Typable Fragments of Polynomial Automatic Amortized Resource Analysis

Long Pham
Carnegie Mellon University, Pittsburgh, PA, USA
longp@andrew.cmu.edu

Jan Hoffmann
Carnegie Mellon University, Pittsburgh, PA, USA
janh@andrew.cmu.edu

Abstract

Being a fully automated technique for resource analysis, automatic amortized resource analysis (AARA) can fail in returning worst-case cost bounds of programs, fundamentally due to the undecidability of resource analysis. For programmers who are unfamiliar with the technical details of AARA, it is difficult to predict whether a program can be successfully analyzed in AARA. Motivated by this problem, this article identifies classes of programs that can be analyzed in type-based polynomial AARA. Firstly, it is shown that the set of functions that are typable in univariate polynomial AARA coincides with the complexity class PTime. Secondly, the article presents a sufficient condition for typability that axiomatically requires every sub-expression of a given program to be polynomial-time. It is proved that this condition implies typability in multivariate polynomial AARA under some syntactic restrictions.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Resource consumption, Quantitative analysis, Amortized analysis, Typability

Digital Object Identifier 10.4230/LIPIcs.CSL.2021.34

Related Version A full version of the paper is available at `https://arxiv.org/abs/2010.16353` [30].

Funding This article is based on research supported by DARPA under AA Contract FA8750-18-C-0092 and by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514, and SHF Awards 1812876 and 2007784. 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. Long Pham gratefully acknowledges the support of the Funai Overseas Scholarship by the Funai Foundation of Information Technology, Japan.

1 Introduction

There exists a wide range of effective techniques for automatically or semi-automatically analyzing the resource consumption of programs. These techniques derive symbolic bounds on the worst-case [24], best-case [10, 28], or expected [7, 29] resource consumption and are based on type systems [8, 33, 9, 26, 2, 6, 12], recurrence relations [34, 11, 1, 25, 23], relational reasoning [6, 31], and term rewriting [3, 5, 19].

State-of-the-art resource analyses can automatically derive complex bounds for large programs, and making analyses more practical by improving their efficiency and range is a main driving force in this area. However, resource analysis for Turing-complete languages is undecidable, and even for the most sophisticated tools there will remain programs that cannot be analyzed automatically. Diagnosing the cause and modifying the program so that the analysis can derive a bound often require in-depth knowledge of the implemented techniques. As a result, the usability of more sophisticated analysis tools is hampered by their complexity.

To improve the usability of automatic resource analysis for non-experts, this article develops easy-to-understand characterizations of the programs that can be analyzed with automatic amortized resource analysis (AARA). Such characterizations can serve as explanations for an unsuccessful resource analysis and guide program development without revealing technical details of the underlying analysis.

AARA is a type-based analysis that is based on the potential method of amortized analysis. It has been first introduced by Hofmann and Jost [18] for deriving linear heap-space bounds for a first-order language with lists. AARA has subsequently been extended to univariate polynomial bounds [16], multivariate polynomial bounds [13, 14], and exponential bounds [22]. Furthermore, AARA has been extended to other language features such as higher-order and polymorphic functions [20, 15], lazy evaluation [21], and probabilistic programming [29]. The analysis has been implemented in the programming language Resource-Aware ML (RaML) [15]. An overview of polynomial AARA can be found in Section 2. We are not aware of previous work that studies the characterization of typable fragments of AARA.

Our first contribution (Section 3) is a characterization of the (mathematical) functions that can be implemented in AARA. We demonstrate that it is possible to embed every polynomial-time Turing machine in AARA. That is, for every such Turing machine, there exists an equivalent polynomial-time program that is typable in polynomial AARA. This result shows that polynomial AARA corresponds to the complexity class PTime and is in the tradition of implicit computational complexity (ICC) [4, 27, 17], which studies linguistic characterizations of complexity classes. For a user of RaML, this result means that an implementation of a PTime function can always be rewritten so that a worst-case cost bound can be automatically derived. However, it does not provide guidance on how to rewrite an implementation.

An ideal resource analysis should automatically derive a cost bound for every program that has a polynomial bound. However, such an analysis does not exist, because the problem of deciding whether a given program runs in polynomial time is undecidable [13]. Moreover, AARA is a type-based analysis that derives the bound of an expression from its sub-expressions. So we can only expect to derive a bound for an expression which is *inherently polynomial time*, that is, every subexpression is in PTime if viewed as a function.

Our second contribution is an axiomatic definition of inherently polynomial time that implies typability in multivariate polynomial AARA for a Turing-complete first-order language with lists (Section 2) under some restrictions: Programs can only use primitive recursion instead of general recursion, some variables are affine, and the use of nested lists is restricted. Although this characterization is far from being a necessary condition, we believe that it can be a valuable guide to users. A key concept is the notion of *uniform resource annotations* which is essential in the proof that inherently polynomial time is a sufficient condition for typability in multivariate polynomial AARA.

2 Automatic Amortized Resource Analysis (AARA)

Among approaches to resource analysis is AARA. Given a program P, consider its history of execution, that is, a sequence of transitioning program states. As in Sleator and Tarjan's potential method in amortized analysis [32], we assign a certain (non-negative) amount of potential to the initial state of this sequence. If we can ensure that (i) the amount of potential never becomes negative throughout P's run and (ii) the actual computational cost in each transition of P is less than or equal to the change in the amount of potential, then we know that the total resource usage of P is bounded above by the initial potential. This is essentially how AARA works.

More concretely, each sub-expression of P is assigned a *resource-annotated type*: a conventional (i.e. simple) type augmented with an expression that indicates how much potential is stored. In polynomial AARA [16, 14], we use polynomial functions to express potential. Initially, AARA only assigns templates of resource-annotated types where coefficients of polynomials are left blank. AARA then collects constraints on these coefficients that respect the cost semantics of P. Finally, as these constraints are all linear, we can simply solve them using an off-the-shelf liner program solver, thereby inferring resource-annotated types. A worst-case cost bound of P can be extracted from its resource-annotated type.

2.1 Resource-Aware ML

Resource-Aware ML (RaML) is a Turing-complete functional programming language used in the study of AARA [16].

The original version of RaML is first-order (i.e. no higher-order types or functions appear in RaML) and only offers a relatively small set of language features. Subsequent versions of RaML support more language features such as higher-order functions and polymorphic functions [15]. In this section, we describe a variant of RaML that only differs from the original version in a few minor details; e.g. the tick construct and the support for sum types.

The base types (denoted by b) and simple types (denoted by τ) of RaML are formed by

$b ::= \mathbf{1}$	unit type		$\tau ::= b$		base type
$b_1 + b_2$	sum type		$b_1 \to b_2$		arrow type
$b_1 \times b_2$	product type				
$L(b)$	list type.				

The set of all base types will be denoted by $\mathbb{B}$.

Fix a set $\mathcal{V} = \{x, y, x_1, x_2, \ldots\}$ of variable symbols and a set $\mathcal{F} = \{f, \ldots\}$ of function symbols. The grammar of RaML is

$e ::= x$	variable
$\mid \langle\rangle$	unit element
$\mid \ell \cdot x \mid r \cdot x \mid \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\}$	sum constructors and destructor
$\mid \langle x_1, x_2\rangle \mid \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\}$	pair constructor and destructor
$\mid [\,] \mid x_1 :: x_2 \mid \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\}$	list constructors and destructor
$\mid \mathsf{fun}\ f\ x = e$	function definition
$\mid f\ x$	function application
$\mid \mathsf{tick}\ q$	resource consumption; $q \in \mathbb{Q}$
$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let-binding
$\mid \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e$	variable sharing.

In a function definition, e is allowed to mention f. Therefore, we can implement not only primitive recursion but also general recursion. As standard, we use the let-normal form, where we only permit function application of the form $x_1\ x_2$ as opposed to $e_1\ e_2$. For convenience in resource analysis, we require each variable symbol to be used in a affine manner (i.e. can only be used at most once). To use a variable symbol multiple times, we duplicate the symbol with the share construct.

