int}, \text{int})) = n$ and it can be paid for from two sources. First, from the potential of the argument x of $\text{pairs}(x)$ and, secondly, from the resources that became available during the execution of $\text{pairs}(x)$. The latter can for example be the case for heap-cells when we add a destructive pattern-matching to our language which is straightforward and only omitted for simplicity.

In order to get bounds on the size of the result of a function we want to be in a situation where the potential of the result can only be paid from the potentials of the inputs, and where we furthermore do not have to deal with any resource consumption during the execution. This can be easily achieved by instantiating the resource-annotated type system to a trivial resource where all constants K that appear in the type rules are set to zero. We call this trivial resource *cost-free* and write

$$f:A \xrightarrow{CF(q/q')} B \quad \text{and} \quad \Sigma; \Gamma \xrightarrow{CF \frac{q}{q'}} e:A$$

for typing judgments that refer to the cost-free resource.

For the function pairs it is for example possible to infer the cost-free type

$$\text{pairs}: L^{(0,1)}(\text{int}) \xrightarrow{CF(0/0)} L^{(1)}(\text{int}, \text{int}).$$

From the above cost-free type we conclude that the potential $\Phi(y:L^{(1)}(\text{int}, \text{int})) = n$ of a result $y = \text{pairs}(x)$ of length n has been paid exclusively from the potential $\Phi(x:L^{(0,1)}(\text{int}))$ of the argument x . Thus we can conclude that $n \leq \Phi(x:L^{(0,1)}(\text{int})) = \binom{m}{2}$ where m is the length of x . In other words we have inferred the sized type

$$\text{pairs}: [\forall m] L_m(\text{int}) \rightarrow [\exists n. n \leq \binom{m}{2}] L_n(\text{int}, \text{int}).$$

To prove this observation for the general case we introduce some notations. For a resource-annotated type A we inductively define a sized type \bar{A} and a polynomial $f_A : \text{Var}(\bar{A}) \rightarrow \mathbb{Q}$.

$$\begin{aligned} \bar{A} &= A \text{ and } f_A = 0 \text{ if } A \in \{\text{unit}, \text{bool}, \text{int}\} \\ \overline{(A_1, A_2)} &= (\bar{A}_1, \bar{A}_2) \text{ and } f_{(A_1, A_2)} = f_{A_1} + f_{A_2} \\ \overline{L_y(B)} &= L_y(\bar{B}) \text{ and } f_{L_y(B)} = f_B + \phi(y, \vec{p}) \text{ for a } y \notin \text{Var}(\bar{B}) \end{aligned}$$

A *resource-annotated type with 0-1 annotations* (0-1 type) is a resource-annotated type A such that the vectors $(0, \dots, 0)$ and $(1, 0, \dots, 0)$ are the only annotations that occur in A . For a 0-1 type A we have that f_A is a sum $y_1 + \dots + y_k$ with $y_i \in \text{Var}(A)$. If for example $A = (L^0(\text{int}), L^1(\text{int}), L^1(\text{int}))$ then $\bar{A} = (L_{y_1}(\text{int}), L_{y_2}(\text{int}), L_{y_3}(\text{int}))$ and $f_A = y_2 + y_3$.

Let Γ be a resource-annotated typing context. Then $\bar{\Gamma}$ is defined by $\bar{\Gamma}(x) = \overline{\Gamma(x)}$ and the polynomial $f_\Gamma : \text{Var}(\bar{\Gamma}) \rightarrow \mathbb{Q}$ is defined by $f_\Gamma = \sum_{x \in \text{dom}(\Gamma)} f_{\Gamma(x)}$ such that $\text{Var}(\bar{\Gamma}(x)) \cap \text{Var}(\bar{\Gamma}(y)) = \emptyset$ if $x \neq y$. Consider for instance $\Gamma = a : L^{(0,4)}(\text{int}), b : L^3(\text{int})$. Then $\bar{\Gamma} = a : L_{x_1}(\text{int}), b : L_{x_2}(\text{int})$ and $f_\Gamma(x_1, x_2) = 4\binom{x_1}{2} + 3x_2$.

Theorem 2. *Let e be an expression and $\Sigma; \Gamma \stackrel{c}{\vdash}_0^q e : A$ be a cost-free resource-annotated type statement involving only simple lists. Let furthermore $\text{Var}(\overline{\Gamma}) = \{x_1, \dots, x_s\}$, $\text{Var}(\overline{A}) = \{y_1, \dots, y_t\}$ and $\text{Var}(\overline{\Gamma}) \cap \text{Var}(\overline{A}) = \emptyset$. Then*

$$[\forall x_1, \dots, x_s] \overline{\Gamma} \Vdash_{\Sigma} e : [\exists y_1, \dots, y_t. f_A \leq f_{\Gamma} + q] \overline{A}$$

Theorem 2 shows how our resource-annotated type system can be applied to derive sized types. As a result we can infer sized types for the same wide range of polynomially bounded functions for which we can infer resource bounds. It is of course also possible to combine multiple constraints into a conjunction of constraints.

