Towards Automatic Resource Bound Analysis for OCaml

Jan Hoffmann Ankush Das
Carnegie Mellon University, USA
{jhoffmann, ankushd}@cs.cmu.edu

Shu-Chun Weng
Yale University, USA
scweng@gmail.com

Abstract

This article presents a resource analysis system for OCaml programs. The system automatically derives worst-case resource bounds for higher-order polymorphic programs with user-defined inductive types. The technique is parametric in the resource and can derive bounds for time, memory allocations and energy usage. The derived bounds are multivariate resource polynomials which are functions of different size parameters that depend on the standard OCaml types. Bound inference is fully automatic and reduced to a linear optimization problem that is passed to an off-the-shelf LP solver. Technically, the analysis system is based on a novel multivariate automatic amortized resource analysis (AARA). It builds on existing work on linear AARA for higher-order programs with user-defined inductive types and on multivariate AARA for first-order programs with built-in lists and binary trees. This is the first amortized analysis, that automatically derives polynomial bounds for higher-order functions and polynomial bounds that depend on user-defined inductive types. Moreover, the analysis handles a limited form of side effects and even outperforms the linear bound inference of previous systems. At the same time, it preserves the expressivity and efficiency of existing AARA techniques. The practicality of the analysis system is demonstrated with an implementation and integration with Inria's OCaml compiler. The implementation is used to automatically derive resource bounds for 411 functions and 6018 lines of code derived from OCaml libraries, the CompCert compiler, and implementations of textbook algorithms. In a case study, the system infers bounds on the number of queries that are sent by OCaml programs to DynamoDB, a commercial NoSQL cloud database service.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability

Keywords Resource Bound Analysis, Static Analysis, Type Systems, Amortized Analysis, LP Solving, Type Inference

1. Introduction

The quality of software crucially depends on the amount of resources —such as time, memory, and energy—that are required for its execution. Statically understanding and controlling the resource usage of software continues to be a pressing issue in software development. Performance bugs are very common and are among the bugs that are most difficult to detect [40, 51]. Moreover, many security vulnerabilities exploit the space and time usage of software [21, 42].

Developers would greatly profit from high-level resource-usage information in the specifications of software libraries and other interfaces, and from automatic warnings about potentially high-resource usage during code review. Such information is particularly relevant in contexts of mobile applications and cloud services, where resources are limited or resource usage is a major cost factor.

Recent years have seen fast progress in developing frameworks for statically reasoning about the resource usage of programs. Many advanced techniques for imperative integer programs apply abstract interpretation to generate numerical invariants. The obtained *size-change information* forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [27], ranking functions [2, 6, 15, 53], recurrence relations [3, 4], and abstract interpretation itself [18, 60]. Automatic resource analysis techniques for functional programs are based on sized types [56], recurrence relations [23], term-rewriting [10], and amortized resource analysis [31, 34, 41, 52].

Despite major steps forward, there are still many obstacles to overcome to make resource analysis available to developers. On the one hand, typed functional programs are particularly well-suited for automatic resource-bound analysis since the use of pattern matching and recursion often results in a relatively regular code structure. Moreover, types provide detailed information about the shape of data structures. On the other hand, existing automatic techniques for higher-order programs can only infer linear bounds [41, 56] or rely on defunctionalization [10]. Furthermore, techniques that can derive polynomial bounds are limited to bounds that depend on predefined lists and binary trees [29, 31], or integers [15, 53]. Finally, resource analyses for functional programs have been implemented for custom languages that are not supported by mature tools for compilation and development [10, 31, 34, 41, 52, 56].

The goal of a long term research effort is to overcome these obstacles by developing Resource Aware ML (RAML), a resource-aware version of the functional programming language OCaml. RAML is based on an automatic amortized resource analysis (AARA) that derives multivariate polynomials that are functions of the sizes of the inputs. In this paper, we report three research results that are part of this effort.

1. We present the first implementation of an AARA that is integrated with an industrial-strength compiler.
2. We develop the first automatic resource analysis that infers multivariate polynomial bounds that depend on size parameters of user-defined tree-like data structures.
3. We present the first AARA that infers polynomial bounds for higher-order functions.

The techniques we develop are not tied to a particular resource but are parametric in the resource of interest. RAML infers tight bounds

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009842

for many complex example programs such as sorting algorithms with complex comparison functions, Dijkstra’s shortest-path algorithm, and the most common higher-order functions such as sequences of nested maps, and folds. The technique is naturally compositional, tracks size changes of data across function boundaries, and can deal with amortization effects that arise, for instance, from the use of a functional queue. Local inference rules generate linear constraints and reduce bound inference to off-the-shelf linear program (LP) solving, despite deriving polynomial bounds.

To ensure compatibility with OCaml’s syntax, we reuse the parser and type inference engine from Inria’s OCaml compiler [48]. We extract a type-annotated syntax tree to perform (resource preserving) code transformations and the actual resource-bound analysis. To precisely model the evaluation of OCaml, we introduce a novel operational semantics that makes the efficient handling of function closures in Inria’s compiler explicit. It can be seen as a big-step formulation of the ZINC abstract machine [45]. The semantics is complemented by a new type system that refines function types.

To express a wide range of bounds, we introduce a novel class of multivariate resource polynomials that map data of a given type to a non-negative number. The set of multivariate resource polynomials that is available for bound inference depends on the types of input data. It can be parametric in (non-negative) integers, lengths of lists, or the number of particular nodes in an inductive data structure. As a special case, a resource polynomial can contain conditional additive factors. These novel multivariate resource polynomials are a generalization of the resource polynomials that have been previously defined for lists and binary trees [31].

To deal with realistic OCaml code, we develop a novel multivariate AARA that handles higher-order functions. To this end, we draw inspirations from multivariate AARA for first-order programs [31] and linear AARA for higher-order programs [41]. However, our new solution is more than the combination of existing techniques. For instance, we infer linear bounds for the curried append function for lists, which was not previously possible [41]. Moreover, we address specifics of Inria’s OCaml compiler such as efficiently avoiding function-closure creation.

OCaml is a complex language and resource bound analysis is an undecidable problem. As a result, there are many programs for which RAML cannot derive bounds. We currently do not support several OCaml features such as modules, object-oriented features, records, calls to native libraries, nested patterns, and optional arguments. The resource analysis is limited to polynomial bounds that depend on the sizes of inductive data structures that have nodes with fixed branching factors. RAML can derive bounds for programs with exceptions, references, and arrays. However, we do not support exception handlers and cannot derive bounds that depend on the sizes of data structures that are stored in arrays or references.

We performed experiments on more than 6018 lines of OCaml code. Since RAML does not support some language features of OCaml, it is not straightforward to automatically analyze complete existing applications. However, the automatic analysis performs well on code that only uses supported language features. For instance, we applied RAML to OCaml’s standard list library *list.ml*: RAML automatically derives evaluation-step bounds for 47 of the 51 top-level functions. All derived bounds are asymptotically tight.

It is also easy to develop and analyze real OCaml applications if we keep the current capabilities of the system in mind. In Section 9, we present a case study in which we automatically bound the number of queries that an OCaml program issues to Amazon’s DynamoDB NoSQL cloud database service. Such bounds are interesting since Amazon charges DynamoDB users based on the number of queries made to the database.

Our experiments are easily reproducible: The source code of RAML, the OCaml code for the experiments, and an easy-to-use interactive web interface are available online [28]. The reviewers of the POPL’17 Artifact Evaluation found that RAML exceeded or greatly exceeded their expectations. All technical details of the theoretical development and an extensive description of the results of the experiments are available in a companion technical report [33].

```
type (’a,’b) ablist =  Acons of ’a * (’a,’b) ablist
                     | Bcons of ’b * (’a,’b) ablist
                     | Nil

let rec abmap f g abs = match abs with
  | Acons (a,abs’) → Acons(f a, abmap f g abs’)
  | Bcons (b,abs’) → Bcons(g b, abmap f g abs’)
  | Nil → Nil

let asort gt abs =
  abmap (quicksort gt) (fun x → x) abs
let asort’ gt abs =
  abmap (quicksort gt) (fun _ → raise Inv_arg) abs
let btick =
  abmap (fun a → a) (fun b → Raml.tick 2.5; b)
```

Excerpt of the RAML output for analyzing evaluation steps:

```
Simplified bound for abmap:
   3 + 12*L + 12*N
Simplified bound for asort:
   11 + 22*K*N + 13*K^2*N + 13*L + 15*N
Simplified bound asort’:
   13 + 22*K*N + 13*K^2*N + 15*N
```

Figure 1. Example bound analysis with RAML ($0.23s$ run time). When reporting the linear bound for *abmap* to the user, we assume that the higher-order arguments *f* and *g* have no resource consumption. L is the number of B-nodes, N is the number of A-nodes, and K is the maximal length of the lists in the A-nodes in the function’s arguments.

2. Overview

Before we describe the technical development, we give an overview of the challenges and achievements of our work.

Example Bound Analysis. To demonstrate user interaction with RAML, Figure 1 contains an example bound analysis. The OCaml code in Figure 1 will serve as a running example in this article. The function *abmap* is a polymorphic map function for a user-defined list that contains *Acons* and *Bcons* nodes. It takes two functions *f* and *g* as arguments and applies *f* to data stored in the A-nodes and *g* to data stored in the B-nodes. The function *asort* takes a comparison function and an A-B-list in which the A-nodes contain lists. It then uses *quicksort* (the code of *quicksort* is also automatically analyzed and available online [28]) to sort the lists in the A-nodes. The B-nodes are left unchanged. The function *asort’* is a variation of *asort* that raises an exception if it encounters a B-node in the list.

To derive a worst-case resource bound with RAML, the user needs to pick a maximal degree of the search space of polynomials and a resource metric. In the example analysis in Figure 1 we picked degree 4 and the *steps* metric which counts the number of evaluation steps in the big-step semantics. RAML reports a bound for all top-level functions in 0.23 seconds. The shown output is only an excerpt. In this case, all derived bounds are tight in the sense that there are inputs for every size that exactly result in the reported number of evaluation steps.

In the derived bound for *abmap*, RAML assumes that the resource cost of f and g is 0. So we get a linear bound. In the case of *asort* we derive a bound which is quadratic in the maximal length of the lists that are stored in the A-nodes ($22K + 13K^2$) for every A-node in the list ($(22K+13K^2)N$) plus an additional linear factor that also depends on the number of B-nodes that are simply traversed ($13L + 15N$). For *asort'* this linear factor only depends on the number of A-nodes: RAML automatically deduces that the traversal is aborted in case we encounter a B-node.

The *tick* metric can be used to derive bounds on user defined metrics. An instructive example is the function *btick*. With the tick metric, RAML derives the bound $2.5L$ where L is the number of B-nodes in the argument list. This is a tight bound on the sum of "ticks" that are executed in an evaluation of *btick*. Ticks can also be negative to express that resources become available.

Automatic Amortized Resource Analysis. The automatic resource analysis system of RAML is based on the potential method of amortized analysis [54]. The idea is to introduce potential functions that depend on data structures. At every point in the program, the available potential needs to be sufficient to pay for the cost of the next evaluation step and the potential at the next program point.

To be able to automate the analysis, we fix a set of possible potential functions for every data type. In RAML, potential functions are non-negative linear combinations of a set of base polynomials. We integrate the coefficients of these linear combinations into a type system for OCaml. To this end, we use annotated types (A, Q) in which A is a standard type[1] and Q is a family of non-negative coefficients; one for every base polynomial. Given a value a of type A, we define the potential $\Phi(a{:}(A, Q))$ as a sum $\sum_i q_i{\cdot}p_i(a)$ where q_i ranges over the coefficients in Q and p_i ranges over the base polynomials for type A. The type rules of RAML's type system manipulate the coefficients Q to ensure correct accounting as potential is assigned to new data structures or used to pay for resource usage. The advantage of this setup is that we can express the relationships between different type annotations with linear constraints. They can be collected during type inference and solved by linear programming in a similar way as ordinary type constraints are collected to be solved by unification.

Multivariate Resource Polynomials. Existing AARA systems are either limited to linear bounds [34, 41] or to polynomial bounds that are functions of the sizes of simple predefined lists and binary-tree data structures [31]. In contrast, this work presents the first analysis that can derive polynomial bounds that depend on size parameters of complex user-defined data structures.

Consider for example the function *btick* of type $(\alpha * \beta)$ ablist $\to$ $(\alpha * \beta)$ ablist in Figure 1. The base polynomials for the argument include $\binom{m}{i}{\cdot}\binom{n}{j}$ where m denotes the number of A-nodes and n denotes the number of B-nodes in the argument *abs*.[2] The derived bound $2.5n$ is represented by $\sum_{i,j} q_{(i,j)}\binom{m}{i}\binom{n}{j}$ where $q_{(0,1)} = 2.5$ and $q_{(i,j)} = 0$ otherwise. The corresponding function type is

$$btick : ((\alpha * \beta)\text{ ablist}, Q) \to ((\alpha * \beta)\text{ ablist}, P)$$

where Q contains the coefficients $q_{(i,j)}$, P contains coefficients $p_{(i,j)}$, and $p_{(i,j)} = 0$ for all (i, j). This reflects the fact that the initial potential $2.5n$ is used up after the evaluation of *btick*.