In the interest space, we will not present a type system of this language here. It is available in Appendix B.1 of the full version of this article [30].

RaML programs are evaluated using the call-by-value strategy. Computational costs accrue only when tick q is executed, and this cost metric is known as the tick metric. The general cost semantics of RaML can be found in [16]. In the case of the running time, which is a specific cost metric, of RaML, the judgment of the cost semantics has the form

$$V \vdash e \Downarrow v \mid n,$$

where V is an environment (i.e. a set of pairs of variable symbols and semantic values), v is a semantic value, and $n \in \mathbb{N}$ is the running time of evaluating program e to v. The running time is formally defined in Appendix B.2 of [30].

2.2 Univariate AARA

In univariate AARA, each list is annotated with a polynomial indicating the amount of the potential stored in the list. Univariate AARA does not let us mix potential of two lists, that is, multiply polynomials of two lists' potential. This is why univariate AARA is called *univariate*.

Resource-annotated base types (denoted by b) and resource-annotated simple types (denoted by τ) are formed by the following grammar:

$$\begin{array}{lll} b ::= \mathbf{1} & \text{unit type} & B ::= \langle b, q\rangle \quad q \in \mathbb{Q}_{\geq 0} \\ \quad b_1 + b_2 & \text{sum type} & \tau ::= b \quad \text{base type} \\ \quad b_1 \times b_2 & \text{product type} & \quad B_1 \to B_2 \quad \text{arrow type} \\ \quad L^{\vec{q}}(b) & \text{list type.} & \end{array}$$

Here, $\vec{q}$ is a finite vector of $\mathbb{Q}_{\geq 0}$.

Given a semantic value $v : b$, where b is a resource-annotated base type, the potential stored in v is inductively defined as

$$\begin{array}{ll} \Phi(v : \mathbf{1}) := 0 & \Phi([\,] : L^{\vec{q}}(b)) := 0 \\ \Phi(\ell \cdot v : b_1 + b_2) := \Phi(v : b_1) & \Phi(v_1 :: v_2 : L^{\vec{q}}(b)) := \Phi(v_1 : b) + \phi(|v_1 :: v_2|, \vec{q}) \\ \Phi(r \cdot v : b_1 + b_2) := \Phi(v : b_2), & \end{array}$$

where $|\cdot|$ denotes the length of an input list. Given $n \in \mathbb{N}$ and $\vec{q} = (q_1, \ldots, q_k)$, $\phi(n, \vec{q})$ is defined as $\phi(n, \vec{q}) := \sum_{i=1}^{k} q_i \binom{n}{i}$. If $n < i$, then $\binom{n}{i} = 0$.

The typing judgment of univariate AARA has the form

$$\Gamma_{\text{anno}}; p \vdash e : B,$$

where Γ_{anno} is a resource-annotated typing context and $p \in \mathbb{Q}_{\geq 0}$. We sometimes write $\Sigma_{\text{anno}}; \Gamma_{\text{anno}}; p \vdash e : B$, where a resource-annotated typing context is split into Σ_{anno} for arrow-type variables and Γ_{anno} for base-type variables. The type system of univariate AARA is available in Appendix C.1 of [30].

To give examples of judgments in univariate AARA, consider two programs: (i) *append* that appends the first input list to the second, and (ii) *quicksort* that performs quicksort. The running time of *append* is proportional to the size of the first input, and the running time of *quicksort* is bounded by the square of the input size. For simplicity, we will not work out the exact coefficients of polynomial bounds. Instead, we simply assume that the running time of *append* is bounded by the function $n, m \mapsto n$, where n and m are the lengths of the two input lists. Likewise, we assume that the running time of *quicksort* is bounded by

$n \mapsto n^2$, respectively. It then makes sense that these two programs can be typed in univariate AARA as

$$\mathit{append} : \langle\langle L^1(b), L^0(b)\rangle, 0\rangle \to \langle L^0(b), 0\rangle \qquad \mathit{quicksort} : \langle L^{(1,2)}(b), 0\rangle \to \langle L^0(b), 0\rangle.$$

The univariate resource annotation $(1, 2)$ of *quicksort* represents polynomial $n \mapsto 1 \cdot \binom{n}{1} + 2 \cdot \binom{n}{2} = n^2$. The implementations of *append* and *quicksort* are given in Appendix C.3 of [30].

Univariate AARA is sound with respect to the cost semantics (specifically, the running time) of RaML:

▶ **Theorem 1** (Soundness of univariate AARA [16]). *Given term e, suppose $\Gamma_{anno}; p \vdash e : \langle b_{anno}, q\rangle$ is derived in univariate AARA. Let V be an environment such that $V \vdash e \Downarrow v \mid n$; that is, e runs in n units of time under V. We then have*

$$n \le p + \Phi(V : \Gamma_{anno}) - q - \Phi(v : b_{anno}),$$

where $\Phi(V : \Gamma_{anno}) = \sum_{x \in dom(\Gamma_{anno})} \Phi(V(x) : \Gamma_{anno}(x))$.

2.3 Multivariate AARA

In contrast to univariate AARA, multivariate AARA allows us to mix potential of different lists. For example, we can have $|\ell_1| \cdot |\ell_2|$'s worth of potential, where $|\cdot|$ denotes the length of a list, in multivariate AARA. Due to this multivariate nature, multivariate AARA has a single global resource annotation represented by a multivariate polynomial over all size variables occurring in a given term. This global resource annotation is separate from individual types in a typing context.

Multivariate AARA is strictly more expressive than univariate one. This is surprising in light of the fact that multivariate polynomials can always be bounded by univariate polynomials; e.g. xy is bounded by $x^2 + y^2$. Examples of programs that cannot be typed in univariate AARA but are typable in multivariate AARA are in Section 4.1 and Section 5.

Resource-Annotated Types

Resource-annotated types in multivariate AARA are formed by

$b ::= \mathbf{1}$		unit type	$B ::= \langle b, Q\rangle$		
	$b_1 + b_2$	sum type	$\tau ::= b$		base type
	$b_1 \times b_2$	product type		$B_1 \to B_2$	arrow type
	$L(b)$	list type.			

In $\langle b, Q\rangle$, Q is a multivariate resource annotation over the size variables inside b. This will be formalized shortly.

Given a base type $b \in \mathbb{B}$, its base polynomial is a function of type $[\![b]\!] \to \mathbb{N}$, where $[\![b]\!]$ is the set of semantic values of type b. The set of base polynomials associated with type b, denoted by $\mathcal{B}(b)$, is inductively defined as follow:

$$\begin{aligned}
\mathcal{B}(\mathbf{1}) &:= \{\lambda v.1\} \\
\mathcal{B}(b_1 + b_2) &:= \{\lambda(\ell \cdot v).p(v) \mid p \in \mathcal{B}(b_1)\} \cup \{\lambda(r \cdot v).p(v) \mid p \in \mathcal{B}(b_2)\} \\
\mathcal{B}(b_1 \times b_2) &:= \{\lambda\langle v_1, v_2\rangle.p_1(v_1) \cdot p_2(v_2) \mid p_i \in \mathcal{B}(b_i)\} \\
\mathcal{B}(L(b)) &:= \{\lambda[v_1, \ldots, v_n]. \sum_{1 \le j_1 < \cdots < j_k \le n} \prod_{1 \le i \le k} p_i(v_{j_i}) \mid k \in \mathbb{N}, p_i \in \mathcal{B}(b)\}.
\end{aligned}$$

For $b_1 + b_2$, we have a set of base polynomials for the ℓ-tag and another set for the r-tag. If a base polynomial is applied to a value with a wrong tag, we assume that the output is 0. For instance, if we feed a value $\ell \cdot \langle\,\rangle$ to $\lambda(r \cdot v).1$, the output should be 0. In the definition of $\mathcal{B}(L(b))$, if $n < k$, the function should return 0 since it is the identity of summation.