Proposition 1. *Let e be a RAML expression and C be a sized type. It is true that $[\forall \vec{x}] \Gamma \Vdash_{\Sigma} e : [\exists \vec{y}. \varphi] C$ and $[\forall \vec{x}] \Gamma \Vdash_{\Sigma} e : [\exists \vec{y}. \psi] C$ if and only if $[\forall \vec{x}] \Gamma \Vdash_{\Sigma} e : [\exists \vec{y}. \varphi \wedge \psi] C$.*

10 Passing Non-Linear Potential

An unsatisfying limitation of the type rules that have been presented in §6 is that they fail to assign super-linear potential to the result of some basic functions that can be typed with linear potential. We show here how to use the methods that have been presented in §9 to overcome this shortcoming by adding a super-linear potential to an expression that already carries a linear potential.

Like in the previous section we present the results only for simple lists for simplicity.

Consider for example the function *append*. With the rules from §6 we are able to derive a type of the form

$$\text{append} : (L^{(p+1)}(\text{int}), L^{(p)}(\text{int})) \stackrel{c/c}{\rightarrow} L^{(p)}(\text{int}).$$

This typing reflects the fact that *append* consumes one resource unit per element of the first list. More importantly, the above type also shows how to pass linear potential to the result of *append*. That is why we can derive an annotated type for an expression like **let** $z = \text{append}(x, y)$ **in** $\text{append}(x, z)$.

However, it is impossible to assign a super-linear potential to the result of *append* even though this can be semantically sound given an appropriate potential for the arguments. For example, the following (sound) quadratic type would be needed to type the expression **let** $z = \text{append}(x, y)$ **in** $\text{pairs}(z)$.

$$\text{append} : L^{(4,12)}(\text{int}), L^{(3,12)}(\text{int}) \stackrel{c/c}{\rightarrow} L^{(0,3)}(\text{int})$$

In the linear case it is easy to assign potential to a conjoint list $z = \text{append}(x, y)$ since the potential $p \cdot |z|$ of the list z is simply the sum of the potentials $p \cdot |x| + p \cdot |y|$ of the lists x and y . But this is not true in the quadratic case since in general $p \cdot |z|^2 \neq p \cdot |x|^2 + p \cdot |y|^2$.

But we have seen in §9 that we can derive size information for functions whose result has a linear resource-annotated type. We now show how one can

use this information to obtain a super-linear resource-annotated type. Assume that the sized type for the function *pairs* is given as in the previous section and that we would like to assign a resource-annotated type of the form $\textit{pairs} : \dots \xrightarrow{c/e} L^{\vec{p}}(\textit{int}, \textit{int})$. Then it is the case that the potential of the expression $\textit{pairs}(x) : L^{\vec{p}}(\textit{int}, \textit{int})$ is not more than $\phi(\binom{m}{2}, \vec{p})$ if m is the length of x .

To formalize this idea we need some notations. For a polynomial f let $\text{Var}(f)$ be the set of variables that occur in f . Let C be a sized zero-order type, $Y \subseteq \text{Var}(C)$ and $\vec{p} \in \mathbb{Q}^n$. We define inductively a resource-annotated type $A = C_Y^{\vec{p}}$ as follows.

$$\begin{aligned} C_Y^{\vec{p}} &= C \text{ if } C \in \{\textit{unit}, \textit{bool}, \textit{int}\} \\ (C_1, C_2)_Y^{\vec{p}} &= ((C_1)_Y^{\vec{p}}, (C_2)_Y^{\vec{p}}) \\ L_y(B)_Y^{\vec{p}} &= \begin{cases} L^{\vec{p}}(D_Y^{\vec{p}}) & \text{if } y \in Y \\ L^0(D_Y^{\vec{p}}) & \text{if } y \notin Y \end{cases} \end{aligned}$$

Theorem 3. *Let e be an expression, C be a sized type and $[\forall \vec{x}] \Gamma' \Vdash_{\Sigma} e : [\exists \vec{y}. y_{i_1} + \dots + y_{i_j} \leq g(\vec{x})] C$ be a sized type statement. If Γ is a resource-annotated context such that $\bar{\Gamma} = \Gamma'$ and $f_{\Gamma}(\vec{x}) + q \geq \phi(g(\vec{x}), \vec{p})$ for all $\vec{x} \in (\mathbb{Q}^+)^n$ then $\Sigma; \Gamma \text{ cF} \frac{q}{0} e : C_{\{y_{i_1}, \dots, y_{i_j}\}}^{\vec{p}}$.*

Theorem 3 and Theorem 2 from §9 can be summarized by the following typing rules for types with simple lists.

