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.2021.26

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 [28], best-case [11, 32], or expected [7, 33] resource consumption and are based on type systems [8, 36, 10, 30, 2, 6, 13], recurrence relations [37, 12, 1, 29, 27], relational reasoning [6, 34], and term rewriting [3, 5, 23].

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 [22] for deriving linear heap-space bounds for a first-order language with lists. AARA has subsequently been extended to univariate polynomial bounds [18], multivariate polynomial bounds [15, 16], and exponential bounds [26]. Furthermore, AARA has been extended to other language features such as higher-order and polymorphic functions [24, 17], lazy evaluation [25], and probabilistic programming [33]. The analysis has been implemented in the programming language Resource-Aware ML (RaML) [17]. 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, 31, 21], 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 [15]. 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 [35], 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 [18, 16], 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 [18].

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 [17]. 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

$$\begin{array}{lll@{\qquad}lll}
b ::= & \mathbf{1} & \text{unit type} & \tau ::= & b & \text{base type} \\
& b_1 + b_2 & \text{sum type} & & b_1 \to b_2 & \text{arrow type} \\
& b_1 \times b_2 & \text{product type} & & & \\
& L(b) & \text{list type.} & & &
\end{array}$$

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

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

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.

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 [18]. 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.

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:

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

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 $\vec{q} = (q_1, \dots, q_k)$, $\phi(n, \vec{q})$ 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.

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

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

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.

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

▶ **Theorem 1** (Soundness of univariate AARA [18]). *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*

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

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

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

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:

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

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:

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

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

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

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

The degree of an index is defined by

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

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 $\Gamma = \{x_1 : b_1, \dots, x_n : b_n\}$, 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 $\Sigma = \{f_1 : b_{1,1} \to b_{1,2}, \dots, f_m : b_{m,1} \to b_{m,2}\}$, its multivariate resource annotation has the form

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

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.

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 $\ell_1 : L(\mathbf{1}), \ell_2 : L(\mathbf{1}); P \vdash$ *append* $\langle \ell_1, \ell_2 \rangle : \langle L(\mathbf{1}), Q \rangle$, where the positive coefficients of P and Q are

$$\begin{aligned} &P(\langle [*], * \rangle) = P(\langle [*,*], * \rangle) = P(\langle [*], [*] \rangle) = P(\langle *, [*,*] \rangle) = 2 && P(\langle *, [*] \rangle) = 1 \\ &Q([*,*]) = 2 && Q([*]) = 1. \end{aligned}$$

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 [16].

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 [15].

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.

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.

Algorithm 1 Operational working of target RaML program M'

```
Require: w ∈ {0,1}*
procedure M'(w)
    Create a singleton list ℓ1 : L(Sym) containing ⊢
    Create a list ℓ2 : L(Sym) of size p(|w|) filled with ⊔
    Prepend ℓ2 with w
    Create a list ps : L(1) of size p(|w|)            ▷ Reservoir of potential
    s ← q0                                            ▷ Initialize the current state
    while s ≠ q_final ∧ ps ≠ [] do
        ps ← tail ps                                  ▷ Potential is released
        Compute δ(s, ℓ2[0])
        Update s and ℓ2[0] appropriately
        Update the tape head's position by moving the head of ℓ1 or ℓ2 to the other
    return append(reverse ℓ1, ℓ2)
```

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 [26], 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

[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.

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

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

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. $\mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: \mathit{ys})\ \mathsf{with}\ z \hookrightarrow e_1\}$, where z denotes the result of the recursive call.

In primitive recursion, e_1 is only allowed to mention $\{y, \mathit{ys}, z\}$. 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

$$\mathsf{fun}\ f\ \langle x, \Gamma\rangle = \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid y :: \mathit{ys} \hookrightarrow \mathsf{share}\ \mathit{ys}\ \mathsf{as}\ \mathit{ys}_1, \mathit{ys}_2\ \mathsf{in}\ \mathsf{let}\ z = f\ \langle \mathit{ys}_1, \Gamma\rangle\ \mathsf{in}\ e_1\}.$$

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 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.

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:

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

---

[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.

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.

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.

▶ **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 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. 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

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

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

[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.

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 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. URL: `https://doi.org/10.1145/3110287`, `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, Jun 1992. URL: `https://doi.org/10.1007/BF01201998`, `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. URL: `https://doi.org/10.1145/3009837.3009858`, `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. URL: `https://doi.org/10.1145/325694.325716`, `doi:10.1145/325694.325716`.

