Amortized Resource Analysis with Polynomial Potential

A Static Inference of Polynomial Bounds for Functional Programs

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 [1] 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 [1] 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. It can also 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 [1, 2] has been successfully applied in the domain of embedded systems [3]. 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 online[1]. 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, 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

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, insertionsort, pairs and triples). Plots of our experiments are available online and in the extended version of this article.

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.

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

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

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 outlines how the typing of an sub-expressions can be improved in order to type complex expressions. The inference algorithm is presented in §9 and §10 gives an overview of the related work.

An extended version of this paper is available on the first authors web page. In addition, we show there how our system can be extended to trees and how it can be applied to infer sized types. Furthermore, it contains a compilation of the experimental results and more detailed versions of §7, §8 and §9.

2 Amortized Analysis: Examples and Intuition

Amortized analysis was initially introduced by Sleator and Tarjan [4] 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 [1] 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:*$(int, L(int))\to L(int, int)$ 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

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

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

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

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 [1]. 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 [2] that the same potential-based approach can be similarly applied to a wide range of resources such as time and stack space [5] as well as to polymorphic, higher-order programs [6].

Now consider the function *pairs*:$L(\mathit{int}) \to L(\mathit{int},\mathit{int})$ 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

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

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

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

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 $\lhd$ of a type annotation which is defined through $\lhd(q_1, \dots, q_k) = (q_1 + q_2, \dots, q_{k-1} + q_k, q_k)$ to reflect the identity from item 3. It is for instance present in the typing $\mathit{tail}{:}L^{\vec{q}}(\mathit{int}) \xrightarrow{0/q_1} L^{\lhd(\vec{q})}(\mathit{int})$ 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 $\mathit{xs}{:}L^{\lhd(\vec{q})}(\mathit{int})$ of a list $(x{::}\mathit{xs}){:}L^{\vec{q}}(\mathit{int})$ (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 $\mathit{xs}{:}L^{(4,4)}(\mathit{int})$ is assigned to the variable *xs*. The potential is then shared between the two occurrences of *xs* in the following expression by using $\mathit{xs}{:}L^{(4,0)}(\mathit{int})$ to pay for *append* and *attach* (ii) and using $\mathit{xs}{:}L^{(0,4)}(\mathit{int})$ 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 §9). We are nevertheless currently investigating an extension to arbitrary multivariate polynomials.

3 RAML – A Functional Programing Language

In this section we define the functional first-order language RAML (Resource Aware ML). RAML is similar to LF (linear functional language) from [1]. 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 $x \in \mathrm{VID}$.

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

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 *cons(x,y)*.

We restrict our attention mainly to *list types* in this paper. However, we discuss extensions to other algebraic data types in an extended version of this article that is available on the web.

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.

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

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 $\Gamma : \mathrm{VID} \to \mathcal{A}_S$ 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 $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. We thus have the implicit side condition $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$ whenever Γ_1, Γ_2 occurs in a typing rule.

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

The typing judgment $\Gamma \vdash_\Sigma e : A$ states that the expression e has type A under the signature Σ in the context Γ. Due to space restrictions we omit the typing rules that define the typing judgment. They are standard and identical

with the resource-annotated typing rules T:Const - T:Share from §6 if the resource annotations are omitted.

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

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

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

$$v ::= l \mid b \mid n \mid \textsc{Null} \mid (v, v)$$

Thus a value $v \in \mathrm{Val}$ is either a location $l \in \mathrm{Loc}$, 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 $\mathcal{H} : \mathrm{Loc} \to \mathrm{Val}$ that maps locations to values. A *stack* is a finite partial mapping $\mathcal{V} : \mathrm{VID} \to \mathrm{Val}$ from variables to values.