$$\frac{\begin{array}{l} A \text{ is a 0-1 type} \quad \Sigma; \Gamma \text{ cF} \frac{q}{0} e : A \quad \text{Var}(\bar{\Gamma}) = \{x_1, \dots, x_s\} \\ \text{Var}(\bar{A}) = \{y_1, \dots, y_t\} \quad \text{Var}(\bar{A}) \cap \text{Var}(\bar{\Gamma}) = \emptyset \end{array}}{[\forall x_1, \dots, x_s] \bar{\Gamma} \Vdash_{\Sigma} e : [\exists y_1, \dots, y_t. f_A \leq f_{\Gamma} + q] \bar{A}} \text{ (TOSIZED)}$$

$$\frac{\begin{array}{l} [\forall \vec{x}] \Gamma' \Vdash_{\Sigma} e : [\exists \vec{y}. y_{i_1} + \dots + y_{i_j} \leq g(\vec{x})] A \\ \bar{\Gamma} = \Gamma' \quad f_{\Gamma}(\vec{x}) + q \geq \phi(g(\vec{x}), \vec{p}) \end{array}}{\Sigma; \Gamma \text{ cF} \frac{q}{0} e : A_{\{y_{i_1}, \dots, y_{i_j}\}}^{\vec{p}}} \text{ (FROMSIZED)}$$

If we use the cost-free resource we can derive the resource-annotated type

$$\Sigma; l1 : L^{(1)}(\textit{int}), l2 : L^{(1)}(\textit{int}) \text{ cF} \frac{0}{0} \textit{append}(l1, l2) : L^{(1)}(\textit{int}).$$

With the rule TOSIZED we derive then the sized type

$$[\forall x, y] l1 : L_x(\textit{int}), l2 : L_y(\textit{int}) \Vdash_{\Sigma} \textit{append}(l1, l2) : [\exists z. z \leq x + y] L_z(\textit{int})$$

Now we would like to assign a quadratic potential $\vec{p} = (0, 3)$ to $\textit{append}(l1, l2)$. Therefore we have to find an annotated context $\Gamma' = l1 : L^{\vec{q}}(\textit{int}), l2 : L^{\vec{r}}(\textit{int})$ such that $f_{\Gamma'} = \phi(x, \vec{q}) + \phi(y, \vec{r}) \geq \phi(x + y, (0, 3)) = 3 \cdot \binom{x+y}{2}$. It is for example true that

$$\begin{aligned} 3 \binom{x+y}{2} &\leq 3 \binom{2x}{2} + 3 \binom{2y}{2} = 3 \left(4 \binom{x}{2} + x \right) + 3 \left(4 \binom{y}{2} + y \right) \\ &= \phi(x, (3, 12)) + \phi(y, (3, 12)) \end{aligned}$$

That is why we can use the rule FROMSIZED to derive

$$\Sigma; l1:L^{(3,12)}(int), l2:L^{(3,12)}(int) \text{ cF} \frac{0}{0} \text{ append}(l1,l2):L^{(0,3)}(int).$$

Moreover it is not hard to prove the following rule correct in the sense of Theorem 1.

$$\frac{\Sigma; \Gamma' \vdash_{\frac{c}{c}} e : A' \quad \Sigma; \Gamma'' \text{ cF} \frac{q}{0} e : A'' \quad \forall (A \mid (A', A'')) \quad \forall (\Gamma \mid (\Gamma', \Gamma''))}{\Sigma; \Gamma \vdash_{\frac{c+q}{c}} e : A} \text{ (COMBINE)}$$

Here the sharing relation $\forall (\Gamma \mid (\Gamma', \Gamma''))$ is generalized to contexts in a per element way. The rule COMBINE can be used to obtain the following type statement for heap-space consumption.

$$l1:L^{(p+4,12)}(int), l2:L^{(p+3,12)}(int) \vdash_{\frac{c}{c}} \text{ append}(l1,l2):L^{(p,3)}(int)$$

In the following we combine the rules TOSIZED, FROMSIZED and COMBINE into a modified let rule that can be used for type inference. Let A be a resource-annotated zero-order type. We write $|A|$ for the simple type that can be derived from A by removing all annotations. We define the *sequence of annotations* $\text{Ann}(A)$ that appear in A inductively as follows.

$$\begin{aligned} \text{Ann}(A) &= () \text{ if } A \in \{\text{unit}, \text{bool}, \text{int}\} \\ \text{Ann}((A_1, A_2)) &= (\vec{p}_i)_{1 \leq i \leq s+t} \text{ if } \text{Ann}(A_1) = (\vec{p}_i)_{1 \leq i \leq s} \text{ and } \text{Ann}(A_2) = (\vec{p}_i)_{s+1 \leq i \leq t} \\ \text{Ann}(L^{\vec{p}}(B)) &= (\vec{p}, \vec{p}_1, \dots, \vec{p}_s) \text{ if } \text{Ann}(B) = (\vec{p}_i)_{1 \leq i \leq s} \end{aligned}$$

Let $f(n)$ be a hereditarily non-negative polynomial (see end of §5). We define $\text{co}(f(n)) = \vec{p}$ to be the *coordinates* of $f(n)$ with respect to the basis $b_k(n) = \phi(n, e_k)$, i.e., $f(n) = \sum_{i=1}^k \binom{n}{i} p_i$.