9 Ugo Dal Lago and Sara Zuppiroli. Probabilistic recursion theory and implicit computational complexity. In Gabriel Ciobanu and Dominique Méry, editors, *Theoretical Aspects of Computing – ICTAC 2014*, pages 97–114, Cham, 2014. Springer International Publishing.

10 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. URL: `https://doi.org/10.1145/1328438.1328457`, `doi:10.1145/1328438.1328457`.

11 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.

12 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. URL: `https://doi.org/10.1145/507635.507666`, `doi:10.1145/507635.507666`.

13 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. URL: `https://doi.org/10.1145/3371092`, `doi:10.1145/3371092`.

14 Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2 edition, 2016. `doi:10.1017/CBO9781316576892`.

15 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/`.

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

17 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. URL: `https://doi.org/10.1145/3009837.3009842`, `doi:10.1145/3009837.3009842`.

18 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.

19 Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic*, pages 275–294, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

20 Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 464–, Washington, DC, USA, 1999. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=788021.788962`.

21 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. URL: `http://doi.acm.org/10.1145/503272.503297`, `doi:10.1145/503272.503297`.

22 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. URL: `https://doi.org/10.1145/604131.604148`, `doi:10.1145/604131.604148`.

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

24 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. URL: `https://doi.org/10.1145/1706299.1706327`, `doi:10.1145/1706299.1706327`.

25 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. URL: `https://doi.org/10.1007/s10817-016-9398-9`.

26 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.

27 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. URL: `https://doi.org/10.1145/3371083`, `doi:10.1145/3371083`.

28 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. URL: `https://doi.org/10.1145/3062341.3062373`, `doi:10.1145/3062341.3062373`.

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

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

31 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.

32 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.

33 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. URL: `https://doi.org/10.1145/3192366.3192394`, `doi:10.1145/3192366.3192394`.

34 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. URL: `https://doi.org/10.1145/3158124`, `doi:10.1145/3158124`.

35 Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Matrix Analysis and Applications*, 6(2):306–13, 04 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`.

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

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

A Supplementary Results for the Embedding of Polynomial-Time Turing Machines in AARA

Throughout Appendix A, we will write $b_1 \xrightarrow{q_1/q_2} b_2$ for resource-annotated arrow types, instead of $\langle b_1, q_1\rangle \to \langle b_2, q_2\rangle$. Here, b_i is a resource-annotated base type and $q_i \in \mathbb{Q}_{\geq 0}$.

A.1 Standard-Form Polynomials in Univariate AARA

In conventional mathematics taught at school, (univariate) polynomials are expressed in the standard form of $a_n x^n + \cdots + a_0 x^0$, where x is a variable and a_i's are coefficients. On the other hand, in univariate AARA, polynomials are encoded as linear combinations of binomial coefficients $\binom{x}{a}$, where x is a variable and a is a constant. The following proposition establishes that these two representations of univariate polynomials are interchangeable.

▶ **Lemma 10.** *For any $d \in \mathbb{N}$, the polynomial function n^d can be expressed as $\sum_{i=0}^{d} q_i \binom{n}{i}$, where $q_i \in \mathbb{Q}_{\geq 0}$ for all $0 \leq i \leq d$.*

Proof. The proof goes by induction on d. The claim clearly holds when $d = 0$.

For the inductive case, by way of example, we will first illustrate how to prove the lemma for $d = 4$, given that the claim holds when $d = 3$. For simplicity, we will use the basis of $i! \cdot \binom{n}{i}$ rather than $\binom{n}{i}$ in this example. Suppose we have

$$n^3 = p_1 n + p_2 n(n-1) + p_3 n(n-1)(n-2).$$

This yields

$$\begin{aligned}
n^4 &= n \cdot (p_1 n + p_2 n(n-1) + p_3 n(n-1)(n-2)) \\
&= p_1 n^2 + p_2 n^2 (n-1) + p_3 n^2 (n-1)(n-2) \\
&= p_1 n((n-1)+1) + p_2 n(n-1)((n-2)+2) + p_3 n(n-1)(n-2)((n-3)+3) \\
&= p_1 n + (p_1 + 2p_2) n(n-1) + (p_2 + 3p_3) n(n-1)(n-2) + p_3 n(n-1)(n-2)(n-3),
\end{aligned}$$

where all coefficients are non-negative, provided that each p_i is non-negative as well. Generalizing this, we learn that $n^k = \sum_{i=0}^{k} p_i \cdot n(n-1)\cdots(n-i+1)$ gives

$$n^{k+1} = p_k \cdot n(n-1)\cdots(n-k) + \sum_{i=1}^{k} (p_{i-1} + i \cdot p_i) \cdot n(n-1)\cdots(n-i+1).$$

If the coefficients p_i for n^k are non-negative, so are the coefficients for n^{k+1}.