Given base type b, a resource polynomial $p : [\![b]\!] \to \mathbb{Q}_{\geq 0}$ is a non-negative linear combination of finitely many base polynomials from $\mathcal{B}(b)$. It is straightforward to prove that $\mathcal{B}(b)$ for any b contains $\lambda v.1$. Therefore, a resource polynomial is always capable of expressing constant potential.

For convenience, it is desirable to have a succinct notation for base polynomials. This is achieved by introducing indexes of base polynomials:

$$\begin{aligned}
\mathcal{I}(\mathbf{1}) &:= \{*\} \\
\mathcal{I}(b_1 + b_2) &:= \{\ell \cdot i \mid i \in \mathcal{I}(b_1)\} \cup \{r \cdot i \mid i \in \mathcal{I}(b_2)\} \\
\mathcal{I}(b_1 \times b_2) &:= \{\langle i_1, i_2\rangle \mid i_1 \in \mathcal{I}(b_1), i_2 \in \mathcal{I}(b_2)\} \\
\mathcal{I}(L(b)) &:= \{[i_1, \ldots, i_k] \mid k \in \mathbb{N}, i_j \in \mathcal{I}(b)\}.
\end{aligned}$$

An index is usually used as a subscript for a (meta)-variable representing a coefficient of a base polynomial. For instance, $q_{\langle *,*\rangle} \in \mathbb{Q}_{\geq 0}$ is a meta-variable representing a coefficient of base polynomial $\lambda\langle v_1, v_2\rangle.1$. For any base type b, we will write 0_b for the index $\lambda v.1$.

For example, consider $\mathcal{I}(L(\mathbf{1})) = \{*, [*], [*,*], [*,*,*], \ldots\}$. The index $[*,*]$ represents the polynomial function

$$\begin{aligned}
\lambda[v_1, \ldots, v_n].\sum_{1 \leq j_1 < j_2 \leq n} \prod_{1 \leq i \leq 2} ((\lambda v.1)\ v_{j_i}) &= \lambda[v_1, \ldots, v_n].\sum_{1 \leq j_1 < j_2 \leq n} 1 \\
&= \lambda[v_1, \ldots, v_n].\binom{n}{2}.
\end{aligned}$$

Thus, the multivariate index $[*,*]$ represents a quadratic function on the input list's length.

The degree of an index is defined by

$$\begin{aligned}
\mathsf{deg}(*) &:= 0 & \mathsf{deg}(\langle i_1, i_2\rangle) &:= \mathsf{deg}(i_1) + \mathsf{deg}(i_2) \\
\mathsf{deg}(\ell \cdot i), \mathsf{deg}(r \cdot i) &:= \mathsf{deg}(i) & \mathsf{deg}([i_1, \ldots, i_k]) &:= k + \sum_{1 \leq j \leq k} \mathsf{deg}(i_j).
\end{aligned}$$

Intuitively, $\mathsf{deg}(i)$ is equal to the degree of the polynomial function that index i represents. Because a resource polynomial can only have non-zero coefficients for finitely many base polynomials, any resource polynomial (or a finite set of resource polynomials) has a bounded degree. In practice, we ask a user of AARA to supply an upper bound on the degree of base polynomials.

Resource Annotations of Typing Contexts

Given a base-type typing context $\Gamma = \{x_1 : b_1, \ldots, x_n : b_n\}$, its multivariate resource annotation is given by a resource polynomial of type $b_1 \times \cdots \times b_n$. In other words, we treat a typing context as one big tuple and assign a single multivariate annotation to this tuple.

With regard to an arrow-type typing context $\Sigma = \{f_1 : b_{1,1} \to b_{1,2}, \ldots, f_m : b_{m,1} \to b_{m,2}\}$, its multivariate resource annotation has the form

$$\Sigma_{\text{anno}} = \{f_1 : B_{1,1} \to B_{1,2}, \ldots, f_m : B_{m,1} \to B_{m,2}\},$$

where each $B_{i,j}$ is a pair $\langle b_{i,j}, Q\rangle$ such that Q is a multivariate resource annotation of $b_{i,j}$.

Typing Judgment

The typing judgment of multivariate AARA takes the form

$$\Gamma; P \vdash e : \langle b, Q\rangle,$$

where Γ and b are free of resource annotations. P and Q are multivariate annotation over Γ and b, respectively. The type system of multivariate AARA is available in Appendix D.2 of [30].

To give examples of judgments in multivariate AARA, consider *append* $\langle \ell_1, \ell_2\rangle$, which appends ℓ_1 to ℓ_2. Suppose that the output must store $n \mapsto n^2$ much potential, where n is the output's length. It is reasonable that the total potential required for this program is $|\ell_1| + (|\ell_1| + |\ell_2|)^2$, out of which $|\ell_1|$ is used to account for the running time. This can be expressed by the judgment $\ell_1 : L(\mathbf{1}), \ell_2 : L(\mathbf{1}); P \vdash \mathit{append}\ \langle \ell_1, \ell_2\rangle : \langle L(\mathbf{1}), Q\rangle$, where the positive coefficients of P and Q are

$$P(\langle [*], *\rangle) = P(\langle [*, *], *\rangle) = P(\langle [*], [*]\rangle) = P(\langle *, [*, *]\rangle) = 2 \qquad P(\langle *, [*]\rangle) = 1$$
$$Q([*, *]) = 2 \qquad Q([*]) = 1.$$

P amounts to $2 \cdot \left(\binom{|\ell_1|}{1} + \binom{|\ell_1|}{2} + \binom{|\ell_1|}{1} \cdot \binom{|\ell_2|}{1} + \binom{|\ell_2|}{2}\right) + 1 \cdot \binom{|\ell_2|}{1}$, which is equal to $|\ell_1| + (|\ell_1| + |\ell_2|)^2$ as desired. Similarly, Q amounts to $2 \cdot \binom{n}{2} + 1 \cdot \binom{n}{1} = n^2$ as desired, where $n = |\ell_1| + |\ell_2|$.

The multivariate equivalent of the soundness theorem (Theorem 1) holds [14].

3 Embedding Polynomial-Time Turing Machines in AARA

In this section, we show that every polynomial-time Turing machine can be expressed as a typable RaML program while preserving the semantics and worst-case cost bounds. More formally, we have

▶ **Theorem 2** (Embedding of polynomial-time Turing machines in RaML). *Let M be a polynomial-time Turing machine that inputs and outputs bit strings from $\{0,1\}^*$. There exists a RaML program $M' : \{0,1\}^* \to \{0,1\}^*$ such that*

- *For every input $w \in \{0,1\}^*$, we have $M(w) = M'(w)$;*
- *The computational cost of M' (according to the tick metric) is larger than or equal to the running time of M;*
- *Univariate AARA can infer a polynomial upper bound of the computational cost of M'.*

Theorem 2 only tells us the existence of a RaML program M' that is typable in univariate AARA and that simulates M faithfully. In our proof of the theorem, we assume that a polynomial bound on the running time of M is known. Thus, if we do not have access to this polynomial bound, we cannot construct M'. In fact, the problem of determining whether a given Turing machine runs in polynomial time or not is undecidable [13].

It is fairly easy to prove that the cost of any program according to the tick metric is asymptotically bounded by its running time. Therefore, in the statement of Theorem 2, we can replace the "tick metric" with the "running time" of RaML.

A detailed proof of Theorem 2 is available in Appendix A of [30].