The rules below define an evaluation judgment of the form $\mathcal{V}, \mathcal{H} \vdash_{q'}^{q} e \rightsquigarrow v, \mathcal{H}'$ 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 $\mathcal{V}, \mathcal{H} \vdash_{q'}^{q} e \rightsquigarrow v, \mathcal{H}'$ and $\mathcal{V}, \mathcal{H} \vdash_{p'}^{p} e \rightsquigarrow v, \mathcal{H}'$ 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 $K^{\mathrm{pair}} = K^{\mathrm{cons}} = 1$ 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 [1] which is omitted here for simplicity.

$$\frac{}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\mathrm{unit}}} () \rightsquigarrow \textsc{Null}, \mathcal{H}}\ (\textsc{E:Const-U}) \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{\mathcal{V}, \mathcal{H} \vdash_{q}^{q+K^{\mathrm{bool}}} b \rightsquigarrow b, \mathcal{H}}\ (\textsc{E:Const-B})$$

$$\frac{n \in \mathbb{Z}}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{\text{int}}}{q}} n \rightsquigarrow n, \mathcal{H}} \text{ (E:Const-I)} \qquad \frac{x \in \text{dom}(\mathcal{V})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{\text{var}}}{q}} x \rightsquigarrow \mathcal{V}(x), \mathcal{H}} \text{ (E:Var)}$$

$$\frac{op \in \{+, -, *, mod, div, and, or\} \quad x_1, x_2 \in \text{dom}(\mathcal{V}) \quad v = op(\mathcal{V}(x_1), \mathcal{V}(x_2))}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{op}}{q}} x_1 \ op \ x_2 \rightsquigarrow v, \mathcal{H}} \text{ (E:BinOp)}$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_k) \to A \quad \forall 1 \le i \le n : \mathcal{V}(x_i) = v_i \quad [y_1^f \mapsto v_1, \ldots, y_k^f \mapsto v_k], \mathcal{H} \vdash^{\frac{q - K_1^{\text{app}}}{q' + K_2^{\text{app}}}} e_f \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} f(x_1, \ldots, x_k) \rightsquigarrow v, \mathcal{H}'} \text{ (E:FunApp)}$$

$$\frac{\mathcal{V}, \mathcal{H} \vdash^{\frac{q_1 - K_1^{\text{let}}}{q_2}} e_1 \rightsquigarrow v_1, \mathcal{H}_1 \quad \mathcal{V}[x \mapsto v_1], \mathcal{H}_1 \vdash^{\frac{q_2 - K_2^{\text{let}}}{q_3 + K_3^{\text{let}}}} e_1 \rightsquigarrow v_2, \mathcal{H}_2}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q_1}{q_3}} let \ x = e_1 \ in \ e_2 \rightsquigarrow v_2, \mathcal{H}_2} \text{ (E:Let)}$$

$$\frac{\mathcal{V}(x) = True \quad \mathcal{V}, \mathcal{H} \vdash^{\frac{q - K_1^{\text{conT}}}{q' + K_2^{\text{conT}}}} e_t \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} if \ x \ then \ e_t \ else \ e_f \rightsquigarrow v, \mathcal{H}'} \text{ (E:Cond-T)}$$

$$\frac{\mathcal{V}(x) = False \quad \mathcal{V}, \mathcal{H} \vdash^{\frac{q - K_1^{\text{conF}}}{q' + K_2^{\text{conF}}}} e_f \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} if \ x \ then \ e_t \ else \ e_f \rightsquigarrow v, \mathcal{H}'} \text{ (E:Cond-F)}$$

$$\frac{x_1, x_2 \in \text{dom}(\mathcal{V}) \quad v = (\mathcal{V}(x_1), \mathcal{V}(x_2)) \quad l \notin \text{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{\text{pair}}}{q}} (x_1, x_2) \rightsquigarrow l, \mathcal{H}[l \mapsto v]} \text{ (E:Pair)}$$