Depending on the call side of *btick*, we might need different potential annotations P and Q. Usually, we pass on some potential from the argument to the result of a function so that it can be consumed in the remainder of the evaluation. Consider for example the inner call to *btick* in an expression like *btick(btick abs)*. Here, we need to cover the cost of the inner call ($2.5n$) and pass potential on to the resulting list ($2.5n$). This is reflected by a potential annotation in which $q_{(0,1)} = 5$ and $p_{(0,1)} = 2.5$.

In general, the bounds we derive are *multivariate resource polynomials* that can take into account individual sizes of inner data structures. While it is possible to simplify the resource polynomials in the user output, it is essential to have this more precise information for intermediate results to derive tight whole-program bounds. In general, the resource bounds are built of functions that count the number of specific tuples that one can form from the nodes in a tree-like data structure. In their simplest form (i.e., without considering the data stored inside the nodes), they have the form

$$\lambda a.|\{\vec{a} \mid a_i \text{ is an } A_{k_i}\text{-node in } a \text{ and if } i < j \text{ then } a_i <^{a}_{pre} a_j\}|$$

where a is an inductive data structure with constructors $A_1, \ldots, A_m$, $\vec{a} = (a_1, \ldots, a_n)$, and $<^{a}_{pre}$ denotes the pre-order (tree traversal) on the tree a. For example, consider the aforementioned polynomial $n{\cdot}m$ for values of type *ablist* where m and n are the number A- and B-nodes, respectively. This is in fact the sum of the two base polynomials $\lambda\ell.|\{(a, b) \mid a <^{\ell}_{pre} b\}|$ and $\lambda\ell.|\{(b, a) \mid b <^{\ell}_{pre} a\}|$ for A-B-lists (a ranges over A-nodes and b ranges over B-nodes).

We are able to keep track of changes of these quantities in pattern matches and data construction fully automatically by generating linear constraints. At the same time, they allow us to accurately describe the resource usage of many common functions in the same way it has been done previously for simple types [31]. As an interesting special case, we can also derive bounds that describe the resource usage as a conditional statement. For instance, for an expression such as

```
match x with | true → quicksort y | false → y
```

we derive a bound that is quadratic in the length of *y* if *x* is *True* and constant if *x* is *False*.

Currying and Function Closures. Currying and function closures pose a challenge to automatic resource analysis systems that has not been addressed in the past. To see why, assume that we want to design a type system to verify resource usage. Now consider for example the curried append function which has the type *append* : α list $\to$ α list $\to$ α list in OCaml. At first glance, we might say that the time complexity of *append* is $O(n)$ if n is the length of the first argument. But a closer inspection of the definition of *append* reveals that this is a gross simplification. In fact, the complexity of the partial function call *app_par = append* ℓ is constant. Moreover, the complexity of the function *app_par* is linear—not in the length of the argument but in the length of the list ℓ that is captured in the function closure. We are not aware of any existing approach that can automatically derive a worst-case time bound for the curried append function. For example, previous AARA systems would fail without deriving a bound [31, 41].

In general, we have to describe the resource consumption of a curried function $f : A_1 \to \cdots \to A_n \to A$ with n expressions $c_i(a_1, \ldots, a_i)$ such that c_i describes the complexity of the computation that takes place after f is applied to i arguments $a_1, \ldots, a_i$. In Inria's OCaml implementation, the situation is even more complex since the resource usage (time and space) depends on how a function is used at its call sites. If *append* is partially applied to one argument then a function closure is created as expected. However, if *append* is applied to both of its arguments at the same time then the intermediate closure is not created and the performance of the function is even better than that of the uncurried version since we do not have to create a pair before the application.

To model the resource usage of curried functions accurately, we refine function types to capture how functions are used at their call sites. For example, *append* can have both of the types $\alpha\,\text{list} \to \alpha\,\text{list} \to \alpha\,\text{list}$ and $[\alpha\,\text{list}, \alpha\,\text{list}] \to \alpha\,\text{list}$. The first type implies that the function is partially applied and the second type implies that the function is applied to both arguments at the same time. Of course, it is possible that the function has both types (technically we achieve this using let polymorphism). For the second type, our system automatically derives tight time and space bounds

[1] This is not true in general since A can contain annotated function types but it is a good approximation to get a first impression.

[2] To be precise, these functions are sums of base polynomials.

that are linear in the first argument. However, our system fails to derive a bound for the first type. The reason is that we made the design decision not to derive bounds that asymptotically depend on data captured in function closures to keep the complexity of the system at a manageable level. Analyzing such functions would require us to significantly extend the type system; for example by introducing dependent types [43, 44].

Fortunately, *append* belongs to a large set of OCaml functions in the standard library that are defined in the form *let rec f x y z = e*. If such a function is partially applied, the only computation that happens is the creation of a closure. As a result, *eta expansion* does not change the resource behavior of programs. This means for example that we can safely replace the expression *let app_par = append ℓ in e* with the expression *let app_par x = append ℓ x in e* prior to the analysis. Consequently, we can always successfully analyze *append* by using the type $[\alpha\,\text{list}, \alpha\,\text{list}] \to \alpha\,\text{list}$.

The conditions under which a function can be analyzed can boiled down to a simple principle:

> The worst-case resource usage of a function must be expressible as a function of the sizes of its arguments.

Higher-Order Arguments. The other main challenge with higher-order resource analysis is functions with higher-order arguments. To a large extent, this problem has been successfully solved for linear resource bounds in previous work [41]. Basically, the higher-order case is reduced to the first-order case if the higher-order arguments are available. It is not necessary to reanalyze such higher-order functions for every call site since we can abstract the resource usage with a constraint system that has holes for the constraints of the function arguments. However, a presentation of the system in such a way mixes type checking with the constraint-based type inference. Therefore, we chose to present the analysis system in a more declarative way in which the bound of a function with higher-order arguments is derived with respect to a given set of resource behaviors of the argument functions.

A concrete advantage of our declarative view is that we can derive a meaningful type for a function like *map* for lists even when the higher-order argument is not available. The function *map* can have the types $(\alpha \to \beta) \to \alpha\,\text{list} \to \beta\,\text{list}$ and $[\alpha \to \beta, \alpha\,\text{list}] \to \beta\,\text{list}$. Unlike *append*, the resource usage of *map* does not depend on the size of the first argument. So both types are equivalent in our system except for the cost of creating an intermediate closure. If the higher-order argument is not available then previous systems [41] produce a constraint system that is not meaningful to a user. An innovation in this work is that we are also able to report a meaningful resource bound for *map* if the arguments are not available. To this end, we assume that the argument function does not consume resources. For example, we report in the case of *map* that the number of evaluation steps needed is $11n + 3$ and the number of heap cells needed is $4n + 2$ where n is the length of the input list. Such bounds are useful for two purposes. First, a developer can see the cost that *map* itself contributes to the total cost of a program. Second, the time bound for *map* proves that *map* is guaranteed to terminate if the higher-order argument terminates for every input.

In contrast, consider the function *rec_scheme* : $(\alpha\,\text{list} \to \alpha\,\text{list}) \to \alpha\,\text{list} \to \beta\,\text{list}$ that is defined as follows.

```
let rec rec_scheme f l =
  match l with | [] → []
               | x::xs → rec_scheme f (f l)
let g = rec_scheme tail
```

Here, RAML is not able to derive an evaluation-step bound for *rec_scheme* since the number of evaluation steps (and even termination) depends on the argument f. However, RAML derives the tight evaluation-step bound $12n + 7$ for the function g.

Effects. Our analysis handles references and arrays by ensuring that the resource cost of the program does not depend on the sizes of values that have been stored in mutable cells. While it has been shown that it is possible to extend AARA to assign potential to mutable state [8, 17], we decided not to add the feature in the current system to focus on the presentation of the main contributions. There are still a lot of interactions of the analysis with mutable state, such as deriving bounds for functions in references.

Assume for example A-B-list was an abstract type and the only function in the signature was *abmap*. How can we compute the length of the longest prefix that only consists of A-nodes? Since we are not allowed to modify the A-B-list module, we use references to compute the length as a side effect of the higher-order arguments of *abmap*. The basic idea of *max_a_prefix* is that the function *f*, which is executed for every A-node, increments a counter *c* whenever it is called. However, if we encounter a B-node then the function *g* will change *f* to have no effect by updating the reference *inc*.

```
let max_a_prefix l =
  let c = ref 0 in
  let inc_c = fun () → Raml.tick(1.0); c := !c+1 in
  let inc = ref inc_c in
  let f () = !inc () in
  let g () = inc := fun () → () in
  let _ = abmap f g l in !c
```

We added a call to the tick function in the increment function *inc_c* to count the number of additions evaluated by *max_a_prefix*. RAML automatically computes the tight bound m where m is the number of A-nodes in the input list ℓ. To derive the bound, the analysis performs a basic alias analysis to statically approximate that a function stored in the reference *inc* performs 1 tick in the worst-case. In the type system, this is expressed by fixing a (set of) potential annotations for function types in mutable structures.

As we see in Figure 1 (function *asort'*), RAML can accurately analyze programs that raise exceptions. It is natural to handle exceptions in AARA by simply allowing arbitrary potential annotations in the result type of the exception.

3. Setting the Stage

We describe and formalize the new resource analysis using Core RAML, a subset of the intermediate language that we use to perform the analysis. Expressions in Core RAML are in share-let-normal form [32], which means that syntactic forms allow only variables instead of arbitrary terms whenever possible without restricting expressivity. We automatically transform user-level OCaml programs to Core RAML without changing their resource behavior before the analysis.

Syntax. The syntax of a subset of the Core RAML expressions is given by the following grammar. The implementation also contains constants and operators for primitive data types such as integers, floats and booleans, arrays and built-in operations for arrays, conditionals, and *free* versions of syntactic forms. These free versions are semantically identical to the standard versions but do not contribute to the resource cost. This is needed for the resource preserving translation of user-level code to share-let-normal form.

$$
\begin{aligned}
e ::= {}& x \mid x\, x_1 \cdots x_n \mid C\, x \mid \lambda x.e \mid \mathsf{ref}\ x \mid {!}\,x \mid x_1 := x_2 \mid \mathsf{fail} \\
& \mid \mathsf{tick}\,(q) \mid \mathsf{match}\ x\ \mathsf{with}\ C\, y \to e_1 \mid e_2 \\
& \mid (x_1, \ldots, x_n) \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1, \ldots, x_n) \to e \\
& \mid \mathsf{share}\ x\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let\ rec}\ F\ \mathsf{in}\ e \\
F ::= {}& f = \lambda x.e \mid F_1\ \mathsf{and}\ F_2
\end{aligned}
$$

In the function application we allow the application of several arguments at once. This is useful to statically determine the cost of closure creation but also introduces ambiguity. The type system will

$$\frac{S \neq \cdot \qquad H(\ell) = (\lambda x.e, V') \qquad V(x_1){::}\ \cdots\ {::}V(x_n), V, H \;{}_M{\vdash}\; x \Downarrow \ell \mid (q, q') \qquad S, V', H \;{}_M{\vdash}\; \lambda x.e \Downarrow w \mid (p, p')}{S, V, H \;{}_M{\vdash}\; x\ x_1 \cdots x_n \Downarrow w \mid M^{\mathsf{app}}_n{\cdot}(q,q'){\cdot}(p,p')} \ (\text{E:AppApp})$$

$$\frac{V(x_1){::}\ \cdots\ {::}V(x_n), V, H \;{}_M{\vdash}\; x \Downarrow w \mid (q, q')}{\cdot, V, H \;{}_M{\vdash}\; x\ x_1 \cdots x_n \Downarrow w \mid M^{\mathsf{app}}_n{\cdot}(q,q')} \ (\text{E:App}) \qquad \frac{S \neq \cdot \qquad H(V(x)) = (\lambda x.e, V') \qquad S, V', H \;{}_M{\vdash}\; \lambda x.e \Downarrow w \mid (q, q')}{S, V, H \;{}_M{\vdash}\; x \Downarrow w \mid M^{\mathsf{var}}{\cdot}(q,q')} \ (\text{E:VarApp})$$

$$\frac{V(x) = \ell}{\cdot, V, H \;{}_M{\vdash}\; x \Downarrow (\ell, H) \mid M^{\mathsf{var}}} \ (\text{E:Var}) \qquad \frac{S, V[x \mapsto \ell], H \;{}_M{\vdash}\; e \Downarrow w \mid (q, q')}{\ell{::}S, V, H \;{}_M{\vdash}\; \lambda x.e \Downarrow w \mid M^{\mathsf{bind}}{\cdot}(q,q')} \ (\text{E:AbsBind}) \qquad \frac{H' = H, \ell \mapsto (\lambda x.e, V)}{\cdot, V, H \;{}_M{\vdash}\; \lambda x.e \Downarrow (\ell, H') \mid M^{\mathsf{abs}}} \ (\text{E:AbsClos})$$

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

determine if an expression like $f\ x_1\ x_2$ is parsed as $(f\ x_1\ x_2)$ or $(f\ x_1)\ x_2$. The sharing expressions share x as (x_1, x_2) in e is not standard and used to explicitly introduce multiple occurrences of a variable. It binds the free variables x_1 and x_2 in e. The expression *fail* is used to raise exceptions. The expression *tick(q)* contains a floating point constant q. It can be used with the tick metric to specify a constant cost. A negative floating point number q means that resources become available.

