Abstract
Understanding the resource usage of programs is crucial for developing software that is safe, secure, and efficient. Consequently, there is ongoing interest in the development of techniques that provide software developers with support for inferring resource bounds at compile time. This article introduces a new resource analysis system for OCaml programs. The system automatically derives worst-case resource bounds for higher-order, polymorphic programs with side-effects and user-defined inductive types. The technique is parametric in the resource and can derive bounds for time, memory, and energy usage. The derived bounds are multivariate resource polynomials that are functions of different size parameters that depend on the standard OCaml types. Bound inference is fully automatic and reduced to a standard 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. For the first time, it is possible to automatically derive polynomial bounds for higher-order functions and polynomial bounds that depend on user-defined inductive types. Moreover, the analysis handles 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 the integration with Inria’s OCaml compiler. 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.

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 among the bugs that are most difficult to detect [40, 46] and large software systems are plagued by performance problems. 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 integers 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, 48], recurrence relations [3, 4], and abstract interpretation itself [18, 54]. Automatic resource analysis techniques for functional programs are based on sized types [50], recurrence relations [23], term-rewriting [10], and amortized resource analysis [32, 34, 41, 47].

Despite major steps forward, there are still many obstacles to overcome to make resource analysis technologies 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, 50]. Furthermore, techniques that can derive polynomial bounds are limited to bounds that depend on predefined lists and binary trees [29, 32] or integers [15, 48]. Finally, resource analyses for functional programs have been implemented for custom languages that are not supported by mature tools for compilation and development [32, 34, 41, 47, 50].

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 on three main contributions 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 system that infers multivariate polynomial bounds that depend on size parameters of complex user-defined 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 for many complex example programs such as sorting algorithms with complex comparison functions, Dijkstra’s single-source 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 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. 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. 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. These novel multivariate resource polynomials are a substantial generalization of the resource polynomials that have been previously defined for lists and binary trees [32]. 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 [32] 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 has not been possible previously [41].

We performed experiments on more than 3000 lines of OCaml code. While it is still not straightforward to automatically analyze complete existing applications, it is easy to develop and analyze real OCaml applications if we keep the current capabilities of the system in mind. In Section 8, 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 a database.

2. Overview

Before we describe the technical development, we give a short overview of the challenges and achievements of our work. Carrying 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 \( \text{append} : \alpha \text{list} \to \alpha \text{list} \to \alpha \text{list} \) in OCaml. At first glance, we might say that the time complexity of \( \text{append} \) is \( O(n) \) if \( n \) is the length of the first argument. But a closer inspection of the definition of \( \text{append} \) reveals that this is a gross simplification. In fact, the complexity of the partial function call \( \text{app\_par} = \text{append} \ell \) is constant. Moreover, the complexity of the function \( \text{app\_par} \) is linear— not in the length of the argument but in the length of the list \( \ell \) 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 [32, 41].

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 \( \text{append} \) is partially applied to one argument then a function closure is created as expected. However— and this is one of the reasons of OCaml’s great performance— if \( \text{append} \) 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 curried 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, \( \text{append} \) can have both of the following types

\[
\alpha \text{list} \to \alpha \text{list} \to \alpha \text{list} \quad \text{and} \quad [\alpha \text{list}, \alpha \text{list}] \to \alpha \text{list}.
\]

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 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 to not derive bounds that asymptotically depend on data captured in function closures to keep the complexity of the system at a manageable level.

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

The conditions under which functions can be analyzed might look complex at first but they can be boiled down to 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 extend, 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 \( \text{map} \) for lists even when the higher-order argument is not available. The function \( \text{map} \) can have the following types

\[
(\alpha \to \beta) \to \alpha \text{list} \to \beta \text{list} \quad [\alpha \to \beta, \alpha \text{list}] \to \beta \text{list}
\]

Unlike \( \text{append} \), the resource usage of \( \text{map} \) 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 \( \text{map} \) 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 \( \text{map} \) 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 \( \text{map} \) itself contributes to the total cost of a program. Second, the time bound for \( \text{map} \) proves that \( \text{map} \) is guaranteed to terminate if the higher-order argument terminates for every input.

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

\[
\begin{align*}
\text{let rec rec\_scheme f l =} \\
\text{match l with [ [] ] \to [ ]} \\
\text{x::xs \to rec\_scheme f (f l));} \\
\text{let g = rec\_scheme tail;;}
\end{align*}
\]

Here, RAML is not able to derive an evaluation-step bound for \( \text{rec\_scheme} \) since the number of evaluation steps (and even termination) depends on the function \( f \). However, RAML derives the tight evaluation-step bound \( 12n + 7 \) for the function \( g \).

Polynomial Bounds and Inductive Types. Existing AARA systems are either limited to linear bounds [34, 41] or to polynomial bounds that are functions of the sizes of simple predefined list and...
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 (quick sort 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 (after 0.68s run time):
10 + 23*K + M + 32*L + M + 20*L + M + Y + 13 + L + M + Y^2

where
- M is the num. of ::-nodes of the 2nd comp. of the arg.
- L is the fraction of Right-nodes in the ::-nodes of the 2nd component of the argument
- Y is the maximal num. of ::-nodes in the Right-nodes in the ::-nodes of the 2nd component of the arg.
- K is the fraction of Left-nodes in the ::-nodes of the 2nd component of the argument

Figure 1. Modified challenge example from Avanzini et al. [10] and shortened output of the automatic bound analysis performed by RAML for the function rev_sort. The derived bound is a tight bound on the number of evaluation steps in the big-step semantics if we do not take into account the cost of the higher-order argument f.

binary-tree data structures [32]. In contrast, this work presents the first analysis that can derive polynomial bounds that depend on size parameters of complex user-defined data structures.