$$\frac{\mathcal{V}(x) = l \quad \mathcal{H}(l) = (v_1, v_2) \quad \mathcal{H}, \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash^{\frac{q - K_1^{\text{matchP}}}{q' + K_2^{\text{matchP}}}} e \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} match \ x \ with \ (x_1, x_2) \to e \rightsquigarrow v, \mathcal{H}'} \text{ (E:Match-P)}$$

$$\frac{}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{\text{nil}}}{q}} nil \rightsquigarrow \text{Null}, \mathcal{H}} \text{ (E:Nil)}$$

$$\frac{x_h, x_t \in \text{dom}(\mathcal{V}) \quad v = (\mathcal{V}(x_1), \mathcal{V}(x_2)) \quad l \notin \text{dom}(\mathcal{H})}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q + K^{\text{cons}}}{q}} cons(x_h, x_t) \rightsquigarrow l, \mathcal{H}[l \mapsto v]} \text{ (E:Cons)}$$

$$\frac{\mathcal{V}(x) = \text{Null} \quad \mathcal{H}, \mathcal{V} \vdash^{\frac{q - K_1^{\text{matchN}}}{q' + K_2^{\text{matchN}}}} e_1 \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} match \ x \ with \mid nil \to e_1 \mid cons(x_h, x_t) \to e_2 \rightsquigarrow v, \mathcal{H}'} \text{ (E:Match-N)}$$

$$\frac{\mathcal{V}(x) = l \quad \mathcal{H}(l) = (v_h, v_t) \quad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H} \vdash^{\frac{q - K_1^{\text{matchC}}}{q' + K_2^{\text{matchC}}}} e_2 \rightsquigarrow v, \mathcal{H}'}{\mathcal{V}, \mathcal{H} \vdash^{\frac{q}{q'}} match \ x \ with \mid nil \to e_1 \mid cons(x_h, x_t) \to e_2 \rightsquigarrow v, \mathcal{H}'} \text{ (E:Match-C)}$$

Actual constants for stack-space, heap-space and clock-cycle consumption have been determined for the abstract machine of the language Hume [7] on the Renesas M32C/85U architecture. A list can be found in the literature [2].

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 $\mathit{attach}\colon (\mathit{int}, L^{(2)}(\mathit{int})) \xrightarrow{0/0} L^{(0)}(\mathit{int}, \mathit{int})$ to the function to indicate that the evaluation of *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 $\mathit{pairs}\colon L^{(0,5)}(\mathit{int}) \xrightarrow{0/0} L^{(2)}(\mathit{int}, \mathit{int})$. Informally, this type says: To evaluate *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.

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

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

For two resource annotations $\vec{p} = (p_1, \ldots, p_k)$ and $\vec{q} = (q_1, \ldots, q_l)$ 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 $\vec{p} + \vec{q} = (p_1 + q_1, \ldots, p_k + q_k, q_{k+1}, \ldots, q_l)$.

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

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.

- $\Phi_{\mathcal{H}}(\textsc{Null}{:}A) = 0$
- If $A \in \{\mathit{unit}, \mathit{int}, \mathit{bool}\}$ 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 $A = L^{(p_1,\ldots,p_k)}(A')$ is a list type and $v = l$ is a location with $\mathcal{H}(l) = (v', l')$ then $\Phi_{\mathcal{H}}(l{:}A) = p_1 + \Phi_{\mathcal{H}}(v'{:}A') + \Phi_{\mathcal{H}}(l'{:}\, L^{\lhd(p_1,\ldots,p_k)}(A'))$.

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 $\mathcal{H}(l_1) = [v_1, \ldots, v_n]$ if $\mathcal{H}(l_i) = (v_i, l_{i+1})$ for $i = 1, \ldots, n$ and $l_{n+1} = \textsc{Null}$. If $l_1 = \textsc{Null}$ then we write $\mathcal{H}(l_1) = []$.

Let for example $\mathcal{H}$ be a heap and $\mathcal{H}(v) = [v_1 \ldots, v_n]$ an integer list. Then

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

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

The proof of Lemma 1 as well as the proofs of the following lemmas are given in the extended version of this article.