Suppose we have given a 0-1 type A such that $\text{Ann}(A) = e_i$ and $f_A = y$, i.e., A has only one list type with the annotation $(1, 0, \dots, 0)$. Let furthermore $\Sigma; \Gamma \text{ cF} \frac{q}{0} e:A$ be a type judgment that involves only simple lists. Then we can deduce for the sized-type judgment $[\forall \vec{x}] \bar{\Gamma} \Vdash_{\Sigma} e: [\exists \vec{y}] \bar{A}$ that

$$f_A \leq q + f_{\Gamma} = q + \sum_{i=1, \dots, s} \phi(x_i, \vec{p}_i)$$

provided that $\text{Ann}(A) = (p_i)_{1 \leq i \leq s}$ and $(x_i)_{1 \leq i \leq s}$ is an appropriate enumeration of $\text{Var}(\bar{\Gamma})$.

Let $u = |\{\vec{p}_i \mid \vec{p}_i \neq \vec{0}\}|$. Then we can conclude that

$$\begin{aligned} \phi(y, \vec{q}) &\leq \phi\left(\sum_{i=1, \dots, s} \phi(x_i, \vec{p}_i), \vec{q}\right) \leq \sum_{i=1, \dots, s} \phi(u \cdot \phi(x_i, \vec{p}_i), \vec{q}) \\ &\leq \sum_{i=1, \dots, s} \phi(\phi(x_i, u \cdot \vec{p}_i), \vec{q}) \end{aligned}$$

Let $f_i(x) = \phi(\phi(x, u \cdot \vec{p}_i), \vec{q})$. We define $Pass(\Gamma, A, \vec{q}) = (\text{co}(f_1), \dots, \text{co}(f_s))$. Note that the coordinates of the polynomials f_i can be computed in polynomial time by standard techniques from linear algebra.

This and the previous discussion shows the correctness of the rule T:LET-PASS for types with simple lists that is defined below.

$$\frac{\begin{array}{c} \Sigma; \Gamma_0 \vdash \frac{r_0 - K_1^{\text{let}}}{p} e_1 : A_0 \\ \Sigma; \Gamma_2, x:A \vdash \frac{p - K_2^{\text{let}}}{q' + K_3^{\text{let}}} e_2 : B \quad \forall (A \mid A_0, A') \quad \forall (\Gamma \mid \Gamma_0, \dots, \Gamma_n) \\ \text{Ann}(A') = (\vec{q}_1, \dots, \vec{q}_s) \quad q = r_0 + \dots + r_n \quad \forall 1 \leq i \leq s : \\ \Sigma; \Gamma'_i \stackrel{cF}{\vdash}_0 e_1 : A'_i \quad \text{Ann}(A'_i) = e_i \quad \text{Ann}(\Gamma_i) = Pass(\Gamma'_i, A'_i, \vec{q}_i) \end{array}}{\Sigma; \Gamma, \Gamma_2 \vdash \frac{q}{q'} \text{let } x = e_1 \text{ in } e_2 : B}$$

As mentioned above a very similar rule can be formulated for arbitrary RAML types. The rule T:LET-PASS will be used by the inference algorithm in §11. It can also be used to type functions that pass more potential through the arguments of a recursive call than was attached to the original arguments. This allows the inference of a limited resource-polymorphic recursion that can be used for instance to infer the typing of the function *pairs'* from §7.

11 Inference of Annotated Types

In the following we present a heuristic which infers resource-annotated types. This inference algorithm is not complete with respect to the declarative rules in the earlier sections. However, it does find types for most example programs that we considered, including all programs in this paper. Additionally, we believe that completeness is not vitally important because the declarative rules are a somewhat arbitrary choice and for decidability reasons there does not exist a semantically complete algorithm anyway.

The type inference for RAML is achieved by multiple runs of a basic type-inference algorithm that works similar to the algorithm of Hofmann and Jost that has been developed for the linear system [15]. The basic algorithm does a classic type inference instrumented with linear constraints for the annotations that are collected during the inference, and that can be solved later by linear programming. We use the function *pairs* as an illustrative example that shows how this algorithm can be applied to infer resource-annotated types with the rules that have been presented in §6. The only difference to the method of Hofmann and Jost is that we have to provide a maximal degree of the resource bounds in order to obtain a finite set of equations. This degree can either be specified by the user or can be incremented successively after an unsuccessful analysis. In most cases it should be sufficient to run the analysis for instance twice, first with a maximal degree of, say, 5 and a second time with maximal degree 10.