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

$$\lambda a.\left[\alpha \mid a_i \text{ is an } A_i \text{-node in } a \text{ and if } i < j \text{ then } a_i \prec a_j\right]$$

where α is an inductive data structure with constructors $A_1, \ldots, A_m$, $\alpha = a_1, a_2, \ldots, a_n$, and $\prec$ denotes the pre-order on the tree a. 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 [28]. As an interesting special case, we can also derive conditional 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 and only if x is True.

Effects. Our analysis handles references and arrays by ensuring that resource cost does not asymptotically depend on values that have been stored in mutable cells. While it has been shown that it is possible to extend AARA to handle mutable state [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 possible interactions with mutable state, such as storing functions in references.

Example Bound Analysis. To demonstrate some of the capabilities of the new analysis system, Figure 1 shows the output of RAML for a concrete example. The code is an adoption of a challenging 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 can automatically derive a tight linear bound on the number of evaluation steps used by rev.

To show more features of our analysis, we modified Avanzini et al.’s rev in Figure 1 by adding an additional argument f and a pattern match to the definition of the function walk. The resulting type of walk is

$$(\alpha \to \alpha \to \text{bool}) \to [(\beta \cdot \alpha) \text{ either list}; (\beta \cdot \alpha) \text{ either list}] \to (\beta \cdot \alpha) \text{ either list}$$

Like before the modification, walk is essentially the append.reverse 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 1. Since the comparison function for quicksort (argument f) is not available, RAML assumes that it does not consume any resources during the analysis. If rev_sort is applied to a concrete argument f then the analysis is repeated to derive a bound for this instance.

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, 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. For the purpose of this article, the syntax of Core RAML expressions is defined by the following grammar. The actual core expressions also contain constants and operators for primitive data types such as integer, float, and boolean; 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.

\[
\begin{align*}
  e &::= x \mid x \cdot x_1 \cdots x_n \mid C \cdot x \mid \lambda x.e \mid \text{ref } x \mid \text{let } x = x_1 := x_2 \mid \text{match } x \text{ with } C \cdot y \rightarrow e_1 \mid e_2 \mid \text{let } x = x_1, x_2 \rightarrow e \mid \text{share } x \text{ as } (x_1, \ldots, x_n) \rightarrow e \mid \text{let rec } F \text{ in } e \\
  F &::= f \mid \lambda x.e \mid F_1 \text{ and } F_2
\end{align*}
\]

The syntax contains forms for variables, function application, data constructors, lambda abstraction, references, tuples, pattern matching, and (recursive) binding. For simplicity, we only allow recursive definitions of functions. 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 determine if an expression like $f \cdot x_1 \cdot x_2$ is parsed as $(f \cdot x_1) \cdot x_2$ or $(f \cdot x_1) \cdot x_2$. 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 $e$.

We focus on this set of language features since it is sufficient to present the main contributions of our work. We sometimes take the
A value $V(x) = \ell$ is given by

$$\frac{S \neq \cdot}{V(H_M = x \Downarrow (\ell, H)) | M^{app}} \quad (E:\text{Var})$$

and

$$\frac{S, V.H_M = e \Downarrow \circ \; \circ}{S, V.H_M = e \Downarrow 0} \quad (E:\text{Abort})$$

with respect to a resource metric $M$. The sub-evaluations one has to also take into account the number of watermark $x$ and function closure $p$.

The semantics of Core RAML is formulated with respect to an environment $V$ and a heap $H$. A stack (to store arguments for function application), an environment, executions by inductively describing finite subtrees of infinite resources are returned. Second, it models terminating and diverging constant cost for each evaluation step. If this cost is negative then resources are returned. If this cost is negative then resources are returned. Second, it models terminating and diverging constant cost for each evaluation step. If this cost is negative then resources are returned. Second, it models terminating and diverging constant cost for each evaluation step. If this cost is negative then resources are returned.

The semantics of Core RAML is formulated with respect to a stack (to store arguments for function application), an environment, and a heap. A heap $H$ is a finite partial mapping $V : \text{Loc} \rightarrow \text{Val}$ that maps locations to values. An environment is a finite partial mapping $V : \text{Var} \rightarrow \text{Loc}$ from variable identifiers to locations. An argument stack $S ::= \cdot$ is a finite list of locations. The set of RAML values $\text{Val}$ is given by

$$v ::= \ell \mid (\ell_1, \ldots, \ell_k) \mid (\lambda x. e, V) \mid (C, \ell)$$

A value $v \in \text{Val}$ is either a location $\ell \in \text{Loc}$, a tuple of locations $(\ell_1, \ldots, \ell_k)$, a function closure $(\lambda x.e, V)$, or a node of a data structure $(C, \ell)$ where $C$ is a constructor and $\ell$ is a location. In a function closure $(\lambda x.e, V)$, $V$ is an environment, $e$ is an expression, and $x$ is a variable.