Finally, to switch from the new basis to the original basis of binomial coefficients, we use the identity $p_i = \frac{q_i}{i!}$. This gives

$$\begin{aligned}
n^{k+1} &= \frac{q_k}{k!} \cdot n(n-1)\cdots(n-k) + \sum_{i=1}^{k} \left(\frac{q_{i-1}}{(i-1)!} + i \cdot \frac{q_i}{i!} \right) \cdot n(n-1)\cdots(n-i+1) \\
&= q_k k \cdot \binom{n}{k+1} + \sum_{i=1}^{k} i(q_{i-1} + q_i) \binom{n}{i}. \qquad \text{(A.1)}
\end{aligned}$$

This concludes the proof. ◀

A.2 Generating Lists of Polynomial Size

We will explain how to generate, in Resource-Aware ML (RaML), a list of polynomial size $p(n)$ with constant potential stored in each cell. To this end, it suffices to show how to generate a list of size $\binom{n}{d}$ for a fixed $d \in \mathbb{N}$ because n^k for any $k \in \mathbb{N}$ can be expressed as a non-negative linear combination of $\binom{n}{0}, \dots, \binom{n}{k}$. This has formally been established by Lemma 10.

Without loss of generality, we assume that the output is a list of blank symbols (as required in line 3 of Algorithm 1). Let $amp_d : L(\mathrm{Sym}) \to L(\mathrm{Sym}) \to L(\mathrm{Sym})$ denote a RaML function that (i) generates a list of size $\binom{n}{d}$ in which each cell stores one unit of potential and (ii) appends it to an accumulator, which is assumed to already contain one unit of potential in each cell. n is the size of the first input to amp_d, and the accumulator is the second input. Here, *amp* stands for amplification.

amp_0 is defined as

$$\textsf{fun}\ amp_0\ w\ acc = \sqcup :: acc. \tag{A.2}$$

For $i \geq 0$, amp_{i+1} is inductively defined as

$$\begin{aligned}
\textsf{fun}\ amp_{i+1}\ w\ acc = \textsf{case}\ w\ \{[\,] \hookrightarrow acc \\
&\mid x :: xs \hookrightarrow \textsf{share}\ xs\ \textsf{as}\ xs_1, xs_2\ \textsf{in} \\
&\qquad \textsf{let}\ _ = \textsf{tick}\ 1\ \textsf{in} \\
&\qquad \textsf{let}\ acc' = amp_i\ xs_1\ acc\ \textsf{in} \\
&\qquad amp_{i+1}\ xs_2\ acc'\}.
\end{aligned} \tag{A.3}$$

We use tick to account for only the cost of function application but not costs of other operations such as the list constructor. This is why (A.2) does not generate any costs. Although the syntax of RaML presented in Section 2.1 only permits uncurried functions, we will use curried functions throughout Appendix A without loss of generality.

This implementation is analogous to the example given in Section 7.1 of [18], where given a list ℓ, all subsets of ℓ with a fixed size are computed. However, our implementation in (A.3) differs from the implementation in [18] in that ours uses an accumulator, while the one in [18] explicitly uses the *append* function. The use of an accumulator allows us to embed both generation of elements and their concatenation in the implementation of amp_d, saving us the need to explicitly reason about the computational cost of *append*. Hence, using an accumulator can simplify the cost analysis of amp_d, although admittedly amp_d with an accumulator is not the most natural implementation from the perspective of programmers.

We consider the generation of lists of size $\binom{n}{d}$ instead of n^d for a similar reason. Due to the identity $\binom{n+1}{d+1} = \binom{n}{d+1} + \binom{n}{d}$, we do not need to appeal to any auxiliary function in (A.3). On the other hand, to recursively create a list of size $n^d = n \cdot n^{d-1}$, one needs to use an iterator function, and this will complicate the analysis of total costs since we will need to account for the cost of invoking the iterator function.

The next proposition establishes the correctness of the above implementation (i.e. (A.2) and (A.3)) and provides an upper bound on the evaluation cost.