Furthermore we show how multiple runs of the basic algorithm can be combined with the rule T:LET-PASS to infer types for a greater class of functions.

$$\begin{array}{c}
\frac{\text{pairs} : L^{(p_1, p_2, p_3)}(\text{int}) \xrightarrow{c/c'} L^{(q_1, q_2, q_3)}(\text{int}) \quad (\text{T:FUNAPP}) \quad \frac{\text{append} : \dots}{\text{nps} : L^{(v_1, v_2, v_3)}(\text{int}, \text{int}), \text{rps} : L^{(u_1, u_2, u_3)}(\text{int}, \text{int})} \quad (\text{T:FUNAPP})}{\frac{\frac{e}{e'} \text{pairs}(xs^2) : L^{(u_1, u_2, u_3)}(\text{int}, \text{int}) \quad \frac{d}{d'} \text{append}(\text{nps}, \text{rps}) : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})}{xs^2 : L^{(t_1, t_2, t_3)}(\text{int}), \text{nps} : L^{(v_1, v_2, v_3)}(\text{int}, \text{int}) \quad \frac{b}{b'} \text{let rps} = \text{pairs}(xs^2) \text{in} \text{append}(\text{nps}, \text{rps}) : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})} \quad (\text{T:LET})} \\
\frac{\text{attach} : \dots}{x : \text{int}, xs^1 : L^{(s_1, s_2, s_3)}(\text{int}) \quad \frac{a}{a'} \text{attach}(x, xs^1) : L^{(v_1, v_2, v_3)}(\text{int}, \text{int})} \quad (\text{T:FUNAPP}) \quad \begin{array}{c} \vdots \\ \vdots \end{array} \\
\frac{x : \text{int}, xs^1 : L^{(s_1, s_2, s_3)}(\text{int}), xs^2 : L^{(t_1, t_2, t_3)}(\text{int}) \quad \frac{c_2}{c_2'} \text{let nps} = \text{attach}(x, xs^1) \text{in} \text{let rps} = \text{pairs}(xs^2) \text{in} \text{append}(\text{nps}, \text{rps}) : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})}{\frac{c_1}{c_1'} \text{nil} : L^{(q_1, q_2, q_3)}(\text{int}, \text{int}) \quad x : \text{int}, xs : L^{(r_1, r_2, r_3)}(\text{int}) \quad \frac{c_2}{c_2'} \text{let nps} = \text{attach}(x, xs) \text{in} \text{let rps} = \text{pairs}(xs) \text{in} \text{append}(\text{nps}, \text{rps}) : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})} \quad (\text{T:SHARE}) \\
\frac{\frac{c_1}{c_1'} \text{nil} : L^{(q_1, q_2, q_3)}(\text{int}, \text{int}) \quad x : \text{int}, xs : L^{(r_1, r_2, r_3)}(\text{int}) \quad \frac{c_2}{c_2'} \text{let nps} = \text{attach}(x, xs) \text{in} \text{let rps} = \text{pairs}(xs) \text{in} \text{append}(\text{nps}, \text{rps}) : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})}{l : L^{(p_1, p_2, p_3)}(\text{int}) \quad \frac{c}{c'} e_{\text{pairs}} : L^{(q_1, q_2, q_3)}(\text{int}, \text{int})} \quad (\text{T:RELAX/MATCH-L})
\end{array}$$

$$\begin{array}{l}
c - K_1^{\text{matchN}} \geq 0 \quad c - K_1^{\text{matchN}} - c_1 \leq c' + K_2^{\text{matchN}} - c_1' \\
c_1 \geq c - K_1^{\text{matchN}} \quad c + p_1 - K_1^{\text{matchC}} - c_2 \leq c' + K_2^{\text{matchC}} - c_2' \\
c_2 \geq c + p_1 - K_1^{\text{matchC}} \quad c + p_1 - K_1^{\text{matchC}} \geq 0 \\
r_1 = p_1 + p_2 \quad r_2 = p_2 + p_3 \quad r_3 = p_3 \quad s_1 + t_1 = r_1 \\
s_2 + t_2 = r_2 \quad s_3 + t_3 = r_3 \quad a = c_2 - K_1^{\text{let}} \quad b = a' - K_2^{\text{let}} \\
b' = c_2' + K_3^{\text{let}} \quad e = b - K_1^{\text{let}} \quad d = e' - K_2^{\text{let}} \quad d' = b' + K_3^{\text{let}} \\
p_1 = t_1 \quad p_2 = t_2 \quad p_3 = t_3 \quad q_1 = u_1 \\
q_2 = u_2 \quad q_3 = u_3 \quad e = c + K_1^{\text{app}} \quad e' = c' - K_2^{\text{app}}
\end{array}$$

Fig. 5. Type inference for the function *pairs* and the resulting linear constraints. The rule T:RELAX has been combined with the rules that occur before the rule T:MATCH-L.

Fig. 5 shows the basic inference of the function *pairs* and the linear constraints that correspond to the derivation. Additionally to the constraints in Fig. 5 we have $v \geq 0$ for every variable v . Note that the rule T:LET-PASS is not used in the derivation. The resource-annotations of the types contain variables that range over \mathbb{Q}^+ .

In addition there are the constraints that follow from the derivation of *append* and *attach*. Consider e.g. our heap-space example where we have $K^{\text{pairs}} = K^{\text{cons}} = 1$ and $K = 0$ for all other constants K . It then follows from the derivation of *append* that in the above example also $v_1 + 1 \geq q_1$, $u_1 \geq q_1$, $u_2 \geq q_2$, $u_3 \geq q_3$, $\triangleleft(v_1, v_2, v_3) = (v_1, v_2, v_3)$ and $\triangleleft(q_1, q_2, q_3) = (q_1, q_2, q_3)$ and therefore it follows that $v_2 = v_3 = q_2 = q_3 = 0$. Similar, from the derivation of *attach* it follows that $s_1 + 2 \geq v_1$ and $v_2 = v_3 = s_2 = s_3 = 0$. That is why the basic algorithm infers only a linear type to the output of *pairs*. Note that the solutions to the above constraint set include the ones that we have given in §7.