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

The big-step operational evaluation rules in Figure 2 are formulated with respect to a resource metric $M$. They define an evaluation judgment of the form

$$S, V.H_M = e \Downarrow (\ell, H') \mid (q, q')$$

It expresses the following. If the argument stack $S$, the environment $V$, and the initial heap $H'$ are given then the expression $e$ evaluates to the location $\ell$ and the new heap $H''$. The evaluation of $e$ needs $q \in \mathbb{Q}^{\text{res}}$ resource units (watermark) and after the evaluation there are $q' \in \mathbb{Q}^{\text{res}}$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity $\delta$ 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 $\odot$. The judgement has the general form

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

Intuitively, this evaluation statement expresses that the watermark of the resource consumption after some number of evaluation steps is $q$ and there are currently $q'$ resource units left. A resource metric $M : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$ defines the resource consumption in each evaluation step of the big-step semantics where $K$ is a set of constants. We write $M^k_n$ for $M(k, n)$.

It is handy to view the pairs $(q, q')$ in the evaluation judgments as elements of a monoid $\mathbb{Q}^{\text{res}} = \mathbb{Q}^{\text{res}} \times \mathbb{Q}^{\text{res}}$. 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 $(0, 0)$ $(p, 0)$ is just $(q + p, 0)$. We identify a rational number $q$ with an element of $\mathbb{Q}$ 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 $K$ that appear in the rules can be negative. In the semantic rules
we use the notation $H' = H$, $\ell \mapsto v$ to indicate that $\ell \notin \text{dom}(H)$, $\text{dom}(H') = \text{dom}(H) \cup \{\ell\}$, $H'(\ell) = v$, and $H'(x) = H(x)$ for all $x \neq \ell$.

To model the treatment of function application in Inria’s OCaml compiler, we use a stack $S$ on which we store the locations of function arguments. The only rules that push locations to $S$ are $E:APPEnd$. 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 $\langle \lambda x.e, V \rangle$ we inspect the argument stack $S$. If $S$ contains a location $\ell$ then we pop it form the stack $S$, bind it to the argument $x$, and evaluate the function body $c$ in the new environment $V \setminus x \mapsto \ell$. This is defined by the rule $E:ABSBind$ and indirectly by the rule $E:VARApp$. Another rule that modifies the argument stack is $E:LET2$. Here, we evaluate the subexpression $e_1$ with an empty argument stack because the arguments on the stack when evaluating the let expressions are consumed by the result of the evaluation of $e_2$.

The argument stack accurately captures Inria’s OCaml compiler’s behavior to avoid the creation of intermediate function closures. It also extends naturally to the evaluation of expressions that are not in share-let-normal form. As we will see in Section 6, the argument stack is also necessary to prove the soundness of the multivariate resource bound analysis.

Another important feature of the big-step semantics, is that it can model failing and diverging evaluations by allowing partial derivation judgments that can be used to derive the resource usage after $n$ steps. Technically, this is realized by the rule $E:ABORT$ which can be applied at any point to abort the current evaluation without additional resource cost. The mechanism of aborting an evaluation is most visible in the rules $E:LET1$ and $E:LET2$: During the evaluation of a let expression we have two possibilities. The first possibility is that the evaluation of the subexpression $e_1$ is aborted using $E:ABORT$ at some point. We can then apply the rule $E:LET1$ to pass the resource usage before the abort. The second possibility is that $e_1$ evaluates to a location $\ell$. We can then apply the $E:LET2$ to bind $\ell$ to the variable $x$ and evaluate the expression $e_2$.

4. Simple Type System

In this section, we introduce a type system that is a refinement of OCaml’s type system. In this type system, we mirror the resource-aware type system and introduce some particularities that explain features of the resource-aware types. For the purpose of this article, we define simple types as follows.

$$T ::= X \mid T \rightarrow \{ [T_1, \ldots, T_n] \rightarrow T \mid \mu X. \langle C_1 : T_1 \times X^{n_1}, \ldots, C_k : T_k \times X^{n_k} \rangle \}$$

A (simple) type $T$ is an uninterpreted type variable $X \in \mathcal{X}$, a type $T$ ref of references of type $T$, a tuple type $T_1 \times \cdots \times T_n$, a function type $[T_1, \ldots, T_n] \rightarrow T$, or an inductive data type $\mu X. \langle C_1 : T_1 \times X^{n_1}, \ldots, C_k : T_k \times X^{n_k} \rangle$.

Two parts of this definition are non-standard and deserve further explanation. First, bracket function types $[T_1, \ldots, T_n] \rightarrow T$ correspond to the standard function type $T_1 \times \cdots \times T_n \rightarrow T$. The meaning of $[T_1, \ldots, T_n] \rightarrow T$ is that the function is applied to its first $n$ arguments at the same time. The type $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T$ 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 program. We account for that with let polymorphism (see the following subsection).

Also note that $[T_1, \ldots, T_n] \rightarrow T$ still describes a higher-order function while $T_1 \times \cdots \times T_n \rightarrow T$ describes a first-order function with $n$ arguments.

Second, inductive types are required to have a particular form. This makes it possible to track cost that depends on size parameters of values of such types. It is of course possible to allow arbitrary inductive types and not to track such cost. Such an extension is straightforward and we do not present it in this article.

We assume that each constructor $C \in \mathcal{C}$ is part of at most one recursive type. Furthermore we assume that each recursive type has at least one constructor. For an inductive type $T = \mu X. \langle C_1 : T_1 \times X^{n_1}, \ldots, C_k : T_k \times X^{n_k} \rangle$ we sometimes write $T = \langle C_1 : (T_1, n_1), \ldots, C_k : (T_k, n_k) \rangle$. We say that $T_1$ is the node type and $n_i$ is the branching number of the constructor $C_i$. The maximal branching number $n = \max\{n_1, \ldots, n_k\}$ of the constructors is the branching number of $T$.