3.1 Preliminaries

▶ **Definition 3** (Turing machine). *A (deterministic) Turing machine M is specified by an 8-tuple $(Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, q_0, q_{final})$, where*

- *Q is a finite set of machine states.*

- Σ *is a finite input alphabet.* Γ *is a finite alphabet for symbols written on* M*'s tape. Since an input will be initially placed on the tape, every input symbol is also a tape symbol.*
- $\vdash \in \Gamma \setminus \Sigma$ *is the left end marker that demarcates the left end of a semi-infinite working tape, and* $\sqcup \in \Gamma \setminus \Sigma$ *is the blank symbol for the tape.*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the transition function.*
- $q_0 \in Q$ *is the initial state, and* $q_{final} \in Q$ *is the final state.*

In the initial configuration of a Turing machine, an input string w is placed immediately after the left end marker $\vdash$ on the tape. The state of the machine is initially q_0, and the read/write head is positioned on the first symbol of w. The rest of the tape is filled with $\sqcup$.

The Turing machine first (i) reads the content of the cell currently under the tape head and (ii) identifies the current state of the machine. The machine then overwrites the current cell (if necessary), updates the machine's state, and moves the head to the left or right according to the transition function δ. The machine terminates as soon as it enters q_{final}. Upon termination, the content of the tape before the first blank symbol is considered as the machine's output. The running time is defined as the number of steps the Turing machine makes before termination.

Without loss of generality, we will henceforth only consider Turing machines with $\Sigma = \{0, 1\}$ and $\Gamma = \Sigma \cup \{\vdash, \sqcup\}$.

To enhance clarity, we will introduce two type aliases, State and Sym, which are defined as $L(\mathbf{1} + \mathbf{1})$; i.e. bit strings or natural numbers. The type State represents machine states of M, and Sym represents tape symbols of M. In fact, because M has finitely many machine states and tape symbols, State and Sym can alternatively be encoded as $\mathbf{1} + \cdots + \mathbf{1}$.

3.2 Embedding

Fix a polynomial-time Turing machine $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, q_0, q_{\text{final}})$. Assume that the running time of M is bounded above by $p(n)$ for some polynomial $p : \mathbb{N} \to \mathbb{N}$. The target program of the translation will be denoted by M', and this is what we are about to define. M' works as described in Algorithm 1. A RaML implementation of M' is available in Appendix A.3 of [30].

Algorithm 1 Operational working of target RaML program M'.

Require: $w \in \{0, 1\}^*$
1: **procedure** $M'(w)$
2: Create a singleton list $\ell_1 : L(\text{Sym})$ containing $\vdash$
3: Create a list $\ell_2 : L(\text{Sym})$ of size $p(|w|)$ filled with $\sqcup$
4: Prepend ℓ_2 with w
5: Create a list $ps : L(\mathbf{1})$ of size $p(|w|)$ ▷ Reservoir of potential
6: $s \leftarrow q_0$ ▷ Initialize the current state
7: **while** $s \neq q_{\text{final}} \wedge ps \neq [\,]$ **do**
8: $ps \leftarrow$ tail ps ▷ Potential is released
9: Compute $\delta(s, \ell_2[0])$
10: Update s and $\ell_2[0]$ appropriately
11: Update the tape head's position by moving the head of ℓ_1 or ℓ_2 to the other
12: **return** append(reverse ℓ_1, ℓ_2)

The list ℓ_1 represents the region on M's tape to the left of the tape head (in the reverse order and excluding the cell where the tape head is currently on), and ℓ_2 represents the region to the right of the head (including the current cell). Since it is assumed that $p(|w|)$,

where $|w|$ denotes the length of input list w, is an upper bound on M's running time, we are assured that M requires at most $p(|w|)$ many cells on the working tape. This is why ℓ_2 initially has size $p(|w|)$. In fact, because we prepend ℓ_2 with w in line 4, we have $|w|$ more cells than necessary.

The list ps acts as a reservoir of potential, storing constant potential in each element. As the head of ps is removed in line 8, the potential stored in this element is freed and will be consumed in subsequent lines inside the loop's body.

It is technically possible to store potential directly in ℓ_1 and ℓ_2, which together simulate M's working tape. However, not all cells on the working tape of M are accessed equally often – some cells are accessed more often than others, and the maximum number of accesses to a given cell may not be bounded by a constant. If we are to store potential in ℓ_1 and ℓ_2, each cell of ℓ_1 and ℓ_2 needs to store $p(n)$ units of potential at the beginning. As a result, the total amount of potential supplied to M' is $p^2(n)$, which is a gross over-approximation of the actual running time. Therefore, to have a tighter cost bound, a separate list, namely ps, is employed as a reservoir of potential.

4 Inherently Polynomial Time

Section 3 investigates the expressive power of AARA from the viewpoint of programming language semantics, disregarding the issue of how to algorithmically turn an arbitrary Turing machine into a typable RaML program. By contrast, in this section, we aim to identify a typable fragment of AARA that is defined statically/axiomatically. Henceforth, we will call the sufficient condition corresponding to the typable fragment that this section presents *inherently polynomial time*.

A key requirement is that the typable fragment should not resemble AARA's type system, which itself is also defined axiomatically. Otherwise, it would be trivial to prove that any term in this fragment is typable in AARA. Because we want users of AARA to benefit from our findings of the present work, another requirement is that the definition (or at least the informal definition) of inherently polynomial time should be easy to convey to users of AARA. On the other hand, it is not our priority to find as large a typable fragment as we can.

In the remaining of the article, we will focus on the running time as a cost metric of RaML, unless stated otherwise.

4.1 High-Level Design

By Theorem 1 (and its multivariate equivalent), AARA is sound: if a program is typable in AARA, its resource-annotated type is a correct upper bound on the running time. Hence, to be typable in AARA, the worst-case running time of a program must be polynomial. To ensure termination of programs, we first restrict recursion to primitive recursion.

Furthermore, the type system of AARA is compositional: if term e is typable, so is every sub-expression of e. Hence, in order for e to be typable, not only e but also all of its sub-expressions must be polynomial-time. This suggests that we should define the sufficient condition inductively, hence the name *inherently* polynomial time.

It is straightforward to determine whether each of the base cases of the inductive definition is typable or not. It remains to work out inductive cases in the inductively defined sufficient condition for typability. The most interesting case is primitive recursion. A primitive recursion will be written as

$$e := \mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\},$$

where x is matched against $y :: ys$ in the second branch, and z is the result of a recursive call. The stepping function e_1 can only contain y, ys, and z as free variables; i.e. $\mathtt{FV}(e_1) \subseteq \{y, ys, z\}$. From compositionality, we know that e_0 and e_1 are both typable and hence run in polynomial time. Under what condition does the entire e run in polynomial time as well?

To answer this question, we first observe the following. Without any restrictions on e_0 and e_1 apart from that they should be typable, AARA may project e's worst-case time complexity to be exponential even if the actual running time of e is polynomial. To illustrate this, consider

$$e := \mathsf{rec}\ x\ \{[\,] \hookrightarrow [\,] \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow \mathsf{share}\ z\ \mathsf{as}\ z_1, z_2\ \mathsf{in}\ \mathit{append}\ \langle z_1, z_2\rangle\}. \tag{4.1}$$

Although the actual running time of e is $O(|x|)$ and hence is linear, e is untypable in polynomial AARA. The problem of (4.1) is that the stepping function doubles the input size. This makes AARA conclude (naïvely) that the worst-case total running time is $O(2^{|x|})$, and this cost bound is beyond the expressive power of AARA (exponential AARA [22], however, can handle exponential cost bounds).

To preclude the example (4.1), it is reasonable to require the running time of e_1 (i.e. a stepping function inside primitive recursion) to be constant in the size of z (i.e. the result of a recursive call). More concretely, if $T(|y|, |ys|, |z|)$ is the running time of a stepping function, we demand $T(|y|, |ys|, |z|) \leq p(|y|, |ys|)$, where $p(|y|, |ys|)$ is a polynomial in $|y|$ and $|ys|$ (i.e. the sizes of y's and ys's semantic values[1]). We will adopt this idea in the formulation of inherently polynomial time.