We show now how we can use the rule T:LET-PASS to infer a super-linear potential for the output of a function. Roughly speaking, we will analyze every function in a program twice. First with respect to the cost-free resource and secondly with respect to the resource R we are actually interested in.

A *cost-free 0-1 type inference* for a function f is defined as follows. Fix a maximal degree d . Infer a type derivation with a set C of constraints with respect

to the cost-free resource like in Fig. 5. Let A be the result type of f and let $\text{Ann}(A) = (\vec{p}_1, \dots, \vec{p}_k)$ with $\vec{p}_i = (p_{i1}, \dots, p_{id})$. Let C_i be the constraint set C together with the equations $p_{i1} = 1$ and $p_{jl} = 0$ if $j \neq i$ or $l \neq 1$. We then compute k solutions to the refined constraint sets $C_1 \dots, C_k$ and obtain k different typings of f . The result of a cost-free 0-1 type inference is thus a 0-1 typing of the function as needed for the rule T:LET-PASS. We call it a *cost-free 0-1 type* of f .

Note that a cost-free 0-1 type of a function f can already be helpful in a cost-free 0-1 type inference of another function that calls f . The full inference algorithm works therefore as follows.

Heuristic 1 *Let P be a RAML program, let R be a resource (i.e., a set of constants K) and fix a maximal degree d for the bounds.*

1. *Compute the strongly connected components (SCCs) of the call graph of the functions in P .*
2. *In the topological order of the SCCs, do for every SCC C : Compute simultaneously the cost-free 0-1 types with maximal degree d of the functions in C with the basic algorithm. Use the rule T:LET-PASS instead of T:LET after a function call $f(\vec{x})$ if a cost-free 0-1 type of f has been inferred already.*
3. *Do a type-inference with the basic algorithm for the whole program with respect to the resource R . Use the rule T:LET-PASS instead of T:LET after a function call.*

12 Adding Trees

One can define a potential for trees of degree d similar as for lists. For a resource annotation $\vec{p} = (p_1, \dots, p_k)$ and an annotated type A let $T^{\vec{p}}(A)$ be the type of trees of degree d . If l is a location that points to a tree of type $T^{\vec{p}}(A)$ in a well-formed heap \mathcal{H} then we have $\mathcal{H}(l) = \text{NULL}$ or $\mathcal{H}(l) = (v, l_1, \dots, l_d)$ such that v matches the type A and l_i points to a tree of type $T^{\vec{p}}(A)$ for all i .

We extend the definition of the potential function Φ for such a $l:T^{\vec{p}}(A)$ through

$$\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) = p_1 + \Phi_{\mathcal{H}}(v:A) + \sum_{i=1}^d \Phi_{\mathcal{H}}(l_i: T^{\prec(\vec{p})}(A)).$$

As in the case of lists we give a closed formula of the potential of a tree. This formula is more complicated for trees since the actual potential of a tree depends heavily on its layout.

Lemma 4 (Closed Formula for Trees). *Let \mathcal{H} be a heap and let l be a location that points to a tree t of height h with nodes v_1, \dots, v_n such that n_i is the number of nodes at level i . If $\vec{p} = (p_1, \dots, p_k)$ is an annotation and $p_i = 0$ for $i > k$ then*

$$\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) = \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^h n_i \left(\sum_{j=1}^{i+1} p_j \binom{i}{j-1} \right)$$

Proof. We prove the statement by induction on h . If $h = 0$ then $n = 0$ and the statement follows directly from the definition of Φ .

Let now $h > 0$. Then

$$\begin{aligned}
\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) &= p_1 + \Phi_{\mathcal{H}}(v_1:A) + \sum_{i=1}^d \Phi_{\mathcal{H}}(t_i: T^{\triangleleft(\vec{p})}(A)) \\
&= p_1 + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^{h-1} n_{i+1} \left(\sum_{j=1}^{i+1} (p_j + p_{j+1}) \binom{i+1}{j-1} \right) \\
&= p_1 + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=2}^h n_i \left(\sum_{j=1}^i (p_j + p_{j+1}) \binom{i-1}{j-1} \right) \\
&= p_1 + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) \\
&\quad + \sum_{i=2}^h n_i \left(p_1 + p_{i+1} + \sum_{j=2}^i p_j \left(\binom{i-1}{j-2} + \binom{i-1}{j-1} \right) \right) \\
&= \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^h n_i \left(\sum_{j=1}^{i+1} p_j \binom{i+1}{j-1} \right)
\end{aligned}$$

Lemma 5 shows two bounds that might be presented to a user after the analysis.