Big-Step Operational Cost Semantics. The resource usage of RAML programs is defined by a big-step operational cost semantics. The semantics has three interesting non-standard features. First, it measures (or defines) the resource consumption of the evaluation of a RAML expression by using a resource metric that defines a constant cost for each evaluation step. If this cost is negative then resources are returned. Second, it models terminating and diverging executions by inductively describing finite subtrees of infinite execution trees. Third, it models OCaml's stack-based mechanism for function application, which avoids creation of intermediate function closure, similar to the ZINC abstract machine (ZAM) [45].

Since we also consider resources like memory that can become available during an evaluation, we have to track the *high-water mark* of the resource usage, that is, the maximal number of resource units that are simultaneously used during an evaluation. To derive a high-water mark of a sequence of evaluations from the high-water marks of the sub evaluations one has to also take into account the number of resource units that are available after each sub evaluation.

Figure 2 contains selected big-step operational evaluation rules. They define an evaluation judgment of the form

$$S, V, H \;{}_M{\vdash}\; e \Downarrow (\ell, H') \mid (q, q') \ .$$

It expresses the following. If the resource metric M, the argument stack S, the environment V, and the initial heap H are given then the expression e evaluates to the location ℓ and the new heap H'. The evaluation of e needs $q \in \mathbb{Q}_0^+$ resource units (high-water mark) and after the evaluation there are $q' \in \mathbb{Q}_0^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available during the execution of e.

There are two other behaviors that we have to express in the semantics: failure (i.e., array access outside array bounds) and divergence. To this end, our semantic judgement not only evaluates expressions to values but also to an error $\bot$ and to incomplete computations expressed by $\circ$. The judgement has the general form

$$S, V, H \;{}_M{\vdash}\; e \Downarrow w \mid (q, q') \quad \text{where} \quad w ::= (\ell, H) \mid \bot \mid \circ \ .$$

Intuitively, this evaluation statement expresses that the high-water mark of the resource consumption after some number of evaluation steps is q and there are currently q' resource units left. A resource metric $M : K \times \mathbb{N} \to \mathbb{Q}$ defines the resource consumption in each evaluation step of the big-step semantics where K is the set of syntactic forms. We write M^k_n for $M(k, n)$ and M^k for $M(k, 0)$. The parameter n can be used to express the cost for steps that have different costs for different types (e.g., creating an n-tuple).

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid $\mathcal{Q} = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, \cdot)$. The neutral element is $(0, 0)$, which means that resources are neither needed before the evaluation nor returned after the evaluation. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

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

If resources are never returned (as with time) then we only have elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$. We identify a rational number q with an element of $\mathcal{Q}$ as follows: $q \geq 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$. This notation avoids case distinctions in the evaluation rules since the constants M^K that appear in the rules can be negative.

For efficiency reasons, Inria's OCaml compiler evaluates function applications $e\ e_1\ \cdots\ e_n$ from right to left, that is, it starts with evaluating e_n. In this way, one can avoid the expensive creation of intermediate function closures. A naive implementation would create n function closures when evaluating the aforementioned expression: one for e, one for the application to the first argument, etc. By starting with the last argument, we are able to put the results of the evaluation on an argument stack and access them when we encounter a function abstraction during the evaluation. In this case, we do not create a closure but simply bind the value on the stack to the name in the abstraction.

In the rules in Figure 2, we use a stack S on which we store the locations of function arguments. The only rules that push locations to S are E:App and E:AppApp. To pop locations from the stack we modify the leaf rules that can return a function closure, namely, the rules E:Var and E:Abs for variables and lambda abstractions: Whenever we would return a function closure $(\lambda x.e, V)$ we inspect the argument stack S. If S contains a location ℓ then we pop it from the stack S, bind it to the argument x, and evaluate the function body e in the new environment $V[x \mapsto \ell]$. This is defined by the rule E:AbsBind and indirectly by the rule E:VarApp.

As mentioned, these evaluation rules are a big-step version of ZAM. The benefit of using a big-step semantics is that the rules are closer to the resource-aware type rules in Section 6 and simplify the soundness proof of the type system (Theorem 1). Details such as differences between *eval-apply* and *push-enter* or ZAM and ZAM2 [46] are not modeled in the cost semantics. However, they can be accounted for by using different cost metrics. The stack-based type system we present in Section 4 performs the analysis needed for implementing eval-apply, which is also implemented by the OCaml compiler. This enables us to statically account for the cost of eval-apply in the resource-annotated type system. So our built-in cost metric for evaluation steps also models eval-apply despite the semantics being closer to push-enter.

$$\frac{}{\cdot; x : [T_1, \ldots, T_n] \to T, x_1{:}T_1, \ldots, x_n{:}T_n \vdash x\, x_1 \cdots x_n : T}\ (\text{T:App}) \qquad \frac{\Sigma; \Gamma \vdash \lambda x.e : T}{\cdot; \Gamma \vdash \lambda x.e : \Sigma \to T}\ (\text{T:AbsPop}) \qquad \frac{\Sigma; \Gamma, x{:}T_1 \vdash e : T_2}{T_1{::}\Sigma; \Gamma \vdash \lambda x.e : T_2}\ (\text{T:AbsPush})$$

$$\frac{}{\Sigma; x : [T_1, \ldots, T_n] \to \Sigma \to T, x_1{:}T_1, \ldots, x_n{:}T_n \vdash x\, x_1 \cdots x_n : T}\ (\text{T:AppPush}) \qquad \frac{}{\cdot; x : T \vdash x : T}\ (\text{T:Var}) \qquad \frac{}{\Sigma; x : \Sigma \to T \vdash x : T}\ (\text{T:VarPush})$$

Figure 3. Selected rules of the affine stack-based type system.

All evaluation rules and more details about the semantics can be found the companion technical report [33].

Example Evaluation. We use the running example defined in Figure 1 to illustrate how the operational cost semantics works. We use the metric *steps* which assigns cost 1 to every evaluation step and the metric *tick* which assigns cost 0 to every evaluation step except *Raml.tick(q)*.

Let *abs* $\equiv$ *Acons ([1;2],Bcons (3, Bcons (4, Nil)))* is a A-B-list and let e_1 the expression that arises by concatenating the expression *asort* $(>)$ *abs* to the code in Figure 1. Then for every H and V there exists H' and ℓ such that $\cdot, V, H \ {}_{\text{tick}}\!\vdash e_1 \Downarrow (\ell, H') \mid (0, 0)$ and $\cdot, V, H \ {}_{\text{steps}}\!\vdash e_1 \Downarrow (\ell, H') \mid (186, 0)$. Moreover, $\cdot, V, H \ {}_{\text{steps}}\!\vdash e_1 \Downarrow \circ \mid (n, 0)$ for every $n < 186$. Let e_2 be the expression that results from appending *btick abs* to the code in Figure 1. Then for every H and V there exists H' and ℓ such that $\cdot, V, H \ {}_{\text{tick}}\!\vdash e_2 \Downarrow (\ell, H') \mid (5, 0)$ and $\cdot, V, H \ {}_{\text{tick}}\!\vdash e_2 \Downarrow \circ \mid (n, 0)$ for every $n \in \{2.5, 0\}$.

4. Stack-Based Type System

In this section, we introduce a type system that is a refinement of OCaml's type system and mirrors the argument stack in the semantics. We define simple types as

$$\begin{aligned} T ::= &\ \text{unit} \mid X \mid T \text{ ref} \mid T_1 * \cdots * T_n \mid [T_1, \ldots, T_n] \to T \\ &\mid \mu X. \langle C_1 : T_1{*}X^{n_1}, \ldots, C_k : T_k{*}X^{n_k} \rangle \end{aligned}$$

Bracket function types $[T_1, \ldots, T_n] \to T$ correspond to the standard function type $T_1 \to \cdots \to T_n \to T$. The meaning of $[T_1, \ldots, T_n] \to T$ is that the function is applied to its first n arguments at the same time. The type $T_1 \to \cdots \to T_n \to T$ indicates that the function is applied to its first n arguments one after another. These two uses of a function can result in a very different resource behavior. For instance, in the latter case we have to create $n-1$ function closures. Also we have n different costs to account for: the evaluation cost after the first argument is present, the cost of the closure when the second argument is present, etc. Of course, it is possible that a function is used in different ways in a program. We account for that with let polymorphism. Also note that $[T_1, \ldots, T_n] \to T$ still describes a curried function while $T_1 * \cdots * T_n \to T$ describes an uncurried function with n arguments.

For inductive types $\mu X. \langle C_1 : U_1, \ldots, C_k : U_k \rangle$ we require that every constructor has a type $U_i = T_i{*}X^{n_i}$ such that T_i does not contain type variables, including X. The type X^n is simply the n-element product type $X * \cdots * X$. This makes it possible to track costs that depend on size parameters of values of such types. It is of course possible to allow arbitrary inductive types and not to track cost that depends on the size of data structures of other types. For a constructor type $C : T{*}X^n$ we sometimes write $C : (T, n)$. We say that T is the node type and n is the branching number of the constructor C.

Let Polymorphism and Sharing. Following the design of the resource-aware type system, our simple type system is affine. That means that a variable in a context can be used at most once in an expression. However, we enable multiple uses of a variable with the sharing expression $\mathsf{share}\ x\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e$ that denotes that x can be used twice in e using the (different) names x_1 and x_2. For input programs we allow multiple uses of a variable x in an expression e in RAML. We then add sharing terms and remove multiple uses of variables before the analysis.

Interestingly, this mechanism is closely related to let polymorphism. To see this relation, first note that our type system is polymorphic but that a value can only be used with a single type in an expression. In practice, that would mean for instance that we have to define a different map function for every list type. A simple and well-known solution to this problem that is often applied in practice is let polymorphism. In principle, let polymorphism replaces variables with their definitions before type checking. For our map function, we would type the expression $[\text{map} \mapsto e_{\text{map}}]e$ instead of typing the expression $\mathsf{let}\ \text{map} = e_{\text{map}}\ \mathsf{in}\ e$.

It would be possible to treat sharing of variables in a similar way as let polymorphism. But if we start from an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ and replace the occurrences of x in the expression e_2 with e_1 then we also change the resource consumption of the evaluation of e_2 because we evaluate e_1 multiple times. Interestingly, this problem coincides with the treatment of let polymorphism for expressions with side effects (the so called value restriction).

In RAML, we support let polymorphism for function closures only. Assume we have a function definition $\mathsf{let}\ f = \lambda x.e_f\ \mathsf{in}\ e$ that is used twice in e. Then the usual approach to enable the analysis in our system would be to use sharing

$$\mathsf{let}\ f = \lambda x.e_f\ \mathsf{in}\ \mathsf{share}\ f\ \mathsf{as}\ (f_1, f_2)\ \mathsf{in}\ e' \ .$$

To enable let polymorphism, we will however define f twice and ensure that we only pay once for the creation of the closure and the let binding:

$$\mathsf{let}\ f_1 = \lambda x.e_f\ \mathsf{in}\ \mathsf{let}\ f_2 = \lambda x.e_f\ \mathsf{in}\ e'$$

The functions f_1 and f_2 can now have different types.

Type Judgements. Type judgements have the form $\Sigma; \Gamma \vdash e : T$ where $\Sigma = T_1, \ldots, T_n$ is a list of types, $\Gamma : \mathit{Var} \rightharpoonup \mathcal{T}$ is a type context that maps variables to types, e is an expression, and T is a (simple) type. The intuitive meaning (which is formalized in the TR) is as follows. Given an evaluation environment that matches the type context Γ and an argument stack that matches the type stack Σ then e evaluates to a value of type T (or does not terminate).

Even though function types can have multiple forms, a well-typed expression often has a unique type (in a given type context). This type is derived from the way a function is used. For instance, we have $\lambda f.\lambda x.\lambda y.f\,x\,y : ([T_1, T_2] \to T) \to T_1 \to T_2 \to T$ and $\lambda f.\lambda x.\lambda y.(f\,x)\,y : (T_1 \to T_2 \to T) \to T_1 \to T_2 \to T$, and the types of the higher-order arguments are both unique.

A type T of an expression e has a unique type derivation that produces a type judgement $\cdot, \Gamma \vdash e : T$ with an empty type stack. We call this a *closed type judgement* for e. If T is a function type $\Sigma \to T'$ then there is a second type derivation for e that we call an *open type derivation*. It derives the *open type judgement* $\Sigma; \Gamma \vdash e : T'$ where $|\Sigma| > 0$.

Open and closed type judgements are *not* interchangeable. An open type judgement $\Sigma; \Gamma \vdash e : T$ can only appear in a derivation with an open root of the form $\Sigma', \Sigma; \Gamma \vdash e : T$, or in a subtree of a derivation whose root is a closed judgement of the form $\cdot; \Gamma \vdash e : \Sigma'', \Sigma \to T$ where $|\Sigma''| > 0$. In other words, in an open derivation $\Sigma; \Gamma \vdash e : T$, the expression e is a function that has to be applied to $n > |\Sigma|$ arguments at the same time. In a given type context and for a fixed function type, a well-typed expression has at most one open type derivation.