▶ **Lemma 11** (Correctness of amp_d). *The computation of $amp_d\ w\ acc$ produces a list of size $\binom{|w|}{d} + |acc|$, where $\binom{n}{k} = 0$ for $n < k$. Also, assuming that each cell in the output list is required to contain one unit of potential, the cost of evaluating $amp_d\ w\ acc$ is bounded above by $2|w|^d$.*

Proof. The proof goes by nested induction: outer induction on d and inner induction on $|w|$. For the base case where $d = 0$, the output size is indeed $\binom{n}{0} + |acc| = 1 + |acc|$. With regard to the evaluation cost, we need one unit of potential to execute (A.2) since the new cell requires one unit of potential. As $2n^0 = 2$ for every $n \in \mathbb{N}$, $2n^d$ is a correct upper bound in this case. Here, we adopt the convention of $0^0 = 1$.

For the inductive case, suppose that the claim holds when $d = k$ for some $k \geq 0$. The proof proceeds by (inner) induction on $|w|$. When $|w| = 0$, we have

$$|amp_{k+1}\ w\ acc| = 0 + |acc|$$

according to the first branch of pattern matching in (A.3). Also, the evaluation cost is 0. Hence, the claim holds when w is empty.

Conversely, if $w = x :: xs$, we have

$$\begin{aligned}
|amp_{k+1}\ w\ acc| &= |amp_{k+1}\ xs\ (amp_k\ xs\ acc)| && \text{by (A.3)}\\
&= \binom{|xs|}{k+1} + \binom{|xs|}{k} + |acc| && \text{by the inductive hypothesis}\\
&= \binom{|w|-1}{k+1} + \binom{|w|-1}{k} + |acc| && \text{because } w = x :: xs\\
&= \binom{|w|}{k+1} + |acc|.
\end{aligned}$$

Regarding the evaluation cost, we write $\mathrm{cost}(amp_{k+1}\ w\ acc)$ for the evaluation cost of $amp_{k+1}\ w\ acc$. If $k \geq 1$, we have

$$\begin{aligned}
\mathrm{cost}(amp_{k+1}\ w\ acc) &= 1 + \mathrm{cost}(amp_k\ xs\ acc) + \mathrm{cost}(amp_{k+1}\ xs\ acc') && \text{by (A.3)}\\
&\leq 1 + 2|xs|^k + 2|xs|^{k+1} && \text{by the inductive hypothesis}\\
&\leq 2(1 + |xs|)^k + 2|xs|^{k+1} && \text{because } k \geq 1\\
&\leq 2(1 + |xs|)^k + 2|xs| \cdot (1 + |xs|)^k\\
&= 2(1 + |xs|)^{k+1}\\
&= 2(|w|)^{k+1}.
\end{aligned}$$

If $k = 0$, we have

$$\begin{aligned}
\mathrm{cost}(amp_1\ xs\ acc) &= 1 + \mathrm{cost}(amp_0\ xs\ acc) + \mathrm{cost}(amp_1\ w\ acc') && \text{by (A.3)}\\
&\leq 1 + 1 + 2|xs| && \text{by the inductive hypothesis}\\
&= 2(|w|),
\end{aligned}$$

where in the second line, we use the tight bound $\mathrm{cost}(amp_0\ w\ acc) = 1$. Therefore, the claim is true regardless of whether $k = 0$ or $k \geq 1$. This concludes the proof. ◀

$2n^d$ is a tight cost bound of amp_d when $d = 1$. However, $2n^d$ is not a tight bound anymore when $d = 0$ or $d > 1$. The general tight bound is probably complicated to express.

The next proposition claims that AARA can infer that $2n^d$ is an upper bound on the evaluation cost.

▶ **Lemma 12** (Typability of amp_d)**.** *AARA can infer the resource-annotated type*

$$amp_d : L^{2\vec{q}_d}(Sym) \xrightarrow{0/0} L^1(Sym) \xrightarrow{0/0} L^1(Sym), \tag{A.4}$$

where vector $\vec{q}_d \in \mathbb{Q}^d_{\geq 0}$ represents the function $n \mapsto n^d$. To be more precise, since vectors from $\mathbb{Q}^0_{\geq 0}$ cannot express constants, when $d = 0$, the resource-annotated type should be written as

$$amp_0 : L^0(Sym) \xrightarrow{2/0} L^1(Sym) \xrightarrow{0/0} \langle L^1(Sym), 0\rangle. \tag{A.5}$$

Keep in mind that the resource annotation that AARA returns in reality can be a more accurate bound than (A.4)*.*

Proof. To prove the claim, it is sufficient to show that (A.4) is a valid resource annotation that satisfies all relevant typing rules of AARA. The proof goes by induction on d. For the base case of $d = 0$, the claim holds since amp_d requires exactly 1 potential unit.