Lemma 5. *Let $\vec{p} = (p_1, \dots, p_k)$, let \mathcal{H} be a heap and let l be a location that points to a tree t of height h with nodes v_1, \dots, v_n .*

1. $\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) \leq \phi(n, \vec{p}) + \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A)$
2. $\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) \leq \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^k p_i \cdot n \cdot (h+1)^{i-1}$

Proof. Part 1 follows by induction on n and from the fact that $\phi(n_1, \vec{p}) + \phi(n_2, \vec{p}) \leq \phi(n_1 + n_2, \vec{p})$.

To prove part 2 let n_i be the number of nodes on level i . It follows from Lemma 4 that $\Phi_{\mathcal{H}}(l:T^{\vec{p}}(A)) =$

$$\begin{aligned}
&\sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^h n_i \left(\sum_{j=1}^{i+1} p_j \binom{i+1}{j-1} \right) \\
&\leq \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + \sum_{i=1}^h n_i \left(\sum_{j=1}^{h+1} p_j \binom{h+1}{j-1} \right) \\
&\leq \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + n \left(\sum_{j=1}^{h+1} p_j \binom{h+1}{j-1} \right) \leq \sum_{i=1}^n \Phi_{\mathcal{H}}(v_i:A) + n \left(\sum_{j=1}^k p_j (h+1)^{j-1} \right)
\end{aligned}$$

Below are the typing rules for the construction and destruction of trees.

$$\frac{\Gamma' = x_0:A, x_1:T^{\triangleleft(\vec{p})}(A), \dots, x_d:T^{\triangleleft(\vec{p})}(A) \quad \vec{p} = (p_1, \dots, p_k) \quad \Sigma; \Gamma \vdash \frac{q - K_1^{\text{matchNT}}}{q' + K_1^{\text{matchNT}}} e_1 : B \quad \Sigma; \Gamma, \Gamma' \vdash \frac{q + p_1 - K_1^{\text{matchCT}}}{q' + K_2^{\text{matchCT}}} e_2 : B}{\Sigma; \Gamma, x:T^{\vec{p}}(A) \vdash \frac{q}{q'} \text{ match } x \text{ with } | \text{ nilT } \rightarrow e_1 : B \quad | \text{ consT}(x_0, \dots, x_d) \rightarrow e_2 : B} \quad (\text{T:MATCH-T})$$

$$\frac{A \in \mathcal{A}}{\Sigma; \emptyset \vdash \frac{K^{\text{nilT}}}{0} \text{ nilT} : T(A)} \quad (\text{T:NIL-T})$$

$$\frac{\vec{p} = (p_1, \dots, p_k)}{\Sigma; x_0:A, x_1:T^{\triangleleft(\vec{p})}(A), \dots, x_d:T^{\triangleleft(\vec{p})}(A) \vdash \frac{p_1 + K^{\text{consT}}}{0} \text{ consT}(x_0, \dots, x_d) : T^{\vec{p}}(A)} \quad (\text{T:CONS-T})$$

We will discuss trees in more detail in a subsequent work.

As an example consider the function *trans* that is defined below. For a tree *t* of degree 2 the expression *trans(t, [])* evaluates to a list *l* such that (x, y) is in *l* if and only if *x* is an ancestor of *y* in *t*. In other words *trans(t, [])* computes the transitive closure of *t*.

```
attach(y,t,acc) = match t with | nilT → acc
                  | consT(x,t1,t2) → let acc1 = attach(y,t1,acc) in
                                     let acc2 = attach(y,t2,acc1) in (y,x)::acc2
```

```
attach: (int, T(p+2)(int), Lp(int))  $\xrightarrow{c/\varepsilon}$  Lp(int)
```

```
trans(t,acc) = match t with | nilT → acc
                   | consT(x,t11,t21) → let acc1 = attach(x,t12,acc) in
                                           let acc2 = attach(x,t22,acc1) in
                                           let acc3 = trans(t13,acc2) in
                                           trans(t23,acc3)
```

```
attach: (T(0,p+2)(int), Lp(int))  $\xrightarrow{c/\varepsilon}$  Lp(int)
```

```
acc1,acc2,acc3:Lp(int) t11,t21:T(p+2,p+2)(int) t12,t22:T(p+2)(int) t13,t23:T(0,p+2)(int)
```

13 Conclusion and Related Work

We have extended amortized resource analysis for first-order functional programs from linear bounds to polynomial bounds.

The main technical innovations of our paper are as follows: 1) the representations of resource bounds as non-negative linear combinations of binomial coefficients enabling a simple and local typing rule for pattern matching; 2) the derivation of constraints solvable by linear programming in spite of the super-linear bounds; 3) the inference of size information on the output of functions and the use of this information to improve amortized analysis.

Most closely related is of course [15] which we extend to polynomial bounds. Other resource analyses that can in principle obtain polynomial bounds are

approaches based on recurrences pioneered by Grobauer [10] and Flajolet [8]. In those systems, an a priori unknown resource bounding function is introduced for each function in the code; by a straightforward intraprocedural analysis a set of recurrence equations or inequations for these functions is then derived. A type-based extraction of such recurrences has been given in [7]. Even for relatively simple programs the resulting recurrences are quite complicated and difficult to solve with standard methods. In the COSTA project [2] progress has been made with the solution of those recurrences. Still, we find that amortization yields better results in cases where resource usage of intermediate functions depends on factors other than input size, e.g., sizes of partitions in QuickSort. Also compositions of functions seem to be better dealt with by amortization.