4.1. Let Polymorphism and Sharing. Modelling 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 $\text{share} x$ as $(x_1, x_2)$ 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 $x$ an expression $e$ in RAML. We then introduce sharing constructs, and replace the occurrences of $x$ in $e$ with the new names 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 it would mean to type the expression $\text{map} [\text{map} \rightarrow \text{map}']$ instead of typing the expression let map = map in $e$. In principle, it would be possible to treat sharing of variables in a similar way as let polymorphism. But if we start form an expression let $x = e_1$ 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 $f = \lambda x.e.f$ in $e$ that is used twice in $e$. Then the usual approach to enable the analysis in our system would be to use sharing

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

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:

$$\text{let } f_1 = \lambda x.e.f \text{ in } f_2 = \lambda x.e.f \text{ in } e'$$

The functions $f_1$ and $f_2$ can now have different types. This method can cause an exponential blow up of the size of the expression. It is nevertheless appealing because it enables us to treat resource polymorphism in the same way as let polymorphism.

4.2. Type Judgements. Type judgements have the form

$$\Sigma, \Gamma \vdash e : T$$

where $\Sigma = T_1, \ldots, T_n$ is a list of types, $T : \text{Var} \rightarrow T$ is a type context that maps variables to types, $e$ is a core expression, and $T$ is a (simple) type. The intuitive meaning (which is formalized later in this section) is as follows. Given an evaluation environment that
matches the type context $\Gamma$ and an argument stack that matches the type stack $\Sigma$ then $e$ evaluates to a value of type $T$.

The most interesting feature of the type judgements is the handling of bracket function types $[T_1, \ldots, T_n] \rightarrow T$. Even though function types can have multiple forms, a well-typed expression has often a unique type (in a given type context). This type is derived from the way a function is used. For instance, we have $\lambda f.\lambda x.\lambda y.((\{T_1, T_2\} \rightarrow T) \rightarrow T_1 \rightarrow T_2 \rightarrow T)$ and $\lambda f.\lambda x.\lambda y.((f y) \rightarrow T_1 \rightarrow T_2 \rightarrow T) \rightarrow T_1 \rightarrow T_2 \rightarrow T$, and the two function types are unique.

A type $T$ of an expression $e$ has a unique type derivation that produces a type judgement $\Gamma ; e : T$ with an empty type stack. We call this canonical type derivation for $e$ and a closed type judgement. If $T$ is a function type $\Sigma \rightarrow T'$ then there is a second type derivation for $e$ that we call an open type derivation. It derives the open type judgement $\Sigma ; \Gamma ; e : T'$ where $|\Sigma| > 0$. The following lemma can be proved by induction on the type derivations.

**Lemma 1.** $\Sigma ; \Gamma ; e : T$ if and only if $\Sigma ; \Gamma ; e : T$.

Open and canonical type judgements are not interchangeable. An open type judgement $\Sigma ; \Gamma ; e : T$ can only appear in a derivation with an open root of the form $\Sigma ; \Sigma ; \Gamma ; e : T$, or in a subtree of a derivation whose root is a closed judgement of the form $\Sigma ; \Gamma ; e : T$ where $|\Sigma'| > 0$. In other words, in an open derivation $\Sigma ; \Gamma ; e : T$, 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 as most one open type derivation.

**Type Rules.** Figure 3 presents selected type rules of the type system. As usual $\Gamma_1$, $\Gamma_2$ denotes the union of the type contexts $\Gamma_1$ and $\Gamma_2$ provided that $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$. We thus have the implicit side condition $\text{dom}(\Gamma_1) \cap \text{dom}(\Sigma) = \emptyset$ whenever $\Gamma_1, \Gamma_2$ occurs in a typing rule. Especially, writing $\Gamma = \Gamma_1 : T_1, \ldots, x_k : T_k$ means that the variables $x_i$ are pairwise distinct.

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. (We view the two rules for pattern match and let binding as one rule, respectively.) The type stack is modified by the rules T:VAR_PUSH, T:APP_PUSH, T:ABS_PUSH, and T:ABS_POP. For every leaf rule that can return a function type, such as T:VAR, T:APP, and T:APP_POP, we add a second rule that derives the equivalent open type. The reason becomes clear in the resource-aware type system in Section 6. The rules that directly control the shape of the function types are T:ABS_PUSH and T:ABS_POP for lambda abstraction. While the other rules are (deterministically) syntax driven, the rules for lambda abstraction introduce a non-deterministic choice. However, there is often only one possible choice depending on how the abstracted function is used.

As mentioned, the type system is affine and every variable in a context can at most be used once in the typed expression. Multiple uses have to be introduced explicitly using the rule T:SHARE. The only exception is the rule T:LET_REC. Here we allow the use of the context $\Delta$ in the body of all defined functions. The reason for this is apparent in the resource aware version: sharing of function types is always possible without any restrictions.

**Well-Formed Environments.** For each simple type $T$ we inductively define a set $[T]$ of values of type $T$. Our goal here is not to advance the state of the art in denotational semantics but rather to capture the tree structure of data structures stored on the heap. To this end, we distinguish mainly inductive types (possible inner nodes of the trees) and other types (leaves). For the formulation of type soundness, we also require that function closures are well-formed.

We simply interpret polymorphic data with the set of locations $\text{Loc}$.