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 $\alpha = \phi(n, (p_1, p_2, \cdots, p_k))$ can be read as follows: a recursive function on a list l of length n that has the potential α can use the potential $\phi(i, (p_2, \cdots, p_k)$ 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 $\vec{p} = (p_1, \ldots, p_k)$ be a resource annotation, let $n \in \mathbb{N}$ and define $\phi(n, ()) = 0$. Then $\phi(n, (p_1, \ldots, p_k)) = n \cdot p_1 + \sum_{i=1}^{n-1} \phi(i, (p_2, \ldots, p_k))$.*

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 $f : \mathbb{N} \to \mathbb{N}$ 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* [8] 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 §8 we add another rule.

As in the case of the simple types, a *typing context* is a partial finite function $\Gamma : \mathrm{VID} \to \mathcal{A}$ 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

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

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

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

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

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* $\Sigma : \mathrm{FID} \to \mathcal{F}$ is a finite, partial mapping of function identifiers to resource-annotated first-order types. A *resource-annotated typing judgment* has the form $\Sigma; \Gamma \vdash^{q}_{q'} e{:}A$ 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 $(e_f, \vec{y}^f)_{f \in \mathrm{dom}(\Sigma)}$ such that for each e_f we have $\Sigma; y^f_1{:}A_1, \ldots, y^f_k{:}A_k \vdash^{q}_{q'} e_f{:}A$ if $\Sigma(f){=}(A_1, \ldots, A_k) \xrightarrow{q/q'} A$.

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.

$$\frac{}{\Sigma; \emptyset \vdash^{K^{\mathrm{unit}}}_{0} ():\mathit{unit}} \ (\text{T:Const-U}) \qquad \frac{n \in \mathbb{Z}}{\Sigma; \emptyset \vdash^{K^{\mathrm{int}}}_{0} n : \mathit{int}} \ (\text{T:Const-I})$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\}}{\Sigma; \emptyset \vdash^{K^{\mathrm{bool}}}_{0} b{:}\mathit{bool}} \ (\text{T:Const-U}) \qquad \frac{}{\Sigma; x{:}A \vdash^{K^{\mathrm{var}}}_{0} x : A} \ (\text{T:Var})$$

$$\frac{\mathit{op} \in \{+, -, *, \mathit{mod}, \mathit{div}\}}{\Sigma; x_1{:}\mathit{int}, x_2{:}\mathit{int} \vdash^{K^{\mathrm{op}}}_{0} x_1 \ \mathit{op} \ x_2 : \mathit{int}} \ (\text{T:BinOp-I})$$

$$\frac{\mathit{op} \in \{\mathit{or}, \mathit{and}\}}{\Sigma; x_1{:}\mathit{bool}, x_2{:}\mathit{bool} \vdash^{K^{\mathrm{op}}}_{0} x_1 \ \mathit{op} \ x_2 : \mathit{bool}} \ (\text{T:BinOp-B})$$

$$\frac{\Sigma(f) = (A_1,\dots,A_k) \xrightarrow{q/q'} A \qquad \Gamma = x_1{:}A_1,\dots,x_k{:}A_k}{\Sigma;\Gamma \vdash^{q+K_1^{\mathrm{app}}}_{q'-K_2^{\mathrm{app}}} f(x_1,\dots,x_k) : A} \text{ (T:FunApp)}$$

$$\frac{\Sigma;\Gamma_1 \vdash^{q-K_1^{\mathrm{let}}}_{p} e_1 : A \qquad \Sigma;\Gamma_2, x{:}A \vdash^{p-K_2^{\mathrm{let}}}_{q'+K_3^{\mathrm{let}}} e_2 : B}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{q}_{q'} \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : B} \text{ (T:Let)}$$