We now turn to the inductive case. We can assign resource-annotated types to some variables appearing in (A.3) as

$$w : L(\langle \mathrm{Sym}, 2\vec{q}_d\rangle) \qquad xs_1 : L(\langle \mathrm{Sym}, (\triangleleft 2\vec{q}_d) - 2\vec{q}_d\rangle) \qquad xs_2 : L(\langle \mathrm{Sym}, 2\vec{q}_d\rangle).$$

We will now argue that this annotation correctly accounts for the evaluation cost in every recursive call.

Firstly, from (A.1), we can derive that the first component of vector $\vec{q}_d$ for any $d \geq 1$ is 1. This means that 2 units of potential is available in each recursive call. Hence, we use this constant potential to account for `tick` 1 in the definition of amp_d.

After deducting 2 units from the potential stored in w, we have $\triangleleft 2\vec{q}_d$ units of potential remaining, and this represents $2n^d - 2$. We need to split it between xs_1 and xs_2 in such a way that we can pay for the costs of $amp_{d-1}\ xs_1\ acc$ and $amp_d\ xs_2\ acc'$. For the former, the inductive hypothesis suggests $amp_{d-1} : L^{2\vec{q}_{d-1}}(\mathrm{Sym}) \xrightarrow{0/0} L^1(\mathrm{Sym}) \xrightarrow{0/0} L^1(\mathrm{Sym})$, provided that $d \geq 2$. If $d = 1$, we need to conduct separate analysis since the type of amp_0 in (A.5) is distinct from the type for amp_d for $d \geq 1$. Nonetheless, we will assume $d \geq 2$ in the present proof as it is straightforward to adapt this proof to the case of $d = 1$. For the recursive call $amp_d\ xs_2\ acc'$, ideally, we would like to reuse the resource-annotated type of amp_d; otherwise, resource-polymorphic recursion would arise, complicating the proof. As a consequence, our goal is to show

$$2\vec{q}_{d-1} + 2\vec{q}_d \leq \triangleleft 2\vec{q}_d,$$

where $+$ and $\leq$ are applied component-wise. This is equivalent to $\vec{q}_{d-1} + \vec{q}_d \leq \triangleleft \vec{q}_d$ because $\triangleleft$ is linear and hence $\triangleleft 2\vec{q}_d = 2 \cdot (\triangleleft \vec{q}_d)$, where $\cdot$ is scalar multiplication.

If $\vec{q}_{d-1} = (q_1, q_2, \ldots, q_{d-1})$, it follows from (A.1) that

$$\vec{q}_d = (q_1, 2(q_1 + q_2), 3(q_2 + q_3), \ldots, (d-1)(q_{d-2} + q_{d-1}), (d-1)q_q).$$

This yields

$$\vec{q}_{d-1} + \vec{q}_d = (2q_1, 2q_1 + 3q_2, \ldots, (d-1)q_{d-1} + dq_{d-1}, (d-1)q_{d-1}),$$

which is smaller than $\triangleleft \vec{q}_d$ component-wise. Therefore, $\vec{q}_{d-1} + \vec{q}_d \leq \triangleleft \vec{q}_d$ indeed holds. ◀

A.3 Target RaML Programs

For convenience, the definition of Turing machines is reproduced below.

▶ **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.*

Given a source program M, the target program M' can be expressed as

$$
\begin{aligned}
\mathsf{fun}\ M'\ w = {} & \mathsf{share}\ w\ \mathsf{as}\ w_1, w_2, w_3\ \mathsf{as} \\
& \mathsf{let}\ \ell_1 = \vdash :: [\,]\ \mathsf{in} \\
& \mathsf{let}\ \ell_2' = \mathit{amp}_{d,\sqcup}\ w_1\ [\,]\ \mathsf{in} \\
& \mathsf{let}\ \ell_2 = \mathit{append}\ w_2\ \ell_2'\ \mathsf{in} \\
& \mathsf{let}\ \mathit{ps} = \mathit{amp}_{d,\langle\rangle}\ w_3\ [\,]\ \mathsf{in} \\
& \mathit{simulate}\ q_0\ \ell_1\ \ell_2\ \mathit{ps},
\end{aligned}
\tag{A.6}
$$