- \([X] = \text{Loc}\)
- \([T \text{ref}] = \{\text{ref}(a) \mid a \in [T]\}\)
- \([\Sigma \rightarrow T] = \{(\lambda x.e, v) \mid \exists \Sigma : H ; \Gamma \vdash V ; \Gamma \vdash x ; e : \Sigma \rightarrow T\}\)
- \([T_1 \ast \ast \ast T_n] = \{T_1 \times \ast \ast \ast T_n\}\)
- \([B] = \text{tr}(B)\) if $B = (C_1(T_1, n_1), \ldots, C_r(T_r, n_r))$

Here, $T = \text{tr}((C_1(T_1, n_1), \ldots, C_r(T_r, n_r)))$ is the set of trees $\tau$ with node labels $C_1, \ldots, C_r$ which are inductively defined as follows. If $i \in \{1, \ldots, k\}$, $a_i \in [T_i]$, and $\tau_j \in T$ for all $1 \leq j \leq n_i$ then $C_i(a_{i_1}, \tau_1, \ldots, \tau_{n_i}) \in T$.

If $H$ is a heap, $\ell$ is a location, $A$ is a type, and $a \in [A]$ then we write $H \models \ell \mapsto a : A$ to mean that $\ell$ defines the semantic value $a \in [A]$ when pointers are followed in $H$ in the obvious way. The judgment is formally defined in Figure 4. For a heap $H$ there may exist different semantic values $\alpha$ and simple types $A$ such that $H \models \ell \mapsto \alpha : A$. However, if we fix a simple type $A$ and a heap $H$ then there exists at most one value $\alpha$ such that $H \models \ell \mapsto \alpha : A$.

We write $H \models \ell : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [A]$ so that $H \models v \mapsto a : A$. An environment $V$ and a heap $H$ are well-formed with respect to a context $\Gamma$ if $H \models V(x) : \Gamma(x)$ holds for every $x \in \text{dom}(\Gamma)$. We then write $H \models V : \Gamma$. Similarly, an argument stack $S = \ell_1, \ell_2, \ldots, \ell_n$ is well-formed with respect to a type stack $\Sigma = T_1, \ldots, T_n$ in heap $H$, written $H \models S : \Sigma$, if $H \models \ell_i : T_i$ for all $1 \leq i \leq n$.

Note that the rules in Figure 4 are interpreted coinductively. The reason is that in the rule V:FUN, the location $\ell$ can be part of the closure environment $V$ if the closure has been created with the rule E:LET_REC. The influence of the coinductive definition on the proofs is minimal since all proofs in this article are by induction.
Type Preservation. Theorem 1 shows that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment.

Theorem 1. If $\Sigma, \Gamma \vdash e : T$, $H \vdash V : \Gamma$, $H \vdash S : \Sigma$, and $S, V, H \mid \mid A \mid e \mid (q, q')$ then $H' \vdash V : \Gamma$, $H' \vdash S : \Sigma$, and $H' \vdash A : T$.

Theorem 1 is proved by induction on the evaluation judgement.

5. Multivariate Resource Polynomials

In this section we define the set of resource polynomials which is a search space of our automatic resource bound analysis. A resource polynomial $p : [T] \rightarrow \mathbb{Q}^+$ maps a semantic value of some simple type $T$ to a non-negative rational number.

An analysis of typical polynomial computations operating on a list $[a_1, \ldots, a_n]$ shows that it consists of operations that are executed for every $k$-tuple $(a_{i_1}, \ldots, a_{i_k})$ with $1 \leq i_1 < \cdots < i_k \leq n$. The simplest examples are linear map operations that perform some operation for every $a_i$. Other common examples are sorting algorithms that perform comparisons for every pair $(a_i, a_j)$ with $1 \leq i < j \leq n$ 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 $(a_1, \ldots, a_k)$ 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 $i$.

As in previous work, which considered binary trees, we will essentially interpret all tree-like data structures as lists with different nodes by flattening them in pre-order. As a result, our resource polynomials only depend on the number of nodes of a certain kind in a tree but not on structural measures like the height of the tree.

To include the height into resource polynomials in a general way, we would need a way to express a maximum (or a choice) in the resource polynomials. We leave this for future research in favor of compositional and modularity. In practice, it is useful that the potential of a data structure is invariant under changes in the structure of the tree.

Base Polynomials and Indices. In Figure 5, we define for each simple type $T$ a set $P(T)$ of functions $p : [T] \rightarrow \mathbb{N}$ 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.

\[
X \in X \quad \ell \in \text{dom}(H) \\
H \models \ell \Rightarrow X : X \\
H \models \ell \Rightarrow \ell : X \\
H \models \ell \Rightarrow R(a) : T : \text{ref} \\
H \models \ell \Rightarrow (\lambda x.e, V) : \Sigma \Rightarrow T \\
H \models \ell \Rightarrow (\lambda x.e, V) : \Sigma \Rightarrow T \\
H \models (\ell, \ldots, \ell_n) : a_1 : \ldots : a_n : T_1 : \ldots : T_n \\
H \models \ell \Rightarrow (a_1, \ldots, a_n) : T_1 : \ldots : T_n
\]

(V:TVAR) (V:REF) (V:FUN) (V:TUPLE) (V:CONS)

Figure 4. Coinductively relating heap cells to semantic values.

\[
\lambda a. \mathbf{1} \in P(T) \\
\lambda a. \prod_{i=1}^k p_i(a_i) \in P(T_1 \ast \cdots \ast T_k) \\
B = (C_1 : (T_1, n_1), \ldots, C_m : (T_k, m_m)) \\
\mathbb{C} = [C_{j_1}, \ldots, C_{j_k}] \\
\forall i : p_i \in P(T_i) \\
\lambda b. \sum_{a \in \mathbb{P}B} p_i(a_i) \in P(B) \\
B = (C_1 : (T_1, n_1), \ldots, C_k : (T_m, m_m)) \\
\forall i : I_i \in \mathbb{I}(T_i) \\
[(I_1, C_{j_1}), \ldots, (I_k, C_{j_k})] \in \mathbb{B}
\]

Figure 5. Defining the set $P(T)$ of base polynomials for type $T$.