Although this idea results in a fairly simple inductive definition of inherently polynomial time, a major drawback is that some realistic programs are not admitted by the current formulation of inherently polynomial time. For instance, consider *multiply* that, given input lists ℓ_1 and ℓ_2, produces a list of size $|\ell_1| \cdot |\ell_2|$:

$$\mathit{multiply} := \lambda\ell_1.\lambda\ell_2.\mathsf{rec}\ \ell_1\ \{[\,] \hookrightarrow \langle \ell_2, [\,]\rangle \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\}, \tag{4.2}$$

where the stepping function of primitive recursion is

$$e_1 \equiv \mathsf{case}\ z\ \{\langle z_1, z_2\rangle \hookrightarrow \mathsf{share}\ z_1\ \mathsf{as}\ z_{1,1}, z_{1,2}\ \mathsf{in}\ \langle z_{1,1}, \mathit{append}\ \langle z_{1,2}, z_2\rangle\rangle\}.$$

The first component of z stores ℓ_2, while the second component of z acts as an accumulator. The running time of e_1 is polynomial in $|z_1|$ but constant in $|z_2|$. Therefore, e_1's running time is only polynomial *partially* in $|z|$. This is why the overall time complexity of e remains polynomial instead of becoming exponential. Nonetheless, (4.2) is not inherently polynomial time according to the current formulation, since the formulation does not allow e_1's running time to have any dependence on $|z|$.

Furthermore, (4.2) can only be typed in multivariate AARA and not in univariate AARA. This means our formulation of inherently polynomial time fails to capture some of the realistic programs that are typable only in multivariate AARA. In view of this, one might wonder whether inherently polynomial time is completely encapsulated by univariate AARA; that is, every inherently polynomial-time RaML program is typable in univariate AARA. The answer is negative.

As a counterexample, consider the standard *append* defined as

$$\mathit{append} := \lambda\ell_1.\lambda\ell_2.\mathsf{rec}\ \ell_1\ \{[\,] \hookrightarrow \ell_2 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow y :: z\}. \tag{4.3}$$

[1] A formal definition of the size of RaML's base-type semantic values is not provided in this article. However, the idea is intuitive. For example, the size of a list is given by the sum of all elements' sizes.

Note that it is inherently polynomial time. *append* alone is typable in univariate AARA as well as multivariate AARA. However, if we require the output of *append* to carry quadratic potential (because it will be later fed to a function that demands quadratic potential from inputs, for example), then univariate AARA cannot type *append* – we need to resort to multivariate AARA to type it.

In summary, our formulation of inherently polynomial time goes beyond the remit of univariate AARA, but does not capture the full range of realistic programs that require multivariate potential.

4.2 Formulation of Inherently Polynomial Time

Restricting the Syntax of Resource-Aware ML

To ensure termination of programs, we require programs to use primitive recursion in place of general recursion. Hence, we will from now on work with a fragment of RaML wherein general recursion is replaced by primitive recursion. This fragment removes $\mathsf{fun}\ f\ x = e$ from the original RaML (Section 2.1) and adds the following:

1. $\lambda(x : b).e$ for a lambda abstraction, where $b \in \mathbb{B}$;
2. $\mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\}$, where z denotes the result of the recursive call.

In primitive recursion, e_1 is only allowed to mention $\{y, ys, z\}$. If e_1 needs access to a global variable v (i.e. a variable from outside the primitive recursion), v should be transferred to e_1 by placing v inside z.

The reason why we deny e_1 access to a global variable is that every variable symbol can only be accessed at most once in RaML. However, this is in fact already violated by e_1 having access to ys (because this means some elements of the input x are accessed multiple times during primitive recursion). Further, even if we let e_1 access global variables, AARA can be easily adapted. Also, it will result in a less strict formulation of inherently polynomial time that admits *multiply* in (4.2). Nonetheless, for simplicity, this article assumes that e_1 can only mention y, ys, and z.

Primitive recursion can be encoded using general recursion as

$$\mathsf{fun}\ f\ \langle x, \Gamma\rangle = \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid y :: ys \hookrightarrow \mathsf{share}\ ys\ \mathsf{as}\ ys_1, ys_2\ \mathsf{in}\ \mathsf{let}\ z = f\ \langle ys_1, \Gamma\rangle\ \mathsf{in}\ e_1\}.$$

Here, Γ is a set/sequence of those variables that do not appear in e_1, but e_0. Variable ys_1 is passed to the recursive call, and ys_2 is used in e_1 (if e_1 mentions ys).

Judgments

The primary judgment of inherently polynomial time is

$$\Delta; \Gamma \vdash e\ \mathsf{inhpoly}(V), \tag{4.4}$$

where

- Γ is a typing context containing both base-type and arrow-type variables such that $\Gamma \vdash e : b$ for base type b.
- $V \subseteq \mathrm{dom}(\Gamma)$ is a set of variables.
- Δ is a set of $f\ \mathsf{time}$, where $f \in \mathrm{dom}(\Gamma)$ is an arrow-type variable and $\mathsf{time} \in \{\mathsf{const}, \mathsf{poly}\}$.

Sometimes we split Γ into Σ for arrow-type variables and Γ for base-type variables, writing the judgment as $\Delta; \Sigma; \Gamma \vdash e\ \mathsf{inhpoly}(V)$. (4.4) is only applicable to base-type expressions e.

An informal interpretation of (4.4) is

- f const denotes that the running time of f is constant with respect to the input size, and likewise, f poly denotes that f's running time is polynomial[2] in the input size.
- The running time of e is (i) polynomial[3] in the sizes of those variables in V but (ii) constant in the sizes of $\mathrm{dom}(\Gamma) \setminus V$.
- Every sub-expression of e runs in polynomial time.

The judgments for an arrow-type expression e are

$$\Delta; \Gamma \vdash e \ \mathsf{const} \qquad \Delta; \Gamma \vdash e \ \mathsf{poly}, \tag{4.5}$$

$\Delta; \Gamma \vdash e$ const means e runs in constant time with respect to the input size, and $\Delta; \Gamma \vdash e$ poly likewise means e's running time is polynomial in the input size.

Inference Rules

The most important inference rules defining (4.4) are displayed in Figure 1. Throughout these rules, b denotes a base type, time is drawn from $\{\mathsf{const}, \mathsf{poly}\}$, and V is a set of variables. The remaining rules are deferred to Figure 10 in Appendix E of [30].

In (IP:Case-Sum), the notation $V[x \mapsto y]$ refers to the result of replacing x in V with y (if $x \in V$); otherwise, V remains intact. If the running time of $\mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\}$ in the rule's conclusion is allowed to be polynomial in $|x|$ (i.e. $x \in V$), then $e_{i \in \{\ell, r\}}$ in the two premises is allowed to run in polynomial time in $|y| = |x| - 1$.

Similarly, in (IP:Case-Prod), $V[x \mapsto x_1, x_2]$ means $(V \setminus \{x\}) \cup \{x_1, x_2\}$ if $x \in V$; otherwise, V remains unchanged.

(IP:Rec) is the crux of the notion of inherently polynomial time. Observe that the stepping function e_1 must be constant-time in $|z|$ (i.e. the size of z's semantic value).

In (IP:Let-Base), we use a finer-grained notation where the typing context of e_1 is split into Σ_1 for arrow-type variables and Γ_1 for base-type variables. V_3 is determined by

$$V_3 := \begin{cases} \mathrm{dom}(\Gamma_1) \cup (V_2 \setminus \{x\}) & \text{if } x \in V_2; \\ V_1 \cup V_2 & \text{otherwise.} \end{cases}$$

If $x \in V_2$, it means that e_2 runs in polynomial time in $|x|$. In the worst case, not only the running time of e_1 but $|e_1|$ (i.e. the output size of e_1) is polynomial in the sizes of those variables in V_1. Hence, in the worst case, the overall running time of $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is polynomial in $\mathrm{dom}(\Gamma_1)$, which contains all base-type variables appearing in e_1, and $V_2 \setminus \{x\}$. Note that (IP:Let-Base) considers the worst case – if we had information about the output size, we might be able to derive a more precise judgment.