Type Rules. Figure 3 presents some type rules that modify the type stack. There is a close correspondence between the evaluation rules and the type rules in the sense that every evaluation rule corresponds to exactly one type rule. For every leaf rule that can return a function type (e.g., T:App), we add a second rule that derives the equivalent open type (e.g., T:AppPush). The rules that directly control the shape of the function types are T:AbsPush and T:AbsPop for lambda abstraction. While the other rules are (deterministically) syntax driven, the rules for lambda abstraction introduce a choice that shapes function types.

All type rules, lemmas, and additional details about the stack-based type system can be found the TR [33].

5. Multivariate Resource Polynomials

We now define the set of resource polynomials which is the search space of our automatic resource bound analysis. A resource polynomial $p : [\![T]\!] \to \mathbb{Q}_0^+$ maps a semantic value of some simple type T to a non-negative rational number. The definition of the set of resource polynomials for each type T is one of our main contributions. The right choice of resource polynomials is the basis for obtaining an analysis system that is simple, expressive, and allows for type inference via LP solving.

A key insight that we use is that the static type information we have gives us important hints on how values of a given type can be used in a program. An analysis of typical polynomial computations operating on a list $[a_1, \dots, a_n]$ shows that they often consist of operations that are executed for every k-tuple $(a_{i_1}, \dots, a_{i_k})$ with $1 \le i_1 < \cdots < i_k \le n$. Simple examples are linear map operations that perform some operation for every a_i or sorting algorithms that perform comparisons for every pair (a_i, a_j) with $1 \le i < j \le n$ in the worst case.

In this article, we generalize this observation to user-defined tree-like data structures. In lists of different node types with constructors C_1, C_2 and C_3, a linear computation is for instance often carried out for all C_1-nodes, all C_2-nodes, or all C_1 and C_3 nodes. In general, a typical polynomial computation is carried out for all tuples $(a_1, \dots, a_k)$ such that a_i is a list element with constructor C_j for some j and a_i appears in the list before a_{i+1} for all i.

Semantics of Types and Well-Formed Environments. For each simple type T we inductively define a set $[\![T]\!]$ of values of type T. Inductive types are interpreted as trees, numbers as numbers, and other types as leaves (we are not interested in their domain).

$$\begin{aligned}
[\![X]\!] &= Loc \\
[\![\text{unit}]\!] &= \{()\} \\
[\![T \text{ ref}]\!] &= \{R(a) \mid a \in [\![T]\!]\} \\
[\![\Sigma \to T]\!] &= \{(\lambda x.e, V) \mid \exists\, \Gamma\,.\; H \vDash V{:}\Gamma \;\wedge\; \cdot;\Gamma \vdash \lambda x.e : \Sigma {\to} T\} \\
[\![T_1 * \cdots * T_n]\!] &= [\![T_1]\!] \times \cdots \times [\![T_n]\!] \\
[\![B]\!] &= Tr(B) \text{ if } B = \langle C_1{:}(T_1, n_1), \dots, C_n{:}(T_k, n_k)\rangle
\end{aligned}$$

Here, $\mathcal{T} = Tr(\langle C_1{:}(T_1, n_1), \dots, C_n{:}(T_k, n_k)\rangle)$ is the set of trees τ with node labels $C_1, \dots, C_k$ which are inductively defined as follows. If $i \in \{1, \dots, k\}$, $a_i \in [\![T_i]\!]$, and $\tau_j \in \mathcal{T}$ for all $1 \le j \le n_i$ then $C_i(a_i, \tau_1, \dots, \tau_{n_i}) \in \mathcal{T}$.

If H is a heap, ℓ is a location, A is a type, and $a \in [\![A]\!]$ then we write $H \vDash \ell \mapsto a{:}A$ to mean that ℓ defines the semantic value $a \in [\![A]\!]$ when pointers are followed in H in the obvious way. We write $H \vDash \ell{:}A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $H \vDash \ell \mapsto a{:}A$. We pointwise extend this to argument stacks and environments and write $H \vDash V : \Gamma$ and $H \vDash S : \Sigma$. The concepts are formally defined in the TR [33].

$$\frac{}{\lambda\, a.\, 1 \;\; \in P(T)} \qquad \frac{\forall i \,:\, p_i \in P(T_i)}{\lambda\, \vec{a}.\, \prod_{i=1,\dots,k} p_i(a_i) \;\; \in P(T_1 * \cdots * T_k)}$$

$$\frac{\begin{array}{c} B = \langle C_1 : (T_1, n_1), \dots, C_m : (T_m, n_m)\rangle \\ \overline{C} = [C_{j_1}, \dots, C_{j_k}] \quad \forall i \,:\, p_i \in P(T_{j_i}) \end{array}}{\lambda\, b.\, \sum_{\vec{a} \in \tau_B(\overline{C}, b)} \; \prod_{i=1,\dots,k} p_i(a_i) \;\; \in P(B)}$$

Figure 4. Defining the set $P(T)$ of base polynomials for T.

$$\frac{}{\star \;\; \in \mathcal{I}(T)} \qquad \frac{\forall j \,:\, I_j \in \mathcal{I}(T_j)}{(I_1, \dots, I_k) \;\; \in \mathcal{I}(T_1 * \cdots * T_k)}$$

$$\frac{B = \langle C_1 : (T_1, n_1), \dots, C_m : (T_m, n_m)\rangle \qquad \forall i \,:\, I_{j_i} \in \mathcal{I}(T_{j_i})}{[\langle I_1, C_{j_1}\rangle, \dots, \langle I_k, C_{j_k}\rangle] \;\; \in \mathcal{I}(B)}$$

Figure 5. Defining the set $\mathcal{I}(T)$ of indices for type T.

Base Polynomials and Indices. The two key notions of this section are base polynomials and indices. They formalize the intuition that has been described at the beginning of this section. While the definitions are succinct and simple, they define a very rich structure that has important closure properties. First, base polynomials are non-negative. Second, they are closed under the discrete difference operators Δ_i where $\Delta_i\, p$ is defined through $\Delta_i\, p(x_1, \dots, x_n) = p(x_1, \dots, x_i + 1, \dots, x_n) - p(x_1, \dots, x_n)$.

In Figure 4, we define for each simple type T a set $P(T)$ of functions $p : [\![T]\!] \to \mathbb{N}$ that map values of type T to natural numbers. The *resource polynomials* for type T are then given as non-negative rational linear combinations of these *base polynomials*.

Let $B = \langle C_1 : (T_1, n_1), \dots, C_m : (T_m, n_m)\rangle$ be an inductive type. Let $\overline{C} = [C_{j_1}, \dots, C_{j_k}]$ and $b \in [\![B]\!]$. We define a set $\tau_B(\overline{C}, b)$ of k-tuples as follow: $\tau_B(\overline{C}, b)$ is the set of k-tuples $(a_1, \dots, a_k)$ such that $C_{j_1}(a_1, \vec{b}_1), \dots, C_{j_k}(a_k, \vec{b}_k)$ are nodes in the tree $b \in [\![B]\!]$ and $C_{j_1}(a_1, \vec{b}_1) <_{\text{pre}} \cdots <_{\text{pre}} C_{j_k}(a_k, \vec{b}_k)$ for the pre-order $<_{\text{pre}}$ on b. Like in the lambda calculus, we use the notation $\lambda\, a.\, e(a)$ for the anonymous function that maps an argument a to the natural number that is defined by the expression $e(a)$. Every set $P(T)$ contains the constant function $\lambda\, a.\, 1$. In the case of an inductive data type B this constant function arises also for $\overline{C} = []$ (one element sum, empty product).

To refer to the base polynomials in a systematic way we use indices to enumerate them. In Figure 5, we inductively define for each simple type T a set of *indices* $\mathcal{I}(T)$. For tuple types $T_1 * \cdots * T_k$ we identify the index $\star$ with the index $(\star, \dots, \star)$. Similarly, we identify the index $\star$ with the index $[]$ for inductive types.

Let T be a base type. For each index $i \in \mathcal{I}(T)$, we define a base polynomial $p_i : [\![T]\!] \to \mathbb{N}$ as follows.

$$\begin{aligned}
p_\star(a) &= 1 \\
p_{(I_1, \dots, I_k)}(a_1, \dots, a_k) &= \prod_{j=1,\dots,k} p_{I_j}(a_j) \\
p_{[\langle I_1, C_1\rangle, \dots, \langle I_k, C_k\rangle]}(b) &= \sum_{\vec{a} \in \tau_B([C_1, \dots, C_k], b)} \; \prod_{j=1,\dots,k} p_{I_j}(a_j)
\end{aligned}$$

Examples. To illustrate the definitions, we construct the set of base polynomials for different data types.

(a) We first consider the inductive type singleton that has only one constructor without arguments.

$$\text{singleton} = \mu X \,\langle\, \mathsf{Nil} \,:\, \text{unit} \rangle$$

Then we have

$$[\![\text{singleton}]\!] = \{\,\mathsf{Nil}\,(())\} \;\text{ and }\; P(\text{singleton}) = \{\lambda\, a.\, 1, \lambda\, a.\, 0\}\,.$$

To see why, consider the set $\mathcal{T}(\overline{C}) = \tau_{\text{singleton}}(\overline{C},\, \mathsf{Nil}\,(()))$ for different list of tuples of constructors $\overline{C}$. If $|\overline{C}| > 1$ then $\mathcal{T}(\overline{C}) = \emptyset$ because the tree $\mathsf{Nil}\,(())$ does not contain any tuples of size 2. Thus we have $p_{[\langle I_1,C_1\rangle,\ldots,\langle I_k,C_k\rangle]}(\,\mathsf{Nil}\,(())) = 0$ in this case (empty sum). The only remaining constructor lists $\overline{C}$ are $[]$ and $[\,\mathsf{Nil}\,]$. As always $p_{[]}(\,\mathsf{Nil}\,(())) = 1$ (singleton sum). Furthermore $p_{[\langle\star,\,\mathsf{Nil}\,\rangle]}(\,\mathsf{Nil}\,(())) = 1$ because $\tau_{\text{singleton}}([\,\mathsf{Nil}\,],\, \mathsf{Nil}\,(())) = \{\,\mathsf{Nil}\,(())\}$ and $P(\text{unit}) = \{\lambda\, a.\, 1\}$.

(b) Let us now consider the usual sum type

$$\text{sum}(T_1, T_2) = \mu X\,\langle\,\mathsf{Left}\,: T_1,\, \mathsf{Right}\,: T_2\rangle\;\;.$$

Then $[\![\text{sum}(T_1,T_2)]\!] = \{\,\mathsf{Left}\,(a) \mid a \in [\![T_1]\!]\} \cup \{\,\mathsf{Right}\,(b) \mid b \in [\![T_2]\!]\}$. If we define

$$\sigma_C(p)(C'(a)) \begin{cases} p(a) & \text{if } C = C' \\ 0 & \text{otherwise} \end{cases}$$

then $P(\text{sum}(T_1,T_2)) = \{\lambda x.1, \lambda x.0\} \cup \{\sigma_{\,\mathsf{Left}}\,(p) \mid p \in P(T_1)\} \cup \{\sigma_{\,\mathsf{Right}}\,(p) \mid p \in P(T_2)\}$.

(c) The next example is the list type

$$\text{list}(T) = \mu X\,\langle\,\mathsf{Cons}\,: T{*}X,\, \mathsf{Nil}\,: \text{unit}\rangle\;\;.$$

Then $[\![\text{list}(T)]\!] = \{\,\mathsf{Nil}\,(()),\, \mathsf{Cons}\,(a_1,\, \mathsf{Nil}\,(())), \ldots\}$ and we write

$$[\![\text{list}(T)]\!] = \{[], [a_1], [a_1,a_2], \ldots \mid a_i \in [\![T]\!]\}\,.$$

We then have $\tau_{\text{list}}([\,\mathsf{Cons}\,], [a_1,\ldots,a_n]) = \{a_1,\ldots,a_n\}$ and $\tau_{\text{list}}([\,\mathsf{Cons}\,,\, \mathsf{Cons}\,], [a_1,\ldots,a_n]) = \{(a_i,a_j) \mid 1 \le i < j \le n\}$. Let now $\overline{C} = [\,\mathsf{Cons}\,,\ldots,\, \mathsf{Cons}\,]$ or $\overline{C} = [\,\mathsf{Cons}\,,\ldots,\, \mathsf{Cons}\,,\, \mathsf{Nil}\,]$ for lists of length k and $k+1$, respectively. Then we have $\tau_{\text{list}}(\overline{C}, [a_1,\ldots,a_n]) = \{(a_{i_1},\ldots,a_{i_k}) \mid 1 \le i_1 < \cdots < i_k \le n\}$. On the other hand, $\tau_{\text{list}}(\overline{D}, [a_1,\ldots,a_n]) = \emptyset$ if $\overline{D} = \,\mathsf{Nil}\,{::}\overline{D'}$ for some $\overline{D'} \neq []$. Since $\sum_{\vec{a}\in\tau_{\text{list}}(\overline{C},[a_1,\ldots,a_n])} 1 = \binom{n}{k}$ and $\lambda\, a.\, 1 \in P(T)$, we have $\{\lambda\, b.\, \binom{|b|}{n} \mid n \in \mathbb{N}\} \subseteq P(\text{list}(T))$.