Let $B = (C_1 : (T_1, n_1), \ldots, C_k : (T_m, m_m))$ be an inductive type. Let $\mathbb{C} = [C_{j_1}, \ldots, C_{j_k}]$ be a list of $B$-constructors and $b \in [B]$. We inductively define a set $\mathbb{P}B(C, b)$ of $k$-tuples as follows: $\mathbb{P}B(C, b)$ is the set of $k$-tuples $(a_1, \ldots, a_k)$ such that $C_{j_1}(a_1, b_1), \ldots, C_{j_k}(a_k, b_k)$ are nodes in the tree $b \in [B]$ and $C_{j_1}(a_1, b_1) \prec_{pr} \cdots \prec_{pr} C_{j_k}(a_k, b_k)$ for the pre-order $\prec_{pr}$ 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 $\lambda a. e(a)$.

In Figure 6, we inductively define for each simple type $T$ a set of indices $\mathbb{I}(T)$. For tuple types $T_1 \ast \cdots \ast T_k$ we identify the index $\times$ with the index $(\ast, \ldots, \ast)$. Similarly, we identify the index $\ast$ with the index $[\ast]$ for inductive types.

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

$\mathbb{P}T(a) = 1 \\
p_{(i_1, \ldots, i_k)}(a_1, \ldots, a_k) = \prod_{j=1}^k p_{i_j}(a_j) \\
p_{(i_1, \ldots, i_k)}(b) = \sum_{a \in \mathbb{P}B([C_1, \ldots, C_m], k)} \prod_{j=1}^k p_{i_j}(a_j)
$
(...singleton sum). Furthermore \( p(\cdot, \text{Nil}) (\text{Nil}(i)) = 1 \) because 
\[ \tau_{\text{singleton}}([\cdot, \text{Nil}], \text{Nil}(i)) = \text{Nil}(i) \] 
and \( P(\text{unit}) = \lambda a \cdot 1 \).

Let us now consider the usual sum type
\[ \text{sum}(T_1, T_2) = \mu X \langle \text{Left} : T_1, \text{Right} : T_2 \rangle \text{.} \]
Then \( \text{sum}(T_1, T_2) = \{ \text{Left}(a) \mid a \in [T_1] \} \cup \{ \text{Right}(b) \mid b \in [T_2] \} \). If we define
\[ \sigma_C(p)(C') \left\{ \begin{array}{ll} p(a) & \text{if } C = C' \\ 0 & \text{otherwise} \end{array} \right. \]
then \( P(\text{sum}(T_1, T_2)) = \{ x \mapsto 1, x \mapsto 0 \} \cup \{ \sigma_{\text{left}}(p) \mid p \in P(T_1) \} \cup \{ \sigma_{\text{right}}(p) \mid p \in P(T_2) \} \).

The next example is the list type
\[ \text{list}(T) = \mu X \langle \text{Cons} : T \times X, \text{Nil} : \text{unit} \rangle \text{.} \]
Then \( \text{list}(T) = \{ \text{Nil}(()) , \text{Cons}(a_1, \text{Nil}()) , \ldots \} \) and we write
\[ \text{list}(T) = \{ [], [a_1], [a_1, a_2], \ldots \mid a_i \in [T] \} \text{.} \] It holds that
\( \tau_{\text{cons}}([\cdot, \text{Cons}], [a_1, \ldots, a_n]) = [a_1, \ldots, a_n] \) and moreover
\( \tau_{\text{nil}}([\cdot, \text{Cons}], [\cdot]) = \{ a_1, a_j \mid 1 \leq i < j \leq n \} \). More general,
\( \tau_{\text{cons}}([\cdot, \text{Cons}], [a_1, \ldots, a_n]) = \{ a_1, \ldots, a_k \} \)
if \( 1 \leq i_1 < \cdots < i_k \leq n \) if \( C = ([\cdot, \text{Cons}], \ldots, [\cdot]) \). On the other hand,
\( \tau_{\text{nil}}([\cdot, \text{Cons}], [a_1, \ldots, a_n]) = \emptyset \) if \( D = ([\cdot, \text{Nil}], \ldots, [\cdot]) \).
Finally consider a list type with two different Cons-nodes
\[ \text{list2}(T_1, T_2) = \mu X \langle \text{Cons} : T_1 \times X, \text{Cons} : T_2 \times X, \text{Nil} : \text{unit} \rangle \text{.} \]
Then \( \text{list2}(T) = \{ [], [a_1], [a_1, a_2], \ldots \mid a_i \in \{ C_1, C_2 \} \times [T] \} \).
We furthermore have \( \tau_{\text{cons}}([\cdot, \text{Cons}], [b_1, \ldots, b_n]) = \{ b_1, \ldots, b_n \} \)
if \( \forall j \exists a : b_j = (C_1, a) \) and 
\( \tau_{\text{nil}}([\cdot, \text{Cons}], [\cdot, \text{Cons}], [b_1, \ldots, b_n]) = \{ (b_1, b_2), \ldots \} \)
if \( \forall j \exists a : b_j = (C_1, a) \). Then we define a heap and
\[ \text{type annotations so that type derivations correspond to proofs of resource bounds.} \]

6. Resource-Aware Type System
In this section, we describe the resource aware type system. Essentially, we annotate the simple type system from Section 4 with resource annotations so that type derivations correspond to proofs of resource bounds.