$$\frac{\Sigma;\Gamma \vdash^{q-K_1^{\mathrm{conT}}}_{q'+K_2^{\mathrm{conT}}} e_t : A \qquad \Sigma;\Gamma \vdash^{q-K_1^{\mathrm{conF}}}_{q'+K_2^{\mathrm{conF}}} e_f : A}{\Sigma;\Gamma, x{:}\mathit{bool} \vdash^{q}_{q'} \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f : A} \text{ (T:Cond)}$$

$$\frac{}{\Sigma; x_1{:}A_1, x_2{:}A_2 \vdash^{K^{\mathrm{pair}}}_{0} (x_1,x_2) : (A_1,A_2)} \text{ (T:Pair)}$$

$$\frac{A = (A_1,A_2) \qquad \Sigma;\Gamma, x_1{:}A_1, x_2{:}A_2 \vdash^{q-K_1^{\mathrm{matchP}}}_{q'+K_2^{\mathrm{matchP}}} e : B}{\Sigma;\Gamma, x{:}A \vdash^{q}_{q'} \mathit{match}\ x\ \mathit{with}\ (x_1,x_2) \to e : B} \text{ (T:Match-P)}$$

$$\frac{A \in \mathcal{A}}{\Sigma;\emptyset \vdash^{K^{\mathrm{nil}}}_{0} \mathit{nil}{:}L(A)} \text{ (T:Nil)}$$

$$\frac{\vec{p} = (p_1 \dots p_k)}{\Sigma; x_h{:}A, x_t{:}L^{\lhd(\vec{p})}(A) \vdash^{p_1+K^{\mathrm{cons}}}_{0} \mathit{cons}(x_h,x_t){:}L^{\vec{p}}(A)} \text{ (T:Cons)}$$

$$\frac{\begin{array}{l}\vec{p} = (p_1,\dots,p_k) \qquad \Sigma;\Gamma \vdash^{q-K_1^{\mathrm{matchN}}}_{q'+K_2^{\mathrm{matchN}}} e_1 : B \\ \Sigma;\Gamma, x_h{:}A, x_t{:}L^{\lhd(\vec{p})}(A) \vdash^{q+p_1-K_1^{\mathrm{matchC}}}_{q'+K_2^{\mathrm{matchC}}} e_2 : B\end{array}}{\Sigma;\Gamma, x{:}L^{\vec{p}}(A) \vdash^{q}_{q'} \begin{array}{l}\mathit{match}\ x\ \mathit{with} \\ \mid\ \mathit{nil} \to e_1 \\ \mid\ \mathit{cons}(x_h,x_t) \to e_2\end{array} : B} \text{ (T:Match-L)}$$

$$\frac{\Sigma;\Gamma, x{:}A_1, y{:}A_2 \vdash^{q}_{q'} e : B \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma;\Gamma, z{:}A \vdash^{q}_{q'} e[z/x, z/y] : B} \text{ (T:Share)}$$

$$\frac{\Sigma;\Gamma, x{:}A \vdash^{q}_{q'} e : B \qquad A' <: A}{\Sigma;\Gamma, x{:}A' \vdash^{q}_{q'} e : B} \text{ (T:Super)} \qquad \frac{\Sigma;\Gamma \vdash^{q}_{q'} e : B \qquad B <: B'}{\Sigma;\Gamma \vdash^{q}_{q'} e : B'} \text{ (T:Sub)}$$

$$\frac{\Sigma;\Gamma \vdash^{p}_{p'} e : B \qquad q \geq p \qquad q - p \geq q' - p'}{\Sigma;\Gamma \vdash^{q}_{q'} e : B} \text{ (T:Relax)}$$

$$\frac{\Sigma;\Gamma \vdash^{q}_{q'} e : B \qquad x \in \mathrm{VID} \qquad A \in \mathcal{A}}{\Sigma;\Gamma, x{:}A \vdash^{q}_{q'} e : B} \text{ (T:Augment)}$$