(d) Consider a list type with two different Cons-nodes (as in the running example in Figure 1):

$$\text{list2}(T_1,T_2) = \mu X\,\langle\,\mathsf{C}_1\,: T_1{*}X,\, \mathsf{C}_2\,: T_2 * X,\, \mathsf{Nil}\,: \text{unit}\rangle$$

Then we write $[\![\text{list2}(T_1,T_2)]\!] = \{[], [a_1], [a_1,a_2], \ldots \mid a_i \in (\{C_1\} \times [\![T_1]\!]) \cup (\{C_2\} \times [\![T_2]\!])\}$. Let $b = [b_1,\ldots,b_n]$. We have for example $\tau_{\text{list2}}([\,\mathsf{C}_1\,], b) = \{b_1,\ldots,b_n \mid \forall i\, \exists a : b_i = (\,\mathsf{C}_1\,, a)\}$ and $\tau_{\text{list2}}([\,\mathsf{C}_1\,,\, \mathsf{C}_2\,], [b_1,\ldots,b_n]) = \{(b_i,b_j) \mid \forall i,j\, \exists a,a' : b_i = (\,\mathsf{C}_1\,, a) \wedge b_j = (\,\mathsf{C}_2\,, a') \wedge 1 \le i < j \le n\}$.

If $\overline{C} = [\,\mathsf{C}_1\,,\ldots,\, \mathsf{C}_1\,]$ and $|\overline{C}| = k$ then $\sum_{\vec{a}\in\tau_{\text{list2}}(\overline{C},b)} 1 = \binom{|b|_{\,\mathsf{C}_1}}{k}$ where $|b|_{\,\mathsf{C}_1}$ denotes the number of C_1-nodes in the list b. Therefore we have $\{\lambda\, b.\, \binom{|b|_{C_1}}{n} \mid n \in \mathbb{N}\} \subseteq P(\text{list2}(T))$ and $\{\lambda\, b.\, \binom{|b|_{C_2}}{n} \mid n \in \mathbb{N}\} \subseteq P(\text{list2}(T))$. Now consider the set $\mathcal{D}$ of constructor lists $\overline{D}$ such that D contains exactly k_1 constructors C_1 and k_2 constructors C_2. If $S = \bigcup_{\overline{D}\in\mathcal{D}} \tau_{\text{list2}}(\overline{D}, b)$ then $\sum_{\vec{a}\in S} 1 = \binom{|b|_{\,\mathsf{C}_1}}{k_1}\binom{|b|_{\,\mathsf{C}_2}}{k_2}$. This means that such products of binomial coefficients are sums of base polynomials.

(e) Coinductive types like $\text{stream}(T) = \mu X\,\langle\,\mathsf{St}\,: T{*}X\rangle$ are not inhabited in our language since we interpret them inductively. A data structure of such a type cannot be created since we allow recursive definitions only for functions.

Spurious Indices. The previous examples illustrate that for some inductive data structures, different indices encode the same resource polynomial. For example, for the type $\text{list}(T)$ we have $p_{[\langle\star,\,\mathsf{Nil}\,\rangle]}(a) = p_{[]}(a) = 1$ for all lists a. Additionally, some indices encode a polynomial that is constantly zero. For the type $\text{list}(T)$ this is for example the case for $p_{\langle\star,\,\mathsf{Nil}\,\rangle\,::\,\overline{C}}$ if $|\overline{C}| > 0$. We call such indices *spurious*.

In practice, it is not beneficial to have spurious indices in the index sets since they slow down the analysis without being useful components of bounds. It is straightforward to identify spurious indices from the data type definition. The index $[\langle I_1, C_1\rangle, \ldots, \langle I_k, C_k\rangle]$ is for example spurious if $k > 1$ and the branching number of C_i is 0 for an $i \in \{1,\ldots,k-1\}$.

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

Example. Consider again our running example from Figure 1. For the function *abmap*, we derived the evaluation-step bound $3 + 12L + 12N$. It corresponds to the following resource polynomial. $12p_{(\star,\star,[\langle\star,\,\mathsf{Acons}\,\rangle])} + 12p_{(\star,\star,[\langle\star,\,\mathsf{Bcons}\,\rangle])} + 3p_{(\star,\star,[])}$. For the function *asort'*, we derived the evaluation-step bound $13 + 22KN + 13K^2N + 15N$, which corresponds to the resource polynomial $26p_{(\star,[\langle[\langle\star,::\rangle,\langle\star,::\rangle],\,\mathsf{Acons}\,\rangle])} + 35p_{(\star,[\langle[\langle\star,::\rangle],\,\mathsf{Acons}\,\rangle])} + 15p_{(\star,[\langle[],\,\mathsf{Acons}\,\rangle])} + 13p_{(\star,[])}$.

Selecting a Finite Index Set. Every resource polynomial is defined by a finite number of base polynomials. In an implementation, we also have to fix a finite set of indices to make possible an effective analysis. The selection of the indices to track can be customized for each inductive data type and for every program. However, we currently allow the user only to select a maximal degree of the bounds.

6. Resource-Aware Type System

In the following, we give an overview of the resource-aware type system. All type rules—with detailed descriptions, details of the soundness proof, and additional explanations—have been included in the companion technical report [33].

Type Annotations. We use the indices and base polynomials to define type annotations and resource polynomials.

A *type annotation* for a simple type T is a family

$$Q_T = (q_I)_{I\in\mathcal{I}(T)} \text{ with } q_I \in \mathbb{Q}_0^+$$

We write $\mathcal{Q}(T)$ for the set of type annotations for type T.

An *annotated type* is a pair (A, Q) where Q is a type annotation for the simple type $|A|$ where A and $|A|$ are defined as follows.

$$\begin{aligned} A ::= X \mid A \text{ ref} \mid A_1 * \cdots * A_n \mid \langle [A_1,\ldots,A_n] \to B, \Theta\rangle \\ \mid \text{unit} \mid \mu X.\, \langle C_1 : A_1{*}X^{n_1}, \ldots, C_k : A_k{*}X^{n_k}\rangle \end{aligned}$$

$|A|$ is the simple type that can be obtained from A by removing all type annotations from function types. A function type $\langle [A_1,\ldots,A_n] \to B, \Theta\rangle$ is annotated with a set

$$\Theta \subseteq \{(Q_A, Q_B) \mid Q_A \in \mathcal{Q}(|A_1 * \cdots * A_n|) \wedge Q_B \in \mathcal{Q}(|B|)\}\,.$$

The set Θ can contain multiple valid resource annotations for arguments and the result of the function.

Potential of Annotated Types and Contexts. Let (A, Q) be an annotated type. Let H be a heap and let v be a value with $H \vDash \ell \mapsto a : |A|$. Then the annotation Q defines the *potential*

$$\Phi_H(v : (A,Q)) = \sum_{I\in\mathcal{I}(T)} q_I \cdot p_I(a)$$

where only finitely many q_I are non-zero. For use in the type system we need to extend the definition of resource polynomials to type contexts and stacks. We treat them like tuple types. Let

$\Gamma = x_1{:}A_1, \dots, x_n{:}A_n$ be a type context and let $\Sigma = B_1, \dots, B_m$ be a list of types. The index set $\mathcal{I}(\Sigma;\Gamma)$ is defined through

$$\mathcal{I}(\Sigma;\Gamma) = \{(I_1, \dots, I_m, J_1, \dots, J_n) \mid I_j {\in} \mathcal{I}(|B_j|), J_i {\in} \mathcal{I}(|A_i|\} \ .$$

A *type annotation* Q for $\Sigma;\Gamma$ is a family $Q = (q_I)_{I \in \mathcal{I}(\Sigma;\Gamma)}$ with $q_I \in \mathbb{Q}_0^+$. We denote a *resource-annotated context* with $\Sigma;\Gamma;Q$. Let H be a heap and V be an environment with $H \vDash V : \Gamma$ where $H \vDash V(x_j) \mapsto a_{x_j} : |\Gamma(x_j)|$. Let furthermore $S = \ell_1, \dots, \ell_m$ be an argument stack with $H \vDash S : \Sigma$ where $H \vDash \ell_i \mapsto b_i : |B_i|$ for all i. The potential of $\Sigma;\Gamma;Q$ with respect to H and V is

$$\Phi_{S,V,H}(\Sigma;\Gamma;Q) = \sum_{\vec{I} \in \mathcal{I}(\Sigma;\Gamma)} q_{\vec{I}} \prod_{j=1}^{m} p_{I_j}(b_j) \prod_{j=m+1}^{m+n} p_{I_j}(a_{x_j})$$

Here, $\vec{I} = (I_1, \cdots, I_{m+n})$. In particular, if $\Sigma = \Gamma = \cdot$ then $\mathcal{I}(\Sigma;\Gamma) = \{()\}$ and $\Phi_{V,H}(\Sigma;\Gamma;q_{()}) = q_{()}$.

Folding of Potential Annotations. A key notion in the type system is the *folding* for potential annotations that is used to assign potential to typing contexts that result from a pattern match (unfolding) or from the application of a constructor of an inductive data type (folding). Folding of potential annotations is conceptually similar to folding and unfolding of inductive data types in type theory.

Let $B = \mu X. \langle \dots, C : A{*}X^n, \dots \rangle$ be an inductive data type. Let Σ be a type stack, $\Gamma, b{:}B$ be a context and let $Q = (q_I)_{I \in \mathcal{I}(\Sigma;\Gamma,y:B)}$ be a context annotation. The C*-unfolding* $\lhd_B^C(Q)$ of Q with respect to B is an annotation $\lhd_B^C(Q) = (q'_I)_{I \in \mathcal{I}(\Sigma;\Gamma')}$ for a context $\Gamma' = \Gamma, x{:}A{*}B^n$ that is defined by

$$q'_{(I,(J,L_1,\dots,L_n))} = \begin{cases} q_{(I,\langle J,C\rangle :: L_1 \cdots L_n)} + q_{(I,L_1 \cdots L_n)} & J = 0 \\ q_{(I,\langle J,C\rangle :: L_1 \cdots L_n)} & J \neq 0 \end{cases}$$

Here, $L_1 \cdots L_n$ is the concatenation of the lists $L_1, \dots, L_n$.

The C-unfolding $\lhd_B^C(Q)$ of Q is used in construction and destruction of inductive data types to pass the potential from the context to the resulting data structures without loss. It can be encoded with simple linear constraints.

Lemma 1. *Let $B = \mu X. \langle \dots, C : A{*}X^n, \dots \rangle$ be an inductive data type. Let $\Sigma;\Gamma, x{:}B;Q$ be an annotated context, $H \vDash V : \Gamma, x{:}B$, $H \vDash S : \Sigma$, $H(V(x)) = (C, \ell)$, and $V' = V[y \mapsto \ell]$. Then $H \vDash V' : \Gamma, y{:}A{*}B^n$ and $\Phi_{S,V,H}(\Sigma;\Gamma, x{:}B;Q) = \Phi_{S,V',H}(\Sigma;\Gamma, y{:}A{*}B^n; \lhd_B^C(Q))$.*

Sharing. Let $\Sigma;\Gamma, x_1{:}A, x_2{:}A;Q$ be an annotated context. The *sharing operation* $\curlyvee Q$ defines an annotation for a context of the form $\Sigma;\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable. Lemma 2 shows that sharing is a linear operation that does not lead to any loss of potential.

Lemma 2. *Let A be a data type. Then there are natural numbers $c_k^{(i,j)}$ for $i, j, k \in \mathcal{I}(|A|)$ such that the following holds. For every context $\Sigma;\Gamma, x_1{:}A, x_2{:}A;Q$ and every H, V with $H \vDash V : \Gamma, x{:}A$ and $H \vDash S : \Sigma$ it holds that $\Phi_{S,V,H}(\Sigma, \Gamma, x{:}A;Q') = \Phi_{S,V',H}(\Sigma;\Gamma, x_1{:}A, x_2{:}A;Q)$ where $V' = V[x_1, x_2 \mapsto V(x)]$ and $q'_{(\ell,k)} = \sum_{i,j \in \mathcal{I}(A)} c_k^{(i,j)} q_{(\ell,i,j)}$.*

The coefficients $c_k^{(i,j)}$ can be computed effectively. We were however not able to derive a closed formula for the coefficients. The proof is similar as in previous work [32]. For a context $\Sigma;\Gamma, x_1{:}A, x_2{:}A;Q$ we define $\curlyvee Q$ to be Q' from Lemma 2.

Type Judgements. A resource-aware type judgement has the form

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

where $\Sigma;\Gamma;Q$ is an annotated context, M is a resource metric, A is an annotated type, and Q' is a type annotation for $|A|$. The intended meaning of this judgment is that if there are more than $\Phi(\Sigma;\Gamma;Q)$ resource units available then this is sufficient to cover the evaluation cost of e under metric M. In addition, there are at least $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

Notations. For $K \in \mathbb{Q}$ we write $Q = Q' + K$ to state that $q_\star = q'_\star + K \geq 0$ and $q_I = q'_I$ for $I \neq \star \in \mathcal{I}$. Let $\Gamma = \Gamma_1, \Gamma_2$ be a context, let $I = (I_1, \dots, I_k) \in \mathcal{I}(\Gamma_1)$ and $J = (J_1, \dots, J_\ell) \in \mathcal{I}(\Gamma_2)$. We write (I, J) for the index $(I_1, \dots, I_k, J_1, \dots, J_\ell) \in \mathcal{I}(\Gamma)$. Let Q be an annotation for a context $\Sigma;\Gamma_1, \Gamma_2$. For $J \in \mathcal{I}(\Gamma_2)$ we define the *projection* $\pi_{(J,J')}^{\Gamma_1}(Q)$ of Q to Γ_1 to be the annotation Q' for $\cdot;\Gamma_1$ with $q'_I = q_{(J,I,J')}$. In the same way, we define the annotations $\pi_J^{\Sigma}(Q)$ for $\Sigma;\cdot$ and $\pi_J^{\Sigma;\Gamma_1}(Q)$ for $\Sigma;\Gamma_1$.