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

A type annotation for a simple type \( T \) is defined to be a family
\[ Q_T = q_T \in \mathcal{Z}(T) \text{ with } q_T \in Q_T^0. \]
We write \( Q(T) \) for the set of type annotations for the 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.
\[ A ::= X | A \rightarrow \text{ref} | A_1 \cdots \cdots A_n \]...
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 be a type stack, be a context and let be an annotation for a context with respect to be an annotation for a context for a context that is defined by

\[
q'_i((j_1, j_2, \ldots, j_n)) = \begin{cases} q_i((j_1, j_2, \ldots, j_n)) & j_0 = 0 \\ q_i((j_1, j_2, \ldots, j_n)) & j_0 \neq 0 \end{cases}
\]

Here, is the concatenation of the lists \(L_1, \ldots, L_n\).

\section{Lemma 2.} Let be an inductive data type. Then be an annotated context, \(H \vdash V : \Gamma, \alpha : B, Q \vdash e : (A, B)\) is an annotation for a context with respect to be an annotation for a context for a context that is defined by

\[
\phi_{S, V, H}(\Sigma; \Gamma, y : A ; B, Q) = H(A, B) = (C, \ell, V') \quad \text{and} \quad \ell' = V[y \mapsto \ell].
\]

Then \(H \vdash V' : \Gamma, \alpha : B^\alpha\) and \(\phi_{S, V, H}(\Sigma; \Gamma, y : A ; B, Q) = \phi_{S, V, H}(\Sigma; \Gamma, y : A \check{B}^\alpha, \ell')\).

\section{Sharing.} Let be an annotated context. The sharing operation \(\overline{Q}\) 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 3 shows that sharing is a linear operation that does not lead to any loss of potential.

\section{Lemma 3.} Let be a data type. Then there are natural numbers \(c_{i}^{j}(k)\) for \(i, j, k \in \mathbb{N}\) such that the following holds. For ev-
every context $\Sigma; \Gamma, x_1:A, x_2:A; Q$ and every $H, V$ with $H \models V : \Gamma, x:A$ and $H \models S : \Sigma$ it holds that $\Phi_{S, V, H}(\Sigma; \Gamma, x:A; Q') = \phi_{S, V, H}(\Sigma; \Gamma, x:A, x_2:A; Q)$ where $V' = V[x_1, x_2 \mapsto V(x)]$ and $q_{(j,k)} = \sum_{j\in J} e_{(k)} q_{(j,k)}$.

The coefficients $e_{(j,k)}$ 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 [33]. For a context $\Sigma; \Gamma, x_1:A, x_2:A; Q$ we define $\mathcal{Q} \subseteq \mathcal{Q}'$ from Lemma 3.

**Type Judgements.** A function type is a subtype of another function type if it allows more arguments. It is however unclear what such a type means for a user and we prefer a more declarative view that clearly separates type checking and type inference. An open problem with constraint based principle types is polymorphism.

**Subtyping.** As usual, subtyping is defined inductively so that types are a principle type is polymorphism.

**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 watermark resource usage, no matter how long we execute the program.

If $\Sigma; \Gamma, Q \models e : (A, Q')$ and $S, V, H \models e \Downarrow \circ \circ (p, p')$ 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 $\Sigma$ and $\Gamma$. Therefore, we adapt our definition of well-formed environments to annotated types. We simply replace the rule $V: \text{FUN}$ in Figure 4 with the following rule. Of course, $H \models V : \Gamma$ refers to the newly defined judgment.

$$H(\ell) = (\lambda x.e, V) \quad \exists \Gamma, Q, Q' : H \models V : \Gamma \land q \exists F : \text{FUN} : (S, V, H) \models \lambda x.e : (\Sigma \rightarrow B, F), Q', Q \quad H \models \ell \mapsto (\lambda x.e, V) : (\Sigma \rightarrow B, F)$$

In addition to the aforementioned soundness, the Theorem 2 states a stronger property for terminating evaluations. If an expression $e$ evaluates to a value $v$ in a well-formed environment then the difference between initial and final potential is an upper bound on the resource usage of the evaluation.

**Theorem 2 (Soundness).** Let $H \models V : \Gamma, H \models S : \Sigma$, and $\Sigma; \Gamma, Q \models e : (B, Q')$. Then $p - p' \leq \Phi_{S, V, H}(\Sigma; \Gamma, Q) - \Phi_{S, V, H}(B, Q')$, and $H \models e : B$.

1. If $S, V, H \models e \Downarrow \circ \circ (p, p')$ then $p \leq \Phi_{S, V, H}(\Sigma; \Gamma, Q)$.
2. If $S, V, H \models e \Downarrow \circ \circ (p, p')$ then $p \leq \Phi_{S, V, H}(\Sigma; \Gamma, Q)$.

Theorem 2 is proved by a nested induction on the derivation of the evaluation judgment and the type judgment $\Sigma; \Gamma, Q \models e : (B, Q')$. The inner induction on the type judgment is needed because of the structural rules. There is one proof for all possible instantiations of the resource constants. An sole induction on the type judgement fails because the size of the type derivation can increase in the case of the function application in which we retrieve a type derivation for the function body from the well-formed judgement as defined by the (updated) rule $V: \text{FUN}$.