The definitions of the relations $\curlyvee(. \mid ., .)$ 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 $\mathit{cons}(x_h, x_t)$, 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 $L^{\lhd(\vec{p})}(A)$ 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 $\lhd(\vec{p})$ 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

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

The Sharing Relation The sharing relation $\curlyvee(. \mid ., .)$ defines how the potential of a zero-order variable can be shared by multiple occurrences of that variable. We will have $\curlyvee(A \mid A_1, A_2)$ 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 $\curlyvee(. \mid ., .)$ is the smallest relation such that

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

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 $x \in \mathrm{dom}(\Gamma)$. We

then write $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$. It is not hard to show that if $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ and $\mathcal{V}, \mathcal{H} \overset{q}{\underset{q'}{\vdash}} e \rightsquigarrow v, \mathcal{H}'$ then also $\mathcal{H}' \vDash \mathcal{V} : \Gamma$.

Theorem 1 (Soundness). *Let Σ be the signature of a given RAML program and let e be an expression. Let $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ and let there exist some $u, u' \in \mathbb{Q}^+$ such that $\mathcal{V}, \mathcal{H} \overset{u}{\underset{u'}{\vdash}} e \rightsquigarrow v, \mathcal{H}'$. If $\Sigma; \Gamma \overset{p}{\underset{p'}{\vdash}} e{:}A$ and $q \geq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma) + p + r$ for a $r \in \mathbb{Q}^+$ then there is a $q' \geq \Phi_{\mathcal{H}'}(v{:}A) + p' + r$ such that $\mathcal{V}, \mathcal{H} \overset{q}{\underset{q'}{\vdash}} e \rightsquigarrow v, \mathcal{H}'$.*

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

7 Examples

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 online[2] 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

- 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

A 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). The plots of the experiments can be found on the website and in the extended version of this article. Therein, we also present a somewhat artificial example (a version of dyadic vector product) that explores some boundaries of our system.

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.

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

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 *attach(x,l)* computes a list of pairs so that x is paired with every element in the list l. The function *pairs(l)* computes a list of all (unordered) pairs that can be built from the elements of l and similarly the function *triples(l)* computes a list of all (unordered) triples.

attach(x,l) = **match** l **with** | nil → nil | (y::ys)→ **let** l' = attach(x,ys) **in** (x,y) ::l'

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

append(l,ys) = **match** l **with** | nil → ys | (x::xs) → **let** l' = append(xs,ys) **in** x::l'

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

pairs(l) = **match** l **with** | nil → nil
| (x::xs) → **let** nps = attach(x,xs^1) **in**
let rps = pairs(xs^2) **in** append(nps,rps)

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

triples(l) = **match** l **with** | nil → nil
| (x::xs) → **let** tps = pairs(xs^1) **in**
let nts = attach(x,tps) **in**
let rts = triples(xs^2) **in** append(nts,rts)

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

In the above functions it is the case that the type used for recursive calls is the same as the type of the function itself (monomorphic recursion). For example in the function *pairs* the type of *append(nps,rps)* and *rps* is identical. That is not the case in general. Suppose for example that one would swap the arguments of *append* in the last line of *pairs*:

pairs'(l) = **match** l **with** | nil → nil
| (x::xs) → **let** nps = attach(x,xs^1) **in**
let rps = pairs'(xs^2) **in** append(rps,nps)

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

The function *pairs'* is used resource polymorphically in its recursive call. That means that the resource annotation of the argument of *pairs'* differs from the annotation of the original argument. The soundness of polymorphic recursion is

unproblematic and covered by our results; the inference of resource polymorphic is restricted to special cases. See §8 and §9 which cover the present example.

At a first glance it might be surprising that the heap-space consumption of *pairs'* is not quadratic but cubic. The reason is that the heap-space consumption of *append* is linear in the length of the first argument and *append* is called $|l|$ times. In the case of *pairs* the length of the first argument is about the length $|l|$ but in the case of *pairs'* the first argument is *rps* which is quadratic in $|l|$.