Finally, the judgment (4.5) is defined by the following inference rules:

$$\frac{\Delta; x : b \vdash e\ \mathsf{inhpoly}(\emptyset)}{\Delta; \cdot \vdash \lambda(x : b).e\ \mathsf{const}}\ (\text{IP:Const}) \qquad \frac{\Delta; x : b \vdash e\ \mathsf{inhpoly}(\{x\})}{\Delta; \cdot \vdash \lambda(x : b).e\ \mathsf{poly}}\ (\text{IP:Poly})$$

In (IP:Const), because the conclusion indicates that the λ-abstraction's running time is constant in the input size, the premise states that the running time of the body e can only be polynomial in $\mathrm{dom}(\Gamma)$, which excludes x. By contrast, in the premise of (IP:Poly), the set of variables contains x.

---

[2] f's running time being polynomial does NOT mean that it is *strictly* polynomial – it can also be constant in the input size.

[3] Again, the running time of e may be constant as well as polynomial in the size of any $v \in V$.

$$\frac{}{\cdot; x : b \vdash x \ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:Base}) \qquad \frac{\Delta = \{f\ \mathsf{time}\}}{\Delta; f : b_1 \to b_2 \vdash f\ \mathsf{time}}\ (\text{IP:Arrow})$$

$$\frac{\cdot; x : b \vdash x\ \mathsf{inhpoly}(\emptyset)}{\cdot; x : b \vdash \ell \cdot x\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:SumL}) \qquad \frac{\cdot; x : b \vdash x\ \mathsf{inhpoly}(\emptyset)}{\cdot; x : b \vdash r \cdot x\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:SumR})$$

$$\frac{\cdot; x_1 : b_1 \vdash x_1\ \mathsf{inhpoly}(\emptyset) \qquad \cdot; x_2 : b_2 \vdash x_2\ \mathsf{inhpoly}(\emptyset)}{\cdot; x_1 : b_1, x_2 : b_2 \vdash \langle x_1, x_2\rangle\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:Pair})$$

$$\frac{}{\cdot; \cdot \vdash \langle\,\rangle\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:Unit}) \qquad \frac{\Delta = \{x_1\ \mathsf{const}\}}{\Delta; x_1 : b_1 \to b_2, x_2 : b_1 \vdash x_1\ x_2\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:App-Const})$$

$$\frac{}{\cdot; \cdot \vdash [\,]\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:Nil}) \qquad \frac{\Delta = \{x_1\ \mathsf{poly}\}}{\Delta; x_1 : b_1 \to b_2, x_2 : b_1 \vdash x_1\ x_2\ \mathsf{inhpoly}(\{x_2\})}\ (\text{IP:App-Poly})$$

$$\frac{\cdot; x_1 : b \vdash x_1\ \mathsf{inhpoly}(\emptyset) \qquad \cdot; x_2 : L(b) \vdash x_2\ \mathsf{inhpoly}(\emptyset)}{\cdot; x_1 : b, x_2 : L(b) \vdash x_1 :: x_2\ \mathsf{inhpoly}(\emptyset)}\ (\text{IP:Cons})$$

$$\frac{\Delta; \Gamma, y : b_1 \vdash e_\ell\ \mathsf{inhpoly}(V[x \mapsto y]) \qquad \Delta; \Gamma, y : b_2 \vdash e_r\ \mathsf{inhpoly}(V[x \mapsto y])}{\Delta; \Gamma, x : b_1 + b_2 \vdash \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\}\ \mathsf{inhpoly}(V)}\ (\text{IP:Case-Sum})$$

$$\frac{\Delta; \Gamma, x_1 : b_1, x_2 : b_2 \vdash e\ \mathsf{inhpoly}(V[x \mapsto x_1, x_2])}{\Delta; \Gamma, x : b_1 \times b_2 \vdash \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\}\ \mathsf{inhpoly}(V)}\ (\text{IP:Case-Prod})$$

$$\frac{\Delta; \Gamma \vdash e_0\ \mathsf{inhpoly}(V \setminus \{x\}) \qquad \Delta; \Gamma, x_1 : b, x_2 : L(b) \vdash e_1\ \mathsf{inhpoly}(V[x \mapsto x_1, x_2])}{\Delta; \Gamma, x : L(b) \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\}\ \mathsf{inhpoly}(V)}\ (\text{IP:Case-List})$$

$$\frac{\Delta; \Gamma \vdash e_0\ \mathsf{inhpoly}(V) \qquad \cdot; y : b, ys : L(b), z : b_2 \vdash e_1\ \mathsf{inhpoly}(\{y, ys\})}{\Delta; \Gamma, x : L(b) \vdash \mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\}\ \mathsf{inhpoly}(V \cup \{x\})}\ (\text{IP:Rec})$$

$$\frac{\Delta_1; \Sigma_1; \Gamma_1 \vdash e_1\ \mathsf{inhpoly}(V_1) \qquad \Delta_2; \Gamma_2, x : b \vdash e_2\ \mathsf{inhpoly}(V_2)}{\Delta_1 \cup \Delta_2; \Sigma_1 \cup \Gamma_1 \cup \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{inhpoly}(V_3)}\ (\text{IP:Let-Base})$$

$$\frac{\Delta; \Gamma, x_1 : b, x_2 : b \vdash e\ \mathsf{inhpoly}(V[x \mapsto x_1, x_2])}{\Delta; \Gamma, x : b \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e\ \mathsf{inhpoly}(V)}\ (\text{IP:Share-Base})$$

Figure 1 Key inference rules of inherently polynomial time.

5 Typable Fragment of Resource-Aware ML

It is nontrivial to prove that inherently polynomial time (Section 4.2) implies typability in multivariate AARA. The chief challenge is to come up with a suitable statement of a typability theorem (i) that we can prove by induction and (ii) that satisfies the following two requirements. Firstly, because a term e may later be used as an input to a function, it must be possible to type e such that a user-specified (i.e. arbitrary) amount of potential remains in e's output. Secondly, to type primitive recursion, we need to establish an invariant of resource annotations that is analogous to a loop invariant in Hoare logic. Specifically, given a primitive recursion $\mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\}$, we must give an (almost) identical annotation to both z, which is the result of a recursive call, and e_1, which is a stepping function.

Typability Theorem

We have partially overcome this challenge, and this section presents the result that inherently polynomial time implies typability in multivariate AARA under some restrictions. Detailed proofs of Theorem 6 and Theorem 9 are available in Appendix E of [30].

▶ **Definition 4** (Variables with zero potential). *Let $\Gamma \cup \{v : b\}$ be a base-type typing context and P be its multivariate annotation. Variable v is said to contain zero potential in P if and only if $P(i, j) = 0$ for every $i \in \mathcal{I}(\Gamma)$ and $j \in \mathcal{I}(\{v : b\})$ such that $j \neq 0_b$. In other words, the potential represented by P is constant with respect to $|v|$.*

▶ **Assumption 5.** *Suppose we are given $\Delta; \Sigma; \Gamma \vdash e\ t$ for $t \in \{\mathsf{inhpoly}(V), \mathsf{const}, \mathsf{poly}\}$. For every sub-derivation $\Delta_s; \Sigma_s; \Gamma_s \vdash e_s\ \mathsf{inhpoly}(V_s)$ inside the derivation of $\Delta; \Sigma; \Gamma \vdash e\ t$, we assume the following:*

- *If $e_s \equiv \mathsf{share}\ v\ \mathsf{as}\ v_1, v_2\ \mathsf{in} \cdots$, then v must be in V_s;*
- *If $e_s \equiv \mathsf{case}\ x\ \{[\,] \hookrightarrow \cdots \mid (y :: ys) \hookrightarrow \cdots\}$, then the type of x is of the form $L(b)$ where $b \in \mathbb{B}$ does not contain a list type; that is, x cannot be a nested list.*

The next theorem establishes that inherently polynomial time implies typability in multivariate AARA under Assumption 5, which restricts variable sharing and pattern matching on nested lists.