The structure of the proof matches the structure of the previous soundness proofs for type systems based on AARA [31, 33, 41]. The induction case of many rules is similar to the induction cases of the corresponding rules for multivariate AARA for first-order programs [33] and linear AARA for higher-order programs [41]. For one thing, additional complexity is introduced by the new resource polynomials for user-defined data types. We designed the system so that this additional complexity is dealt with locally in the rules $\text{A:MAT}$, $\text{A:CONS}$, and $\text{A:SHARE}$. The soundness of these rules follows directly from an application of Lemma 2 and Lemma 3, respectively. As in previous work [34] the well-formed judgement that captures type derivations enables us to treat function abstraction and application in a very similar fashion as in the first-order case [33]. The coinductive definition of the well-formedness judgement does not cause any difficulties. A major novel aspect in the proof is the typed argument stack $S : \Sigma$ that also carries potential. Surprisingly, this typed stack is simply treated like a typed environment $V : \Gamma$ in the proof. It is already incorporated in the shift and share operations (Lemma 2 and Lemma 3).

We deal with the mutable heap by requiring that array elements do not influence the potential of an array. As a result, we can prove the following lemma, which is used in the proof of Theorem 2.

**Lemma 4.** If $H \models V : \Gamma, H \models S : \Sigma, \Sigma; \Gamma, Q \models e : (B, Q')$ and stack, $V, H \models e \Downarrow \circ \circ (\ell, H') \circ (p, p')$ then $\Phi_{S, V, H}(\Gamma, Q) = \Phi_{S, V, H}(\Gamma, Q)$. 

Draft 10 2013/7/27
7. Implementation and Bound Inference

Figure 8 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 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 forms that are introduced 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 heat cells, evaluation steps, and ticks. The letter 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 actual bound inference works similarly as in previous AARA systems [32, 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 fantastic 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 express that minimization of higher-degree annotations should take priority.

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 [33]. 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 return type. The annotations here are symbolic and the actual number are yet to be determined by the LP solver. Function annotations have the side effect of sending constraints to the LP solver. It would be possible to keep a constraint set for the respective function in memory and to send a copy with fresh variables to the LP solver at every call. However, it is more efficient to lazily trigger the constraint generation from the function body at every call site when the function is provided with a return annotation.

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

For the most part, our constraints have the form of a so-called network (or network-flow) problem [49]. LP solvers can handle network problems very efficiently and in practice CLP solves the constraints RAML generates in linear time. Because our problem sizes are large, we can save memory and time by reducing the number of constraints that are generated during typing. A representative example of an optimization is that we try to reuse constraint names instead of producing constraints like p = q.

RAML provides two ways of analyzing a program. In main mode RAML derives a bound for evaluation cost of the main expression of the program, that is, the last expression in the top-level list of let bindings. In module mode, RAML derives a bound for every top-level let binding that has a function type.

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. 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.

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

```ocaml
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 · 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.

\[
\text{let geq sid1 sid2 cour_ids =}
\text{avg_grade sid1 cour_ids >= avg_grade sid2 cour_ids}
\]

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.

\[
\text{let rec partition gt acc l =}
\text{match l with}
\text{| [] -> let (cs,bs,_) = acc in (cs,bs)}
\text{| x::xs -> let (cs,bs,aux) = acc in}
\text{let acc' = if gt x aux then (cs, xs::bs, aux) else (cs::xs,bs,aux)}
\text{in partition gt acc' xs}
\]

\[
\text{let rec qsort gt aux l = match l with | [] -> []}
\text{| x::xs ->}
\text{let ys,zs = partition (gt x) (\[
\text{| [],aux} \] xs in}
\text{append (qsort gt aux ys) (x::(qsort gt aux zs))}
\]

\[
\text{let sort_students s_ids c_ids = qsort geq c_ids s_ids}
\]

In 0.31s RAML computes the tight bound \(n^2m - nm\) for \(\text{sort\_students}\) 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 \(\text{db\_query}\) with the function \(\text{lookup}\).

\[
\text{let lookup sid cid table =}
\text{let cid_map = find (fun id -> id = sid) table in}
\text{find (fun id -> id = cid) cid_map}
\]

For the resulting sorting function, RAML computes the tight bound \(nm\) in 0.87s.

9. 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 [47, 52], to object-oriented programs [35, 38], and to low-level code by integrating it with separation logic [8]. All the aforementioned amortized-analysis–based systems are limited to linear bounds. Hofmann et al. [29, 32, 33] presented a multivariance 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 curved 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. [50, 51] 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. Cray and Weirich [20] presented a type system for specifying and certifying resource consumption. Danielsson [22] developed a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of functional programs. 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 [53] 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.

Benzenzer [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. Dannen et al. [23, 24] propose an interesting technique to derive higher-order recurrence relations from higher-order functional 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.

Abstract interpretation based approaches to resource analysis [12, 18, 27, 28, 45] focus on first-order integer programs with loops. Cicek et al. [19] study a type system for incremental complexity.

In an active area of research, techniques from term rewriting are applied to complexity analysis [9, 15, 45]; sometimes in combination with amortized analysis [36]. These techniques are usually restricted to first-order programs and time complexity. Recently, 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.

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.

10. Conclusion

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

As the title of this article indicates, there are many open problems left on the way to a usable resource analysis system for OCaml. In the future, we plan to improve the bound analysis for programs with side-effects and exceptions. We will also work on mechanisms that allow user interaction for manually deriving bounds if the automation fails. Furthermore, 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 now have the tools to further push the boundaries of practical quantitative software verification.
References