Cost Free Types. We write $\Sigma;\Gamma;Q \ {}_{\mathsf{cf}}{\vdash}\ e : (A, Q')$ to refer to cost-free type judgments where cf is the cost-free metric with $\mathsf{cf}(K) = 0$ for constants K. We use it to assign potential to an extended context in the let rule. More info is available in previous work [30].

Subtyping. As usual, subtyping is defined inductively so that types have to be structurally identical. The most interesting rule is the one for function types:

$$\frac{\Theta' \subseteq \Theta \qquad \forall i : A'_i <: A_i \qquad B <: B'}{\langle [A_1, \dots, A_n] \to B, \Theta \rangle <: \langle [A'_1, \dots, A'_n] \to B', \Theta' \rangle} \ (\text{S:F}\textsc{un})$$

A function type is a subtype of another function type if it allows more resource behaviors ($\Theta' \subseteq \Theta$). Result types are treated covariant and arguments are treated contravariant.

Type Rules. Figure 6 shows selected type rules for annotated types. All rules can be found in the TR.

The rule A:V<span style="font-variant:small-caps">ar</span> can only be applied if the type stack Σ is empty. It then simply accounts for the cost M^{var} and passes the potential that is assigned to the variable by the type context to the result type. If the type stack is not empty then the rule A:VARPUSH has to be applied. In this case, the variable x must have a function type. We then look up a possible type annotation for the arguments and the result $(P, P') \in \Theta$ in the type context, account for the cost of variable look-up (M^{var}) and behave as specified by (P, P'). We do not account for the cost of the "function application" because is cost is handled in the rules A:APP and A:APPPUSH.

The rules A:APP and A:APPPUSH correspond to the simple type rules T:APP and T:APPPUSH. In A:APP we assume that the type stack is empty. We account for the cost $M_n^{\mathsf{app}}$of applying a function to n arguments and look up valid potential annotations (P, P') for the function body in the function annotation Θ. We then require that we have the potential specified by P available and return potential as specified by P'. In the rule A:APPPUSH we account for two applications: We first account for the function application as in the rule A:APP. We then assume that the return type is a function type and apply the arguments that are stored on the type stack Σ as we do in the rule A:VARPUSH.

The rules A:ABSPUSH and A:ABSPOP for lambda abstraction correspond the rules T:ABSPUSH and T:ABSPOP. As in the simple type system we can use them to non-deterministically pop the type stack Σ. When we do so in the rule A:ABSPOP, we create the function annotation Θ by essentially deriving $\Sigma;\Gamma;P \ {}_{M}{\vdash}\ \lambda x.e : (B, P')$ for every $(P, P') \in \Theta$. However, we throw away all potential that depends on the context Γ and only use the potential that is assigned the arguments Σ (annotation R).

The rule A:CONS assigns potential to a new node of an inductive data structure. The The C-unfolding $\lhd_B^C(Q')$ transforms the annotation Q' to an annotation Q for the context $\cdot; x{:}A{*}B^n$. In A:MAT, the initial potential defined by the annotation Q of the context $\Sigma;\Gamma, x{:}B$ has to be sufficient to pay the costs of the evaluation of e_1 or e_2 and the potential defined by the annotation Q' of the result type. To type the expression e_2 we basically just use the annotation Q. To type the expression e_1, we rely on the C-unfolding $\lhd_B^C(Q)$ that results in an annotation for the context $\Sigma;\Gamma, y{:}A{*}B^n$. In both rules, there is no loss of potential (see Lemma 1).

$$\frac{Q = Q' + M^{\mathsf{var}}}{\cdot; x{:}B; Q \;{}_M\!\vdash x : (B, Q')}\ (\text{A:V}\textsc{ar})$$

$$\frac{\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n \qquad (P, P') \in \Theta \qquad \pi^{\Gamma}_{\star}(Q) = P + M^{\mathsf{app}}_n \qquad Q' = P'}{\cdot; x : \langle [A_1, \ldots, A_n] \to B, \Theta \rangle, \Gamma; Q \;{}_M\!\vdash x\, x_1 \cdots x_n : (B, Q')}\ (\text{A:A}\textsc{pp})$$

$$\frac{\begin{gathered}(P, P') \in \Theta \qquad P' = Q' \\ \pi^{\Sigma;\cdot}_{\star}(Q) = P + M^{\mathsf{var}}\end{gathered}}{\Sigma; x : \langle \Sigma \to B, \Theta \rangle; Q \;{}_M\!\vdash x : (B, Q')}\ (\text{A:V}\textsc{ar}\text{P}\textsc{ush})$$

$$\frac{\begin{gathered}\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n \qquad (P, P') \in \Theta \qquad (R, R') \in \Theta' \\ \pi^{\Gamma}_{\star}(Q) = P + M^{\mathsf{app}}_n \qquad \pi^{\Sigma}_{\star}(Q) - q_\star + p'_\star = R \qquad R' = Q'\end{gathered}}{\Sigma; x : \left\langle [A_1, \ldots, A_n] \to \left\langle \Sigma \to B, \Theta' \right\rangle, \Theta \right\rangle, \Gamma; Q \;{}_M\!\vdash x\, x_1 \cdots x_n : (B, Q')}\ (\text{A:A}\textsc{pp}\text{P}\textsc{ush})$$

$$\frac{\begin{gathered}\Sigma; \Gamma, x{:}A; P \;{}_M\!\vdash e : (B, Q') \\ Q = R + M^{\mathsf{bind}} \qquad \forall\, I, \vec{J} : r_{(I, \vec{J})} = p_{(\vec{J}, I)}\end{gathered}}{A{::}\Sigma; \Gamma; Q \;{}_M\!\vdash \lambda x.e : (B, Q')}\ (\text{A:A}\textsc{bs}\text{P}\textsc{ush})$$

$$\frac{\begin{gathered}Q = Q' + M^{\mathsf{abs}} \\ \forall (P, P') \in \Theta :\ \Sigma; \Gamma; R \;{}_M\!\vdash \lambda x.e : (B, P') \wedge r_{(\vec{I}, \vec{J})} = \begin{cases} p_{\vec{I}} & \text{if } \vec{J} = \vec{\star} \\ 0 & \text{otherwise}\end{cases}\end{gathered}}{\cdot; \Gamma; Q \;{}_M\!\vdash \lambda x.e : (\langle \Sigma \to B, \Theta \rangle, Q')}\ (\text{A:A}\textsc{bs}\text{P}\textsc{op})$$

$$\frac{\begin{gathered}B = \mu X. \langle \ldots C : A {*} X^n \ldots \rangle \\ Q = \lhd^{C}_{B}(Q') + M^{\mathsf{cons}}\end{gathered}}{\cdot; x{:}A {*} B^n; Q \;{}_M\!\vdash C\, x : (B, Q')}\ (\text{A:C}\textsc{ons})$$

$$\frac{\begin{gathered}B = \mu X. \langle \ldots C : A {*} X^n \ldots \rangle \qquad \Sigma; \Gamma, y{:}A {*} B^n; P \;{}_M\!\vdash e_1 : (A', P') \qquad P' = Q' \\ \Sigma; \Gamma, x{:}B; R \;{}_M\!\vdash e_2 : (A', R') \qquad \lhd^{C}_{B}(Q) = P + M^{\mathsf{mat}}_1 \qquad Q = R + M^{\mathsf{mat}}_2 \qquad R' = Q'\end{gathered}}{\Sigma; \Gamma, x{:}B; Q \;{}_M\!\vdash \mathsf{match}\ x\ \mathsf{with}\ C\, y \to e_1 \mid e_2 : (A', Q')}\ (\text{A:M}\textsc{at})$$

$$\frac{\begin{gathered}Q = M^{\mathsf{share}} + \curlyvee(P) \\ \Sigma; \Gamma, x_1{:}A, x_2{:}A; P \;{}_M\!\vdash e : (B, Q')\end{gathered}}{\Sigma; \Gamma, x{:}A; Q \;{}_M\!\vdash \mathsf{share}\ x\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e : (B, Q')}\ (\text{A:S}\textsc{hare})$$

$$\frac{\begin{gathered}\Sigma; \Gamma_2, \Gamma_1; P \;{}_M\!\vdash e_1 \rightsquigarrow \Sigma; \Gamma_2, x{:}A; P' \qquad Q = P + M^{\mathsf{let}}_1 \\ \Sigma; \Gamma_2, x{:}A; R \;{}_M\!\vdash e_2 : (B, Q') \qquad P' = R + M^{\mathsf{let}}_2\end{gathered}}{\Sigma; \Gamma_2, \Gamma_1; Q \;{}_M\!\vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (B, Q')}\ (\text{A:L}\textsc{et})$$

$$\frac{\begin{gathered}F \triangleq f_1 = \lambda x_1.e_1\ \mathsf{and} \cdots \mathsf{and}\ f_n = \lambda x_n.e_n \qquad \Delta = f_1{:}A_1, \ldots, f_n{:}A_n \\ \forall i\ : \cdot; \Gamma_i, \Delta; P_i \;{}_M\!\vdash \lambda x_i.e_i : (A_i, P'_i) \qquad \pi^{\Sigma;\Gamma_0}_{\vec{\star}}(Q) = \pi^{\Sigma;\Gamma_0}_{\vec{\star}}(P) + M^{\mathsf{rec}} + n{\cdot}M^{\mathsf{abs}} \qquad \Sigma; \Gamma_0, \Delta; P \;{}_M\!\vdash e : (B, Q')\end{gathered}}{\Sigma; \Gamma_0, \ldots, \Gamma_n; Q \;{}_M\!\vdash \mathsf{let\ \ rec}\ F\ \mathsf{in}\ e : (B, Q')}\ (\text{A:L}\textsc{et}\text{R}\textsc{ec})$$

$$\diamond \quad \diamond \quad \diamond$$

$$\frac{\forall j \in \mathcal{I}(\Sigma; \Delta): \quad j{=}\vec{\star} \implies \cdot; \Gamma; \pi^{\Gamma}_{j}(Q) \;{}_M\!\vdash e : (A, \pi^{x:A}_{j}(Q')) \qquad j{\neq}\vec{\star} \implies \cdot; \Gamma; \pi^{\Gamma}_{j}(Q) \;{}_{\mathsf{cf}}\!\vdash e : (A, \pi^{x:A}_{j}(Q'))}{\Sigma; \Delta, \Gamma; Q \;{}_M\!\vdash e \rightsquigarrow \Sigma; \Delta, x{:}A; Q'}\ (\text{B:B}\textsc{ind})$$

Figure 6. Selected type rules of the resource-aware type system.

The rule A:SHARE uses the sharing operation $\curlyvee P$ to related the potentials defined by $\Sigma; \Gamma, x{:}A; Q$ and $\Sigma; \Gamma, x_1{:}A, x_2{:}A; P$. As with matching, there is no loss of potential (see Lemma 2).

In the rule A:LET the result of the evaluation of an expression e_1 is bound to a variable x. The problem that arises is that the resulting annotated context $\Sigma; \Gamma_2, x{:}A; R$ features potential functions whose domain consists of data that is referenced by x as well as data that is referenced in the type context Γ_2. This potential has to be related to data that is referenced by Γ_2 and the free variables in e_1 (i.e., the variables in the type context Γ_1).

To express the relations between mixed potentials before and after the evaluation of e_1, we introduce an auxiliary binding judgement of the from

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

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

Soundness. Our goal is to prove the following soundness statement for type judgements. Intuitively, it says that the initial potential is an upper bound on the high-water mark of the resource usage, no matter how long we execute the program.

> If $\Sigma; \Gamma; Q \;{}_M\!\vdash e : (A, Q')$ and $S, V, H \;{}_M\!\vdash e \Downarrow w \mid (p, p')$ then $p \leq \Phi_{S,V,H}(\Sigma; \Gamma; Q)$.

To prove this statement by induction, we need to prove a stronger statement that takes into account the return value and the annotated type (A, Q') of e. Moreover, the previous statement is only true if the values in S, V and H respect the types required by Σ and Γ. In addition to the aforementioned soundness, Theorem 1 states a stronger property for terminating evaluations. If an expression e evaluates to a value v then the difference between initial and final potential is an upper bound on the resource usage of the evaluation.