▶ **Theorem 6** (Inherently polynomial time implies typability). *Suppose we are given a term $\Sigma; \Gamma \vdash e : b$ with base type $b \in \mathbb{B}$, where $\Delta; \Sigma; \Gamma \vdash e\ \mathsf{inhpoly}(V)$ holds for some $V \subseteq dom(\Gamma)$. Additionally, assume Assumption 5. There exist P and Q satisfying $\Sigma; \Gamma; P \vdash e : \langle b, Q\rangle$ such that each $v \in dom(\Gamma) \setminus V$ contains zero potential (Definition 4).*

Consider an arrow-type term $\Sigma; \cdot \vdash e : b_1 \to b_2$ and assume Assumption 5. There exist P and Q such that $\Sigma; \cdot; 1 \vdash e : \langle b_1, P\rangle \to \langle b_2, Q\rangle$. Additionally, if $\Delta; \cdot; \Gamma \vdash e\ \mathsf{const}$ is true, P contains constant potential; i.e. b_1 stores zero potential in P.

Given a base-type expression e, if $\Delta; \Sigma; \Gamma \vdash e\ \mathsf{inhpoly}(V)$ holds, the running time of e is constant in the size of any $v \in \mathrm{dom}(\Gamma) \setminus V$. In other words, such v does not contribute to the computational cost of e. Therefore, it intuitively makes sense that such v contains zero potential in Theorem 6.

However, Theorem 6 cannot be immediately proved by induction on $\mathsf{inhpoly}(V)$, since the statement of the theorem is not strong enough for an inductive proof to go through. Specifically, a problem arises in the inductive case for (IP:Let-Base). In a let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, e_1 must carry sufficient potential to be transferred to e_2. However, Theorem 6 does not allow us to specify how much potential will remain available in the output of e.

Prior to remedying this issue, we first introduce the notion of *uniform resource annotations* for multivariate AARA.

▶ **Definition 7** (Uniform resource annotations for base types in multivariate AARA). *Given a base type $b \in \mathbb{B}$, let P be a multivariate resource annotation of b. P is said to be a* uniform multivariate annotation *with degree $d \in \mathbb{N}$ and number $n \in \mathbb{N}$ if and only if the following conditions hold*

1. *The maximum degree of P is at most d;*
2. *$P(i) = n$ for every $i \in \mathcal{I}(b)$ such that $\mathsf{deg}(i) = d$.*

In words, all coefficients of base polynomials with degree d (which should be the maximum degree) are equal to n. This will be denoted by a judgment $P\ \mathsf{uniform}(d, n)$.

▶ **Definition 8** (Uniform annotations for typing contexts in multivariate AARA). *Consider a term* $\Sigma;\Gamma \vdash e : b$ *of base type. Suppose that* $\Delta;\Sigma;\Gamma \vdash e$ $\mathsf{inhpoly}(V)$ *holds. Let* P *be a multivariate annotation for the base-type typing context* Γ*. We say that* P *is* uniform *with respect to degree* $d \in \mathbb{N}$*, number* $n \in \mathbb{N}$*, and set* V *of variables if and only if the following conditions hold:*

1. *For any base-type variable* $v \in dom(\Gamma) \setminus V$ *of type* b_v*, we have*

$$\forall i \in \mathcal{I}(\{v : b_v\}), j \in \mathcal{I}(\Gamma \setminus \{v : b_v\}).\mathsf{deg}(i) > d \implies P(i,j) = 0.$$

   In words, for any base polynomial with a non-zero coefficient in P*, its projection on* v *must have degree at most* d*.*
2. *For any* $v \in dom(\Gamma) \setminus V$ *of base type* b_v*, we have*

$$\forall i \in \mathcal{I}(\{v : b_v\}), j \in \mathcal{I}(\Gamma \setminus \{v : b_v\}).(\mathsf{deg}(i) = d \wedge j \neq 0) \implies P(i,j) = 0.$$

   In words, if a base polynomial has a non-zero coefficient and its projection on v *has degree* d*, then the base polynomial is not allowed to involve size variables of any other base-type variables from* $dom(\Gamma)$*.*
3. *For any* $v \in dom(\Gamma) \setminus V$ *of base type* b_v*, we have*

$$\forall i \in \mathcal{I}(\{v : b_v\}).\mathsf{deg}(i) = d \implies P(i,0) = n.$$

   That is, every base polynomial whose projection on v *has degree* d *has coefficient* n*.*

If these conditions hold, we denote P *being a uniform annotation by a judgment* P $\mathsf{uniform}(d,n,V)$*.*

Note that Definition 8 is a generalization of Definition 7. P $\mathsf{uniform}(d,n)$ in Definition 7 is equivalent to P $\mathsf{uniform}(d,n,\emptyset)$ in Definition 8.

Now that we have the notion of uniform annotations in place, we next present Theorem 9 that allows us to specify the amount of potential remaining in the output of a program. The major difficulty of the proof lies in establishing an invariant for primitive recursion as explained at the start of Section 5. We employ the notion of uniform annotations to characterize this invariant.

▶ **Theorem 9** (Existence of a multivariate annotation with arbitrary potential in the output). *Given a term* $\Sigma;\Gamma \vdash e : b$ *with* $b \in \mathbb{B}$*, suppose that* $\Delta;\Sigma;\Gamma \vdash e$ $\mathsf{inhpoly}(V)$ *holds, where* $V \subseteq dom(\Gamma)$*. Also, assume Assumption 5. Fix a multivariate annotation* Q *for the base type* b *such that* Q $\mathsf{uniform}(d,n)$*. Then there exists a multivariate annotation* P *such that* $\Sigma;\Gamma;P \vdash e : \langle b, Q\rangle$ *under the cost-free metric. Furthermore,* P $\mathsf{uniform}(d,n,V)$ *holds.*

Consider an arrow-type term $\Sigma;\cdot \vdash e : b_1 \to b_2$ *and assume Assumption 5. Fix a multivariate annotation* Q *for base type* b_2 *such that* Q $\mathsf{uniform}(d,n)$*. Then there exists* P *such that* $\Sigma;\cdot;0 \vdash e : \langle b_1, P\rangle \to \langle b_2, Q\rangle$ *under the cost-free metric. Furthermore, if* $\Delta;\Sigma;\cdot \vdash e$ const *is true,* P $\mathsf{uniform}(d,n)$ *holds.*

The cost-free metric in Theorem 9 refers to the cost metric in which all evaluation costs are zero. For instance, if $f : L(\mathbf{1}) \to L(\mathbf{1})$ is a function that doubles the size of an input list, it can be typed as $f : \langle L^2(\mathbf{1}), 0\rangle \to \langle L^1(\mathbf{1}), 0\rangle$ under the cost-free metric[4]. That is, the potential stored in each element is halved because the length of the list is doubled. If the cost

[4] For readability, I use univariate AARA instead of multivariate AARA to denote resource-annotated types, although Theorem 9 concerns multivariate AARA

metric is the running time, we instead have $f : \langle L^{2+c}(\mathbf{1}), 0\rangle \to \langle L^1(\mathbf{1}), 0\rangle$, where c is the cost of processing each list element. The type system of multivariate AARA under the cost-free metric is provided in Appendix D.2 of [30]. Theorem 9 uses the cost-free metric (as opposed to the running time) since Theorem 6 has already considers the cost of evaluating programs.