where $\mathit{amp}_{d,\sqcup}$ creates a list of size $\binom{|w|}{d}$ filled with blank symbols, and $\mathit{amp}_{d,\langle\rangle}$ performs the same task, except that the output is filled with $\langle\rangle$ instead of $\sqcup$. If $p(n)$, which is the polynomial representing M's running time, cannot be expressed in the form of $\binom{n}{d}$ for any $d \in \mathbb{N}$, we express $p(n)$ as a linear combination of binomial coefficients (due to Lemma 10) and hard-code this linear combination inside (A.6).

The auxiliary functions *append* and *simulate* are defined as

$$
\begin{aligned}
\mathsf{fun}\ \mathit{append}\ \ell_1\ \ell_2 = \mathsf{case}\ \ell_1\ \{ & [\,] \hookrightarrow \ell_2 \\
\mid {} & x :: \mathit{xs} \hookrightarrow \mathsf{let}\ \_ = \mathsf{tick}\ 1\ \mathsf{in} \\
& \qquad \mathsf{let}\ \mathit{xs}' = \mathit{append}\ \mathit{xs}\ \ell_2\ \mathsf{in} \\
& \qquad x :: \mathit{xs}' \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{fun}\ \mathit{simulate}\ s\ \ell_1\ \ell_2\ \mathit{ps} \\
& \quad = \mathsf{case}\ \mathit{ps}\ \{ [\,] \hookrightarrow \quad \mathit{shift}\ \ell_1\ \ell_2 \\
& \qquad\qquad \mid p :: \mathit{ps}' \hookrightarrow \quad \mathsf{let}\ \_ = \mathsf{tick}\ 1\ \mathsf{in} \\
& \qquad\qquad\qquad \mathsf{let}\ (s', b, \mathrm{direction}) = \delta(s, \mathit{head}\ \ell_2)\ \mathsf{in} \\
& \qquad\qquad\qquad \mathsf{if}\ s' = q_{\mathrm{final}}\ \mathsf{then} \\
& \qquad\qquad\qquad \mathit{shift}\ \ell_1\ \ell_2 \\
& \qquad\qquad\qquad \mathsf{else\ if}\ \mathrm{direction} = L\ \mathsf{then} \\
& \qquad\qquad\qquad \mathit{simulate}\ s'\ (\mathit{tail}\ \ell_1)\ ((\mathit{head}\ \ell_1) :: b :: (\mathit{tail}\ \ell_2))\ \mathit{ps}' \\
& \qquad\qquad\qquad \mathsf{else} \\
& \qquad\qquad\qquad \mathit{simulate}\ s'\ (b :: \ell_1)\ (\mathit{tail}\ \ell_2)\ \mathit{ps}' \}.
\end{aligned}
\tag{A.7}
$$

The if-else constructs in (A.7) (e.g. $\mathsf{if}\ s' = q_{\mathrm{final}}\ \mathsf{then}\ \cdots$) can be encoded using case for the Boolean type (i.e. $\mathbf{1} + \mathbf{1}$). The function *shift* used in (A.7) reverses the first input list and appends it to the second input list:

$$\begin{aligned}\mathsf{fun}\ \mathit{shift}\ \ell_1\ \ell_2 = \mathsf{case}\ \ell_1\ \{[\,] &\hookrightarrow \ell_2\\ \mid x :: \mathit{xs} &\hookrightarrow \mathsf{let}\ \_ = \mathsf{tick}\ 1\ \mathsf{in}\\ &\quad\ \mathsf{let}\ \mathit{ys} = x :: \ell_2\ \mathsf{in}\\ &\quad\ \mathit{shift}\ \mathit{xs}\ \mathit{ys}\}.\end{aligned}$$

In (A.7), for the sake of brevity, we use the standard form of function application in place of let-normal form. Also, $(s', b, \text{direction}) = \delta(s, \mathit{head}\ \ell_2)$ is a slight abuse of notation because this is ill-formed with respect to the syntax of RaML and also because we would need to introduce a new type for directions (i.e. L or R). Nevertheless, we write it this way to keep δ general. If we are given a specific transition function, we can embed it in the code, dedicating one branch of if-else statements (or pattern matching) to each possible combination of s (i.e. the current machine state) and $\mathit{head}\ \ell_2$ (i.e. the symbol in the current cell).