Theorem 1 (Soundness)**.** *Let $H \models V : \Gamma$, $H \models S : \Sigma$, and $\Sigma; \Gamma; Q \;{}_M\!\vdash e : (B, Q')$.*

1. *If $S, V, H \;{}_M\!\vdash e \Downarrow (\ell, H') \mid (p, p')$ then $p \leq \Phi_{S,V,H}(\Sigma; \Gamma; Q)$, $p - p' \leq \Phi_{S,V,H}(\Sigma; \Gamma; Q) - \Phi_{H'}(\ell{:}(B, Q'))$, and $H \models \ell : B$.*
2. *If $S, V, H \;{}_M\!\vdash e \Downarrow \circ \mid (p, p')$ then $p \leq \Phi_{S,V,H}(\Sigma; \Gamma; Q)$.*

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

7. Implementation and Bound Inference

Figure 7 shows an overview of the implementation of RAML. It consists of about 12000 lines of OCaml code, excluding the parts that we reused from Inria's OCaml implementation. The development took around 8 person months. We found it very helpful to develop the implementation and the theory in parallel, and many theoretical ideas have been inspired by implementation challenges.

We reuse the parser and type inference algorithm from OCaml 4.01 to derive a typed OCaml syntax tree from the source program. We then analyze the function applications to introduce bracket function types. To this end, we copy a lambda abstraction for every call site. We still have to implement a unification algorithm since functions, such as *let g = f x*, that are defined by partial application may be used at different call sites. Moreover, we have to deal with functions that are stored in references.

In the next step, we convert the typed OCaml syntax tree into a typed RAML syntax tree. Furthermore, we transform the program into share-let-normal form without changing the resource

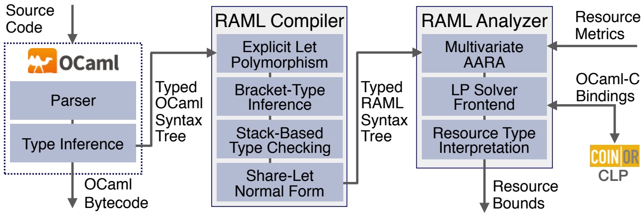

Figure 7. Implementation of RAML.

behavior. For this purpose, each syntactic form has a *free* flag that specifies whether it contributes to the cost of the original program. For example, all share expressions that are introduced in the transformation are *free*. We also insert eta expansions whenever they do not influence resource usage.

After this compilation phase, we perform the actual multivariate AARA on the program in share-let-normal form. Resource metrics can be easily specified by a user. We include a metric for heap cells, evaluation steps, and *ticks*. Ticks allows the user to flexibly specify the resource cost of programs by inserting tick commands *Raml.tick(q)* where *q* is a (possibly negative) floating-point number. In principle, the bound inference works similarly as in previous AARA systems [31, 34]: First, we fix a maximal degree of the bounds and annotate all types in the derivation of the simple types with variables that correspond to type annotations for resource polynomials of that degree. Second, we generate a set of linear inequalities, which express the relationships between the added annotation variables as specified by the type rules. Third, we solve the inequalities with Coin-Or's LP solver CLP. A solution of the linear program corresponds to a type derivation in which the variables in the type annotations are instantiated according to the solution. The objective function contains the coefficients of the resource annotation of the program inputs to minimize the initial potential. Modern LP solvers provide support for iterative solving that allows us to prioritize minimization of higher-degree annotations.

The type system we use in the implementation significantly differs from the declarative version we describe in this article. For one thing, we have to use algorithmic versions of the type rules in the inference in which the non-syntax-directed rules are integrated into the syntax-directed ones [32]. For another thing, we annotate function types not with a set of type annotations but with a function that returns an annotation for the result type if presented with an annotation of the argument type. These annotations are symbolic and the actual numbers are yet to be determined. So function annotations have the side effect of sending constraints to the LP solver.

To make the resource analysis more expressive, we also allow *resource-polymorphic recursion*. This means that the type annotation in the recursive call differs from the annotations in the argument and result types of the function. To infer such types we successively infer type annotations of higher degree [30, 32].

Most of the generated constraints have the form of a so-called network-flow problem [55]. LP solvers can handle network problems very efficiently and in practice CLP solves the constraints RAML generates in linear time.

Apart from the analysis itself, we also implemented the conversion of the derived resource polynomials into easily-understood polynomial bounds and a pretty printer for RAML types and expressions. Additionally, we implemented an efficient RAML interpreter that we use for debugging and to determine the quality of the bounds.

8. Experimental Evaluation

The development of RAML has been driven by an ongoing experimental evaluation with OCaml code. Our goal has been to ensure that the derived bounds are precise, that different programming styles are supported, that the analysis is efficient, and that existing code can be analyzed. During the evaluation, we applied our automatic resource bound analysis to 411 functions and 6018 lines of code. The experiments have been performed with a 2.6 GHz Intel Core i5 MacBook Pro with 16GB RAM. The source code of RAML as well as all OCaml files used in the experiments are available online [28]. The website also provides an easy-to-use web interface that can be used to experiment with RAML.

```
let comp f x g = fun z → f x (g z)

let rec walk f xs =
 match xs with | [] → (fun z →  z)
  | x::ys → match x with
      | Left _ →
        fun y → comp (walk f) ys (fun z → x::z) y
      | Right l →
        let x' = Right (quicksort f l) in
        fun y → comp (walk f) ys (fun z → x'::z) y

let rev_sort f l = walk f l []
```

RAML output for *rev_sort* (0.68*s* run time; steps metric):

```
10 + 23*K' + 32*L' + 20*L'*Y + 13*L'*Y^2
```

Figure 8. Modified challenge example from Avanzini et al. [10] and shortened output of the automatic bound analysis. We assume 0 cost of the higher-order argument *f*. The derived bound on the number of steps is tight. L' is the number of Right-nodes in the input list, K' is the number Left-nodes, and Y is the maximal length of the lists in the Right-nodes.

Analyzed Code and Limitations. The experiments have been performed with code from four sources: extracted OCaml code from Coq specifications in CompCert [47], code from an OCaml tutorial [49], the OCaml standard library, and code written by us. For the handwritten code, we mostly implemented classical textbook algorithms and use cases inspired from real-word applications. The textbook algorithms include algorithms for matrices, graph algorithms, search algorithms, and classic examples from amortized analysis such as functional queues and binary counters. The use cases include energy management in an autonomous mobile device and calling Amazon's Dynamo DB from OCaml (see Section 9).

OCaml is a complex programming language and RAML does not yet support all language features of OCaml. This includes modules, object-oriented features, record types, built-in equality, strings, nested patterns, and calls to native C functions. If RAML can be applied to existing code then the results are often satisfactory. For instance, we applied RAML to OCaml's standard list library *list.ml*: In 3.2 seconds RAML automatically derives evaluation-step bounds for 47 of the 51 top-level functions. All derived bounds are asymptotically tight. The 4 functions that cannot be bounded by RAML all rely on functions whose termination (and thus resource usage) depends on an arithmetic integer operations, which are currently unsupported. The file *list.ml* consists of 428 LOC.

RAML fails if the resource usage can only be bounded by a measure that depends on a semantic property of the program or a measure that depends on the difference of the sizes of two data structures. Loose bounds are often the result of inter-procedural dependencies. For instance, the worst-case behaviors of two functions f and g might be triggered by different inputs. However, the analysis would add up the worst-case behaviors of both functions in a program such as *f(a);g(a)*. Another reason for loose bounds is that a tight bound cannot be represented by our resource polynomials.

Example Experiment. To give an impression of the experiments we performed, Figure 8 contains the output of an analysis of a challenging function in RAML. The code is an adoption of an example that has been recently presented by Avanzini et al. [10] as a function that can not be handled by existing tools. To illustrate the challenges of resource analysis for higher-order programs, Avanzini et al. implemented a (somewhat contrived) reverse function *rev* for lists using higher-order functions. RAML automatically derives a tight linear bound on the number of steps used by *rev*.

To use more features of our analysis, we modified Avanzini et al.'s *rev* in Figure 8 by adding an additional argument *f* and a pattern match to the definition of the function *walk*. The resulting type of walk is $\alpha \to \alpha \to$ bool) $\to$ [$(\beta * \alpha$ list) either list; $(\beta * \alpha$ list) either list] $\to$ $(\beta{*}\alpha$ list) either list . Like before the modification, *walk* is essentially the *rev_append* function for lists. However, we assume that the input lists contain nodes of the form *Left a* or *Right b* so that *b* is a list. During the reverse process of the first list in the argument, we sort each list that is contained in a *Right*-node using the standard implementation of quick sort (not given here). RAML derives the tight evaluation-step bound that is shown in Figure 8. Since the comparison function for *quicksort* (argument *f*) is not available, RAML assumes that it does not consume any resources. If *rev_sort* is applied to a concrete argument *f* then the analysis is repeated to derive a bound for this instance.

CompCert Evaluation. We also performed an evaluation with the OCaml code that is created by Coq's code extraction mechanism during the compilation of the verified CompCert compilers [47]. We sorted the files topologically from their dependency requirements, and analyzed 13 files from the top. We could not process the files further down the dependency graph because they heavily relied on modules which we do not currently support. Using the evaluation-step metric, we analyzed 164 functions, 2740 LOC in 1300 seconds.

Figure 9 shows example functions from the CompCert code base. As an artifact from the Coq code extraction, CompCert uses two implementations of the reverse function for lists. The function *rev* is a naive quadratic implementation that uses *append* and the function *rev'* is an efficient tail-recursive linear implementation. RAML automatically derives precise evaluation step bounds for both functions. As a result, a Coq user who is inspecting the derived bounds for the extracted OCaml code is likely to spot performance problems resulting from the use of *rev*.

```
let rec app l m = match l with | [] → m
                  | a :: l1 → a :: (app l1 m)

let rec rev = function | [] → []
      | x :: l' → app (rev l') (x :: [])

let rec rev_append l l' = match l with | [] → l'
      | a :: l0 → rev_append l0 (a :: l')

let rev' l = rev_append l []
```

RAML output for *rev* ($0.1s$ run time; steps metric):

```
3 + 9.5*M + 4.5*M^2
```

RAML output for *rev'* ($0.05s$ run time; steps metric):

```
7 + 9*M
```

Figure 9. Two implementations of list reverse that are used in the CompCert C compiler [47]; one is linear and the other is quadratic.

Summary of Results. Table 1 contains a compilation of the experimental results. The first 3 rows show the results for OCaml libraries, handwritten code, and the OCaml tutorial [49]. The last row shows the results for CompCert [47]. The column *LOC* contains the total number of lines of OCaml code that has been analyzed with the respective metric. Similarly, the column *Time* contains the total time of all analyses with this metric. The column *#Poly* contains the number of functions for which RAML automatically derived a bound. The columns *#Const*, *#Lin*, *#Quad*, and *#Cubic* show the number of derived bounds that are constant, linear, quadratic, and cubic. Finally, columns *#Failed* and *Asym.Tight* contain the number of examples for which RAML is unable to derive a bound and the number of bounds that are asymptotically tight, respectively. We also experimented with example inputs to determine the precision of the constant factors in the bounds. In general, the bounds are very precise and often match the actual worse-case behavior.

The reported numbers result from the analysis of 411 non-trivial functions that are (with a few exceptions) recursive and higher order. The TR contains a short description of every function that is part of the evaluation, along with its type, the run time of the analysis, and the derived bounds. The functions have been automatically analyzed using the *steps* metric that counts the number of evaluation steps and the *heap* metric that counts the number of allocated heap cells. Moreover, we have used the *tick* metric to add custom cost measures to some of the functions. These measures vary from program to program and include number of function calls, energy consumption, and amount of data sent to the cloud. Details can be found in the source code [28].

There are two main reasons for the difference between the runtime of the analysis per function for the CompCert code ($7.9s$) and the other evaluated code ($0.29s$). First, the CompCert code contains more complex data structures and we thus track more coefficients. Second, there is a larger percentage of functions for which we cannot derive a bound (15.8% vs. 1.6%). As a result, RAML looks for bounds of higher degree before giving up. Both leads to a larger number of constraints to solve for the LP solver. Finally, there are a few outlier functions that cause an unusually long analysis time. This is possibly due to a performance bug.

In general, the analysis is efficient in practice. However, RAML is slowing down if the analyzed program contains many variables or functions with many arguments. Another source of complexity is the maximal degree of polynomials in the search space. Depending on the complexity of the program, the analysis becomes unusable when searching for bounds with maximal degree $7 - 9$. The efficiency could be improved by combining amortized resource analysis with data-flow analyses and heuristics that predict the parts of the input that cause higher-degree resource usage.

9. Case Study: Bounds for DynamoDB Queries

Having integrated the analysis with Inria's OCaml compiler enables us to analyze and compile real programs. An interesting use case of our resource bound analysis is to infer worst-case bounds on DynamoDB queries. DynamoDB is a commercial NoSQL cloud database service, which is part of Amazon Web Services (AWS). Amazon charges DynamoDB users on a combination of number of queries, transmitted fields, and throughput. Since DynamoDB is a NoSQL service, it is often only possible to retrieve the whole table—which can be expensive for large data sets—or single entries that are identified by a key value. The DynamoDB API is available through the Opam package *aws*. We make the API available to the analysis by using *tick* functions that specify resource usage. Since the query cost for different tables can be different, we provide one function per action and table.

```
let db_query student_id course_id =
  Raml.tick(1.0); Awslib.get_item ...
```

In the following, we describe the analysis of a specific OCaml application that uses a database that contains a large table that stores

	Metric	#Funs	LOC	Time	#Const	#Lin	#Quad	#Cubic	#Poly	#Failed	Asym.Tight
	steps	243	3218	72.10s	16	130	60	28	239	4	225
	heap	243	3218	70.36s	41	112	60	22	239	4	225
	tick	174	2144	64.68s	19	79	53	19	174	0	160
CompCert	steps	164	2740	1300.91s	32	99	7	0	138	26	137

Table 1. Overview of experimental results.

grades of students for different courses. Our first function computes the average grade of a student for a given list of courses.

```
let avge_grade student_id course_ids =
  let f acc cid =
    let (length,sum) = acc in
    let grade = match db_query student_id cid with
      | Some q → q
      | None → raise (Not_found (student_id,cid))
    in (length +. 1.0, sum +. grade)
  in
  let (length,sum) = foldl f (0.0,0.0) course_ids in
  sum /. length
```

In $0.03s$ RAML computes the tight bound $1 \cdot m$ where m is the length of the argument *course_ids*. We omit the standard definitions of functions like *foldl* and *map*. However, they are not built-in into our systems but the bounds are derived form first principles.