Note that a run-time analysis of *pairs* and *pairs'* would result in analogous types as above with different constants. That is to say the analysis of *pairs* would result in a quadratic bound while we would get a cubic bound for *pairs'*. But in contrast to the heap-space use, the run-time of *pairs'* would be cubic even in the presence of garbage collection or in an extended system that enjoys a destructive pattern matching. So this is a nice example where our system might help a programmer to produce more efficient code.

7.2 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 $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, 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 $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$. 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' : $L^p(int) \xrightarrow{c/c} L^p(int)$ firstline: $L^p(L^q(int)) \xrightarrow{c/c} L^q(int)$
lastvals: $L^p(int) \xrightarrow{c/c} (int, int)$

```
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')
```

```
newline (y, lastline ,l) = match l with | nil → nil
    | (x::xs) → 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

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

addcolumn: $(L^p(L^q(int)), int, L^{p+q+2}(int)) \xrightarrow{c/c} L^p(L^q(int))$
newline: $(int, L^q(int), L^{q+1}(int)) \xrightarrow{c/c} L^q(int)$
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 → nil
    | (x::xs^1) → match l2^1 with | nil → nil
        | (y::ys) → let m = lcstable(xs^2,ys) in
                    let m' = addline(m,y,xs^3) in addcolumn(m',x,l2^2)
```

lcstable: $(L^{(0,q+1)}(int), L^{(2p+q+3,p+q+2)}(int)) \xrightarrow{c/c} L^p(L^q(int))$
ys :$L^{(2p+q+3,p+q+2)}(int)$, xs^3:$L^{(q+1,0)}(int)$, m':$L^p(L^q(int))$, l2^1:$L^{(p+1,p+q+2)}(int)$
xs^1:$L^{(q+1,q+1)}(int)$, xs^2:$L^{(0,q+1)}(int)$, m :$L^p(L^q(int))$, l2^2:$L^{(p+q+2)}(int)$

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

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

8 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 output of some basic functions that can be typed with a linear output type. To overcome is limitation one can use linear algebra to compute linear constraints that state how a super-linear potential can be assigned to the output of a function, provided that a function type with a linear output is given.

Due to the limited space we can only describe this idea by way of example. A formal description and a type rule is given is the extended version of this paper.

Consider the function *append*. With the rules from §6 we are able to derive a type of the form *append:* $(L^{(1)}(int), L^{(1)}(int)) \xrightarrow{0/0} L^{(1)}(int)$ if we use the *cost-free* resource metric in which all constants equal 0. Then it follows that the length of the output is bounded by $n + m$ if n and m are the lengths of the inputs of *append*. This information suffices to compute (once and for all) constraints for a super-linear output via linear algebra. In the (cost-free) quadratic case we obtain for example *append:* $(L^{(p_1,p_2)}(int), L^{(q_1,q_2)}(int)) \xrightarrow{0/0} L^{(0,r_2)}(int)$ if $4p_1 \geq r_2, 4q_1 \geq r_2, p_2 \geq r_2$ and $q_2 \geq r_2$. Such a cost-free type can then be additively combined with a typing of the function that was inferred with respect to another resource-metric by adding the numbers in the type annotations. For example, for the heap metric from §7 we obtain the typing *append:* $(L^{(4,12)}(int), L^{(3,12)}(int)) \xrightarrow{0/0} L^{(0,3)}(int)$.

9 Inference of Annotated Types

The type-inference algorithm for RAML works similar to the algorithm of Hofmann and Jost that has been developed for the linear system [1].

The basic algorithm does a classic type inference generating linear constraints for the annotations that are collected during the inference, and that can be solved later by linear programming. 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. If the degree is too low then the generated linear program is unsolvable. It 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.

In order to apply the technique that has been outlined in §8 we have to run the basic algorithm multiple times since we have to consider strongly connected components in the call graph one after another. More details are given in the extended version of this work.