Lastly, for completeness, *head* and *tail* are defined as

$$\begin{aligned}\mathsf{fun}\ \mathit{head}\ \ell &= \mathsf{case}\ \ell\ \{[\,] \hookrightarrow \mathsf{error} \mid x :: \mathit{xs} \hookrightarrow x\}\\ \mathsf{fun}\ \mathit{tail}\ \ell &= \mathsf{case}\ \ell\ \{[\,] \hookrightarrow \mathsf{error} \mid x :: \mathit{xs} \hookrightarrow \mathit{xs}\}.\end{aligned}$$

Finally, we are now in a position to prove the theorem about embedding polynomial-time Turing machines in RaML.

▶ **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'.*

Proof. Without loss of generality, assume that the running time of M is bounded by $n \mapsto \binom{n}{d}$ for some fixed $d \in \mathbb{N}$. In this case, a desirable M' is defined in (A.6). If this assumption is false, we can use Lemma 10 to express a polynomial as a liner combination of binomial coefficients and hard-code it in (A.6). By construction, $M'(w) = M(w)$ for every $w \in \{0,1\}^*$.

Throughout the execution of M', ℓ_1, ℓ_2, and *ps* must contain one unit of potential in each cell. The potential in ℓ_1 and ℓ_2 will be used to account for $\mathit{shift}\ \ell_1\ \ell_2$ right before M' terminates. The potential stored in *ps* is for the execution of *simulate*.

Hence, the input w to M' must contain sufficient potential to pay for the following costs:

- Creating singleton list ℓ_1 that contains $\vdash$. Due to the invariant we impose on ℓ_1's potential, it requires one unit of potential to create ℓ_1 in the initial configuration.
- Creating list ℓ_2' of size $\binom{|w|}{d}$, which has one unit of potential in each cell.
- Appending w to ℓ_2' to create ℓ_2, which has size $|w| + \binom{|w|}{d}$ and stores one unit of potential in each cell. Thus, we have $|w|$ many units of potential to execute *append* and another $|w|$ units to be stored in the first $|w|$ cells of ℓ_2. This gives a total of $2|w|$ units of potential.
- Creating list *ps* of size $\binom{|w|}{d}$, which stores one unit of potential in each cell.

By Proposition 12, the second (for ℓ_2') and fourth (for *ps*) costs above can be each covered by $2|w|^d$ units of potential. Hence, summing the above three costs, we obtain $1 + 2|w|^d + 2|w|^d + 2|w|$. This is the amount of potential that must be stored in the input w to M' at the start of computation.

Each auxiliary function appearing in (A.6) can be type-annotated. More concretely, amp_d can be type-annotated as shown in Proposition 12, and *simulate* can be assigned this type:

$$simulate : \mathrm{State} \to L^1(\mathrm{Sym}) \xrightarrow{0/0} L^1(\mathrm{Sym}) \xrightarrow{0/0} L^1(\mathrm{Sym}) \xrightarrow{0/0} L^0(\mathrm{Sym}).$$

Since it is relatively easy to see that this type can be inferred using AARA, we will omit its formal proof. In summary, univariate AARA can infer a polynomial cost bound of M'. ◀

B Resource-Aware ML (RaML)

This section gives the type system of RaML (Section 2.1) and defines its the running time.

B.1 Simple Type System of RaML

The simple (i.e. non-resource-annotated) type system of RaML is displayed in Figure 2. Throughout Figure 2, $b \in \mathbb{B}$ denotes a base type, and τ denotes a simple type. Because RaML is a first-order language, τ is either $b \in \mathbb{B}$ or $b_1 \to b_2$ (Section 2.2).

$$\frac{}{x : \tau \vdash x : \tau}\ (\text{T:Var}) \qquad \frac{\Gamma \vdash x : b_1}{\Gamma \vdash \ell \cdot x : b_1 + b_2}\ (\text{T:SumL}) \qquad \frac{\Gamma \vdash x : b_2}{\Gamma \vdash r \cdot x : b_1 + b_2}\ (\text{T:SumR})$$

$$\frac{}{\cdot \vdash \langle\rangle : \mathbf{1}}\ (\text{T:Unit}) \qquad \frac{\Gamma_1 \vdash x_1 : b_1 \qquad \Gamma_2 \vdash x_2 : b_2}{\Gamma_1 \cup \Gamma_2 \vdash \langle x_1, x_2 \rangle : b_1 \times b_2}\ (\text{T:Pair})$$