Next, we sort a given list of students based on the average grades in a given list of classes using quick sort. As a first approximation we use a comparison function that is based on *average_grade*.

```
let geq sid1 sid2 cour_ids =
  avge_grade sid1 cour_ids >= avge_grade sid2 cour_ids
```

This results in $O(n^2m)$ database queries where n is the number of students and m is the number of courses. The reason is that there are $O(n^2)$ comparisons during a run of quick sort. Since the resource usage of quick sort depends on the number of courses, we have to make the list of courses an explicit argument and cannot store it in the closure of the comparison function.

```
let rec partition gt acc l =
  match l with
    | [] → let (cs,bs,_) = acc in (cs,bs)
    | x::xs → let (cs,bs,aux) = acc in
      let acc' = if gt x aux then (cs,x::bs,aux)
                 else (x::cs,bs,aux)
      in partition gt acc' xs

let rec qsort gt aux l = match l with | [] → []
  | x::xs →
    let ys,zs = partition (gt x) ([],[],aux) xs in
    append (qsort gt aux ys) (x::(qsort gt aux zs))

let sort_students s_ids c_ids = qsort geq c_ids s_ids
```

In $0.31s$ RAML computes the tight bound $n^2m - nm$ for *sort_students* where n is the length of the argument *s_ids* and m is the length of the argument *c_ids*. The negative factor arises from the translation of the resource polynomials to the standard basis.

Given the alarming cubic bound, we reimplement our sorting function using memoization. To this end we create a table that looks up and stores for each student and course the grade in the DynamoDB. We then replace the function *db_query* with the function *lookup*.

```
let lookup sid cid table =
  let cid_map = find (fun id → id = sid) table in
  find (fun id → id = cid) cid_map
```

For the resulting sorting function, RAML computes the tight bound nm in $0.87s$.

10. Related Work

Our work builds on past research on automatic amortized resource analysis (AARA). AARA has been introduced by Hofmann and Jost for a strict first-order functional language with built-in data types [34]. The technique has been applied to higher-order functional programs and user defined types [41], to derive stack-space bounds [16], to programs with lazy evaluation [52, 58], to object-oriented programs [35, 38], to imperative integer programs [17] and to heap-manipulating programs by integrating it with separation logic [8]. All the aforementioned amortized-analysis–based systems are limited to linear bounds. Hoffmann et al. [29, 31, 32] presented a multivariate AARA for a first-order language with built-in lists and binary trees. Hofmann and Moser [37] have proposed a generalization of this system in the context of (first-order) term rewrite systems. However, it is unclear how to automate this system. In this article, we introduce the first AARA that is able to automatically derive (multivariate) polynomial bounds that depend on user-defined inductive data structures. Our system is the only one that can derive polynomial bounds for higher-order functions. Even for linear bounds, our analysis is more expressive than existing systems for strict languages [41]. For instance, we can for the first time derive an evaluation-step bound for the curried append function for lists. Moreover, we integrated AARA for the first time with an existing industrial-strength compiler.

Type systems for inferring and verifying resource bounds have been extensively studied. Vasconcelos et al. [56, 57] described an automatic analysis system that is based on sized-types [39] and derives linear bounds for higher-order functional programs. Here we derive polynomial bounds.

Dal Lago et al. [43, 44] introduced linear dependent types to obtain a complete analysis system for the time complexity of the call-by-name and call-by-value lambda calculus. Crary and Weirich [20] presented a type system for specifying and certifying resource consumption. Danielsson [22] developed a library for complexity analyses of lazy functional programs that is based on dependent types and manual cost annotations. The advantage of our technique is that it is fully automatic.

Classically, cost analyses are often based on deriving and solving recurrence relations. This approach was pioneered by Wegbreit [59] and is actively studied for imperative languages [1, 5, 7, 25]. These works are not concerned with higher-order functions and bounds do not depend on user-defined data structures. Benzinger [11] has applied Wegbreit's method in an automatic complexity analysis for Nuprl terms. However, complexity information for higher-order functions has to be provided explicitly. Grobauer [26] reported a mechanism to automatically derive cost recurrences from DML programs using dependent types. Danner et al. [23, 24] propose a technique to derive higher-order recurrence relations from higher-order programs. Solving the recurrences is not discussed in these works and in contrast to our work they are not able to automatically infer closed-form bounds.

In an active area of research, techniques from term rewriting are applied to complexity analysis [9, 15, 50]; sometimes in combination with amortized analysis [36]. These techniques are usually restricted to first-order programs and time complexity. Avanzini et al. [10] proposed a complexity preserving defunctionalization to deal with higher-order programs. While the transformation is asymptotically complexity preserving, it is unclear whether this technique can derive bounds with precise constant factors. The implementation of the defunctionalization is combined with an existing first-order bound analysis based on term rewriting. In contrast to RAML, the resulting analysis tool can derive bounds for functions whose cost depends on the sizes of data captured in closures. Advantages of RAML include more precise (non-asymptotic) bounds, amortized analysis, and efficient bound inference using LP solving.

Abstract interpretation based approaches to resource analysis [12, 18, 27, 53, 60] focus on first-order integer programs with loops. Çiçek et al. [19] study a type system for incremental complexity. A similar approach can be used to reason about the difference of the resource usage of two programs [61]. An advantage of our approach is that it is automatic.

Finally, there exists research that studies cost models to formally analyze parallel programs. Blelloch and Greiner [13] pioneered the cost measures work and depth. There are more advanced cost models that take into account caches and IO (see, e.g., Blelloch and Harper [14]), However, these works do not provide machine support for deriving static cost bounds.

11. Conclusion

We have presented important steps towards a practical automatic resource bound analysis for OCaml. Our three main contributions are (1) a novel automatic resource analysis system that infers multivariate polynomial bounds that depend on size parameters of user-defined data structures, (2) the first AARA that infers polynomial bounds for higher-order functions, and (3) the integration of AARA with the OCaml compiler.

As the title of this article indicates, there are many open problems left on the way to a resource analysis system for OCaml that can be used in every-day development. In the future, we plan to improve the bound analysis for programs with side effects and exceptions by adding potential to exceptions and mutable heap cells. We will also work on mechanisms that allow user interaction for manually deriving bounds if the automation fails. Another future research topic is modules and generating efficient function summaries. We are also working on generalization of our resource polynomials to enable the inference of non-polynomial bounds and bounds that depend on records, heights of data structures, and integers [17].

To make the system more practical, we will work on taking into account garbage collection and the runtime system when deriving time and space bounds. Finally, we will investigate techniques to link the high-level bounds with hardware and the low-level code that is produced by the compiler. These open questions are certainly challenging but we are now in a good position to further push the boundaries of resource bound analysis.

Acknowledgments

We thank Zhong Shao, Quentin Carbonneaux, and the members of the PoP Group at Carnegie Mellon for discussions, comments, and suggestions that improved this article and RAML. A special thanks goes to Martin Hofmann, whose suggestion to integrate RAML with OCaml at POPL'14 in San Diego started this work. Finally, we thank the anonymous reviewers for their helpful feedback.

This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082, by NSF under grant 1319671 (VeriQ), and by a Google Research Award. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming (ESOP'07)*, pages 157–172, 2007.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Automatic Inference of Resource Consumption Bounds. In *Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12)*, pages 1–11, 2012.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142 – 159, 2012.

[5] E. Albert, J. C. Fernández, and G. Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, pages 85–100, 2015.

[6] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th International Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[7] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th International Static Analysis Symposium (SAS'12)*, pages 405–421, 2012.

[8] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th European Symposium on Programming (ESOP'10)*, pages 85–103, 2010.

[9] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, pages 55–70, 2013.

[10] M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th International Conference on Functional Programming (ICFP'15)*, pages 152–164, 2012.

[11] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.

[12] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference (LPAR'10)*, pages 103–118, 2010.

[13] G. E. Blelloch and J. Greiner. A Provable Time and Space Efficient Implementation of NESL. In *First International Conference on Functional Programming (ICFP'96)*, pages 213–225, 1996.

[14] G. E. Blelloch and R. Harper. Cache and I/O Efficent Functional Algorithms. In *40th Symposium on Principles of Programming Languages (POPL'13)*, pages 39–50, 2013.

[15] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference (TACAS'14)*, pages 140–155, 2014.

[16] B. Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *18th European Symposium on Programming (ESOP'09)*, pages 190–204, 2009.

[17] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*, pages 467–478, 2015.

[18] P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, pages 105–131, 2015.

[19] E. Çiçek, D. Garg, and U. A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, pages 406–431, 2015.

[20] K. Crary and S. Weirich. Resource Bound Certification. In *27th Symposium on Principles of Programming Languages (POPL'00)*, pages 184–198, 2000.

[21] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX'12)*, pages 3–3, 2003.

[22] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th Symposium on Principles of Programming Languauges (POPL'08)*, pages 133–144, 2008.

[23] N. Danner, D. R. Licata, and R. Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th International Conference on Functional Programming (ICFP'15)*, pages 140–151, 2012.

[24] N. Danner, J. Paykin, and J. S. Royer. A Static Cost Analysis for a Higher-Order Language. In *7th Workshop on Programming Languages Meets Program Verification (PLPV'13)*, pages 25–34, 2013.

[25] A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposium (APLAS'14)*, pages 275–295, 2014.

[26] B. Grobauer. Cost Recurrences for DML Programs. In *6th International Conference on Functional Programming (ICFP'01)*, pages 253–264, 2001.

[27] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th Symposium on Principles of Programming Languauges (POPL'09)*, pages 127–139, 2009.

[28] J. Hoffmann. RAML Web Site. `http://raml.co`, 2016.

[29] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*, pages 287–306, 2010.

[30] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Programming Languages and Systems - 8th Asian Symposium (APLAS'10)*, pages 172–187, 2010.

[31] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*, pages 357–370, 2011.

[32] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Transactions on Programming Languages and Systems*, 34:14:1–14:62, 2012.

[33] J. Hoffmann, A. Das, and S.-C. Weng. Towards Automatic Resource Bound Analysis for OCaml. *ArXiv e-prints*, 2016.

[34] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197, 2003.

[35] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming (ESOP'06)*, pages 22–37, 2006.

[36] M. Hofmann and G. Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, pages 272–286, 2014.

[37] M. Hofmann and G. Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 241–256, 2015.

[38] M. Hofmann and D. Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *22nd European Symposium on Programming (ESOP'13)*, pages 593–613, 2013.

[39] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, 1996.

[40] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-World Performance Bugs. In *33rd Conference on Programming Language Design and Implementation PLDI'12*, pages 77–88, 2012.

[41] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th Symposium on Principles of Programming Languages (POPL'10)*, pages 223–236, 2010.

[42] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - 16th Annual International Cryptology Conference (CRYPTO'96)*, pages 104–113, 1996.

[43] U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symposium on Logic in Computer Science (LICS'11)*, pages 133–142, 2011.

[44] U. D. Lago and B. Petit. The Geometry of Types. In *40th Symposium on Principles of Programming Languages (POPL'13)*, pages 167–178, 2013.

[45] X. Leroy. The ZINC Experiment: An Economical Implementation of the ML Language. Technical report 117, INRIA, 1990.

[46] X. Leroy. From Krivine's Machine to the Caml Implementation. `http://pauillac.inria.fr/~xleroy/talks/zam-kazam05.pdf`, 2005. Invited talk, May 17, Workshop on the Krivine and ZINC Abstract Machines.

[47] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[48] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02. Technical report, Institut National de Recherche en Informatique et en Automatique, 2014.

[49] V. Nicollet. 99 problems (solved) in ocaml. `https://ocaml.org/learn/tutorials/99problems.html`, 2016.

[50] L. Noschinski, F. Emmes, and J. Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.

[51] O. Olivo, I. Dillig, and C. Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Conference on Programming Language Design and Implementation (PLDI'15)*, pages 369–378, 2015.

[52] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th International Conference on Functional Programming (ICFP'12)*, pages 165–176, 2012.

[53] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference (CAV'14)*, page 743–759, 2014.

[54] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[55] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer US, 2001.

[56] P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[57] P. B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *15th International Conference on Implementation of Functional Languages (IFL'03)*, pages 86–101, 2003.

[58] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, pages 787–811, 2015.

[59] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[60] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th International Static Analysis Symposium (SAS'11)*, pages 280–297, 2011.

[61] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017. Forthcoming.