Theorem 9 assumes Assumption 5 as the proof of the theorem poses technical challenges in variable sharing and pattern matching on nested lists. We will now look at these challenges more closely.

Variable Sharing

Theorem 9 is false if we impose no restrictions on variable sharing. To illustrate this, consider e defined as

$$e := \mathsf{rec}\ x\ \{[\,] \hookrightarrow \langle \ell, \ell\rangle \mid (y :: \_)\ \mathsf{with}\ z \hookrightarrow e_1\}, \tag{5.1}$$

where the stepping function is $e_1 \equiv \mathsf{case}\ z\ \{\langle z_1, z_2\rangle \hookrightarrow \mathsf{share}\ z_1\ \mathsf{as}\ z_{1,1}, z_{1,2}\ \mathsf{in}\ \langle z_{1,1}, z_{1,2}\rangle\}$. The typing context of e in (5.1) is $\Gamma = \{x : L(\mathbf{1}), \ell : L(\mathbf{1})\}$. The stepping function satisfies $e_1\ \mathsf{inhpoly}(\{y, ys\})$. Hence, (5.1) is indeed inherently polynomial time. However, inside e_1, we have $\mathsf{share}\ z_1$, which Assumption 5 forbids.

Let $\langle \ell_1, \ell_2\rangle$ be the output of (5.1). Suppose that both ℓ_1 and ℓ_2 are to be annotated with $L^1(\mathbf{1})$. To type (5.1) under the cost-free metric such that $\ell_1, \ell_2 : L^1(\mathbf{1})$, the typing context Γ of e needs to be annotated with $2|\ell| + |x| \cdot |\ell|$, where $|\cdot|$ denotes the size of an input list. Observe that we need to use multivariate AARA rather than univariate AARA to type (5.1).

In the notation[5] of univariate AARA, the stepping function of (5.1) can be typed as

$$y : \mathbf{1}, ys : L^0(\mathbf{1}), z : L^2(\mathbf{1}) \times L^0(\mathbf{1}); 0 \vdash e_1 : \langle L^1(\mathbf{1}) \times L^1(\mathbf{1}), 0\rangle.$$

Here, the maximum degree is $d = 1$. It is impossible for both z and e_1 to have the same coefficient for all base polynomials of degree $d = 1$. Therefore, Theorem 9 is false for (5.1). To accommodate the multivariate annotation of (5.1), it is necessary to relax the notion of uniform resource annotations, but this will make the typability proof more challenging.

Nested Lists in Pattern Matching

Theorem 9 is false for pattern matching on nested lists. For example, consider e defined as

$$e := \mathsf{case}\ x\ \{[\,] \hookrightarrow \_ \mid (y :: ys) \hookrightarrow \langle y, ys\rangle\},$$

where the first branch is unimportant in the present discussion. The typing context of e is $\Gamma = \{x : L(L(\mathbf{1}))\}$. Assume that we consider multivariate annotations of degree up to $d = 2$. Let P denote a multivariate annotation of Γ. The multivariate annotation for context $\{y : L(\mathbf{1}), ys : L(L(\mathbf{1}))\}$ as a result of pattern matching on $x : L(L(\mathbf{1}))$ is given by the *additive shift* of P, denoted by $\lhd(P)$. It is defined as

$$\lhd(P)(i, j) := \begin{cases} P(0_{L(\mathbf{1})} :: j) + P(j) & \text{if } i = 0_{L(\mathbf{1})}; \\ P(i :: j) & \text{otherwise,} \end{cases} \tag{5.2}$$

where $i \in \mathcal{I}(\{y : L(\mathbf{1})\})$ and $j \in \mathcal{I}(\{ys : L(L(\mathbf{1}))\})$. The problem is that the base polynomial (i, j) on the left hand side of (5.2) has degree $\mathsf{deg}(i)+\mathsf{deg}(j)$, while $(i :: j)$ in the second branch

[5] Although we are concerned with multivariate AARA, I will use univariate AARA to denote the resource annotation of e_1 because it happens to be describable by univariate AARA and it is easier to read.

of the right hand side has degree $1+\mathsf{deg}(i)+\mathsf{deg}(j)$. As a consequence, if $1+\mathsf{deg}(i)+\mathsf{deg}(j) = 2$, $P(i :: j)$ is required to be equal to n because Theorem 9 requires P $\mathsf{uniform}(d, n)$ to be true. This means $\lhd(P)(i, j) = n$ must hold as well. But $\lhd(P)(i, j) = n$ is not necessarily the case, since Theorem 9 imposes no requirements on the coefficients of lower-degree base polynomials.

6 Conclusion

In this work, we have shown that polynomial-time Turing machines can be embedded in a typable fragment of RaML in such a way that the semantics and worst-case cost bounds are preserved. Moreover, we have proved that if a first-order program P satisfies the following conditions, it is guaranteed to be typable in multivariate polynomial AARA:

1. P uses primitive recursion instead of general recursion;
2. P is (axiomatically) inherently polynomial-time;
3. No variable sharing is applied to variable v, where P's running time is (axiomatically) constant in v;
4. No pattern matching is applied to a nested list.

We have neither found a counterexample to the full typability theorem (i.e. Theorem 6 without Assumption 5) nor proved it. As future work, we are looking to investigate how to prove or disprove the full typability theorem. To lift the restriction on nested lists, we expect that it suffices to modify the statement of the theorem such that we can keep track of the largest coefficient. However, lifting the restriction on variable sharing will be more challenging because it certainly requires a drastically different inductive hypothesis.

References

1 Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*, 2007.

2 Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110287.

3 Martin Avanzini and Georg Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

4 Spephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *computational complexity*, 2(2):97–110, June 1992. doi:10.1007/BF01201998.

5 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

6 Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 316–329, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009858.

7 Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds. In *Computer Aided Verification - 29th Int. Conf. (CAV'17)*, 2017.

8 Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 184–198, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325716.

9 Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 133–144, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328457.

10 Florian Frohn, M. Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. Lower Runtime Bounds for Integer Programs. In *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*, 2016.

11 Bernd Grobauer. Cost recurrences for dml programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, page 253–264, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507666.

12 Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371092.

13 Jan Hoffmann. *Types with potential: polynomial resource bounds via automatic amortized analysis*. PhD thesis, Ludwig Maximilians University Munich, 2011. URL: http://edoc.ub.uni-muenchen.de/13955/.

14 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3), November 2012. doi:10.1145/2362389.2362393.

15 Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 359–373, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009842.

16 Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 287–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

17 Martin Hofmann. The strength of non-size increasing computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 260–269, New York, NY, USA, 2002. ACM. doi:10.1145/503272.503297.

18 Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 185–197, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/604131.604148.

19 Martin Hofmann and Georg Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, 2014.

20 Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 223–236, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706327.

21 Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59:87–120, 2017. doi:10.1007/s10817-016-9398-9.

22 David M. Kahn and Jan Hoffmann. Exponential automatic amortized resource analysis. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, pages 359–380, Cham, 2020. Springer International Publishing.

23 G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371083.

24 Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 248–262, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062373.

25 Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158142.

26 Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

27 Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 274–288, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

28 V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, 2017.

29 Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 496–512, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192394.

30 Long Pham and Jan Hoffmann. Typable fragments of polynomial automatic amortized resource analysis, 2020. arXiv:2010.16353.

31 Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158124.

32 Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Matrix Analysis and Applications*, 6(2):306–13, April 1985. Copyright - Copyright] © 1985 © Society for Industrial and Applied Mathematics; Last updated - 2012-02-28. URL: https://search-proquest-com.proxy.library.cmu.edu/docview/923648267?accountid=9902.

33 Pedro B. Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, UK, 2008. URL: http://hdl.handle.net/10023/564.

34 Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975. doi:10.1145/361002.361016.