The inference algorithm finds types for most example programs that we considered, including all programs in this paper. Nevertheless, it is not complete with respect to the declarative rules in the earlier sections. The reason is that it sometimes fails to infer a *resource-polymorphic* typing of a function, i.e., a typing in which the annotations of a recursive call differ from the annotations of the top-level function type. We are working on a more involved inference algorithm that is complete. However, we find that this algorithm exhibits some interesting ideas that should be explained in detail in separate work.

10 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.

Most closely related is of course [1] which we extend with polynomial bounds. Other resource analyses that can in principle obtain polynomial bounds are approaches based on recurrences pioneered by Grobauer [9] and Flajolet [10]. 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 [11]. Even for relatively simple programs the resulting recurrences are quite complicated and difficult to solve with standard methods. In the COSTA project [12] 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. [13, 14] 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 [15]. 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 [16–19] 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.

References

1. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03).

(2003) 185–197
2. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In: 16th Intl. Symp. on Form. Meth. (FM'09). (2009) 354–369
3. Hammond, K., Dyckhoff, R., Ferdinand, C., Heckmann, R., Hofmann, M., Loidl, H.W., Michaelson, G., Sérot, J., Wallace, A.: The EmBounded Project: Automatic Prediction of Resource Bounds for Embedded Systems. In: Trends in Fun. Prog., Vol 6. (2006)
4. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Algebraic Discrete Methods **6**(2) (1985) 306–318
5. Campbell, B.: Amortised Memory Analysis using the Depth of Data Structures. In: 18th Euro. Symp. on Prog. (ESOP'09). (2009) 190–204
6. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th ACM Symp. on Principles of Prog. Langs. (POPL'10). (2010) To appear.
7. Hammond, K., Michaelson, G.: Hume: a Domain-Specific Language for Real-Time Embedded Systems. In: Intl. Conf. on Generative Prog. and Component Eng. (GPCE'03), LNCS 2830 (2003) 37–56
8. Girard, J.Y., Scedrov, A., Scott, P.: Bounded Linear Logic. Theoret. Comput. Sci. **97**(1) (1992) 1–66
9. Grobauer, B.: Cost Recurrences for DML Programs. In: 6th Intl. Conf. on Funct. Prog. (ICFP'01). (2001) 253–264
10. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic Average-case Analysis of Algorithms. Theoret. Comput. Sci. **79**(1) (1991) 37–109
11. Crary, K., Weirich, S.: Resource Bound Certification. In: 27th ACM Symp. on Principles of Prog. Langs. (POPL'00). (2000) 184–198
12. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Static Analysis, 15th Intl. Symp. (SAS'08). (2008) 221–237
13. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
14. Gulavani, B.S., Gulwani, S.: A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In: Comp. Aid. Verification, 20th Int. Conf. (CAV '08). (2008) 370–384
15. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic Certification of Heap Consumption. In: Logic for Prog., AI, and Reasoning, 11th Int. Conf. (LPAR'04). (2004) 347–362
16. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Symp. Princ. of Prog. Langs. (POPL'96). (1996) 410–423
17. Hughes, J., Pareto, L.: Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In: 4th Intl. Conf. on Funct. Prog. (ICFP'99). (1999) 70–81
18. Chin, W.N., Khoo, S.C.: Calculating Sized Types. High.-Ord. and Symb. Comp. **14**(2-3) (2001) 261–300
19. Chin, W.N., Khoo, S.C., Qin, S., Popeea, C., Nguyen, H.H.: Verifying Safety Policies with Size Properties and Alias Controls. In: Intl. Conf. on Software Eng. (ICSE'05). (2005) 186–195
20. Shkaravska, O., van Kesteren, R., van Eekelen, M.C.: Polynomial Size Analysis of First-Order Functions. In: Typed Lambda Calc. Apps. (TLCA'07). (2007) 351–365