A successful method to estimate time bounds for C++ procedures with loops and recursion was recently developed by Gulwani et al. [12, 11] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. If the loops iterate over data-structures then the user needs to define numerical “quantitative functions” for the data-structures. In contrast our method is fully automatic. A methodological difference is that we infer (using linear programming) an abstract potential function which indirectly yields a resource-bounding function. As explained in the introduction the potential-based approach may be favorable in the presence of compositions and data scattered over different locations (partitions in QuickSort). Indeed, the examples from *loc. cit.* suggest that the two approaches are complementary in the sense that the method of Gulwani et al. works well for programs with little or no recursion but intricate interaction of linear arithmetic with loops. Our method, on the other hand, does not model the interaction of integer arithmetic with resource usage, but is particularly good for analyzing recursive programs involving inductive data types. As any type system, our approach is naturally compositional and lends itself to the smooth integration of components whose implementation is not available. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [3].

However, we find the possibility of incorporating existing program analyses to be a particularly attractive feature of the SPEED approach. It would be interesting to investigate to what extent such analyses could also be harnessed for our method. Another pragmatic but interesting aspect is the use of slicing techniques to eliminate large code portions that do not contribute to the resource being analyzed.

Another related approach is the use of sized types [17, 16, 5, 6] which provide a general framework to represent the size of the data in its type. Sized types are a very important concept and we also employ them indirectly. Our method adds a certain amount of data dependency and dispenses with the explicit manipulation of symbolic expressions in favour of numerical potential annotations. As we have demonstrated, there is a fruitful interaction between sized types and amortization.

Polynomial resource bounds have also been studied in [20]. Interestingly, the motivation of that paper is to extend amortized analysis to super-linear bounds; however loc. cit. only addresses the derivation of polynomial size bounds which is identified there as a necessary precursor to amortized analysis. Moreover, the analysis is restricted to functions whose exact growth rate is polynomial, and efficiency of inference remains unclear.

Acknowledgments We thank Klaus Aehlig, Lennart Beringer, Robert Grabowski, Markus Latte and Hans-Wolfgang Loidl for helpful discussions and suggestions that improved this work.

References

1. Andreas Abel. Semi-continuous Sized Types and Termination. *Log. Methods Comput. Sci.*, 4(2), 2008.
2. Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15th Intl. Symp. (SAS'08)*, pages 221–237, 2008.
3. Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic Certification of Heap Consumption. In *Logic for Prog., AI, and Reasoning, 11th Int. Conf. (LPAR'04)*, pages 347–362, 2004.
4. Brian Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *18th Euro. Symp. on Prog. (ESOP'09)*, pages 190–204, 2009.
5. Wei-Ngan Chin and Siau-Cheng Khoo. Calculating Sized Types. *High.-Ord. and Symb. Comp.*, 14(2-3):261–300, 2001.
6. Wei-Ngan Chin, Siau-Cheng Khoo, Shengchao Qin, Corneliu Popeea, and Huu Hai Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *Intl. Conf. on Software Eng. (ICSE'05)*, pages 186–195, 2005.
7. Karl Crary and Stephanie Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.
8. Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic Average-case Analysis of Algorithms. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.
9. Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded Linear Logic. *Theoret. Comput. Sci.*, 97(1):1–66, 1992.
10. Bernd Grobauer. Cost Recurrences for DML Programs. In *6th Intl. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.
11. Bhargav S. Gulavani and Sumit Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Comp. Aid. Verification, 20th Int. Conf. (CAV '08)*, pages 370–384, 2008.
12. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.
13. K. Hammond, R. Dyckhoff, C. Ferdinand, R. Heckmann, M. Hofmann, H.-W. Loidl, G. Michaelson, J. Sérot, and A. Wallace. The EmBounded Project: Automatic Prediction of Resource Bounds for Embedded Systems. In *Trends in Fun. Prog., Vol 6*, 2006.
14. Kevin Hammond and Greg Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Intl. Conf. on Generative Prog. and Component Eng. (GPCE'03)*, pages 37–56. LNCS 2830, 2003.

15. Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.
16. John Hughes and Lars Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *4th Intl. Conf. on Funct. Prog. (ICFP'99)*, pages 70–81, 1999.
17. John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Symp. Princ. of Prog. Langs. (POPL'96)*, pages 410–423, 1996.
18. Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010. To appear.
19. Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Intl. Symp. on Form. Meth. (FM'09)*, pages 354–369, 2009.
20. Olha Shkaravska, Ron van Kesteren, and Marko C. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calc. Apps. (TLCA'07)*, pages 351–365, 2007.
21. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.
22. Hongwei Xi. Dependent ML: an Approach to Practical Programming with Dependent Types. *J. Funct. Programming*, 17(2):215–286, 2007.