$$\frac{}{\cdot \vdash [\,] : L(b)}\ (\text{T:Nil}) \qquad \frac{\Gamma_1 \vdash x_1 : b \qquad \Gamma_2 \vdash x_2 : L(b)}{\Gamma_1 \cup \Gamma_2 \vdash (x_1 :: x_2) : L(b)}\ (\text{T:Cons})$$

$$\frac{\Gamma, f : b_1 \to b_2, x : b_1 \vdash e : b_2}{\Gamma \vdash \mathsf{fun}\ f\ x = e : b_1 \to b_2}\ (\text{T:Fun}) \qquad \frac{\Gamma_1 \vdash x_1 : b_1 \to b_2 \qquad \Gamma_2 \vdash x_2 : b_1}{\Gamma_1 \cup \Gamma_2 \vdash x_1\ x_2 : b_2}\ (\text{T:App})$$

$$\frac{\Gamma_1 \vdash x : b_1 + b_2 \qquad \Gamma_2, y : b_1 \vdash e_\ell : b_3 \qquad \Gamma_2, y : b_2 \vdash e_r : b_3}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\} : b_3}\ (\text{T:Case-Sum})$$

$$\frac{\Gamma_1 \vdash x : b_1 \times b_2 \qquad \Gamma_2, x_1 : b_1, x_2 : b_2 \vdash e : b_3}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{case}\ x\ \{\langle x_1, x_2 \rangle \hookrightarrow e\} : b_3}\ (\text{T:Case-Product})$$

$$\frac{\Gamma_1 \vdash x : L(b) \qquad \Gamma_2 \vdash e_0 : b_2 \qquad \Gamma_2, x_1 : b, x_2 : L(b) \vdash e_1 : b_2}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\} : b_2}\ (\text{T:Case-List})$$

$$\frac{}{\cdot \vdash \mathsf{tick}\ q : \mathbf{1}}\ (\text{T:Tick}) \qquad \frac{\Gamma_1 \vdash e_1 : \tau \qquad \Gamma_2, x : \tau \vdash e_2 : b}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : b}\ (\text{T:Let})$$

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \sigma}{\Gamma, x : \tau \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e : \sigma}\ (\text{T:Share}) \qquad \frac{\Gamma_1 \vdash e : \tau \qquad \Gamma_1 \subseteq \Gamma_2}{\Gamma_2 \vdash e : \tau}\ (\text{T:Weak})$$

Figure 2 Simple type system of RaML.

B.2 Running Time of RaML

A *cost semantics* of programming language L is a mapping $\mathrm{cost}(\cdot) : [\![L]\!] \to \mathbb{Q}_{\geq 0}$. Here, $[\![L]\!]$ is the set of all programs in L where inputs are already included in them; that is, programs in $[\![L]\!]$ have base types as opposed to arrow types. In automatic amortized resource analysis

(AARA), cost semantics are defined by specifying how each inference rule of the big-step operational semantics gives rise to computational costs. A mapping from inference rules of the operational semantics to $\mathbb{Q}$ (or vectors of $\mathbb{Q}$) is referred to as a *cost metric*. Examples of cost metrics include the running time and memory usage.

The running time is given by a judgment $V \vdash e \Downarrow v \mid n$, 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. This judgment is defined in Figure 3.

Environment V may contain redundant variables that are not allowed to appear in the program. However, this is not problematic—specifying what variables can appear and what cannot is the job of RaML's type system (Figure 2), not the job of RaML's cost semantics.

$$\frac{}{V \vdash x \Downarrow V(x) \mid 1}\ (\text{E:Var}) \qquad \frac{}{V \vdash \langle\rangle \Downarrow \langle\rangle \mid 0}\ (\text{E:Unit})$$

$$\frac{d \in \{\ell, r\} \qquad V \vdash x \Downarrow v \mid 1}{V \vdash d \cdot x \Downarrow d \cdot v \mid 2}\ (\text{E:Sum}) \qquad \frac{V \vdash x_1 \Downarrow v_1 \mid 1 \qquad V \vdash x_2 \Downarrow v_2 \mid 1}{V \vdash \langle x_1, x_2\rangle \Downarrow \langle v_1, v_2\rangle \mid 3}\ (\text{E:Pair})$$

$$\frac{}{V \vdash [\,] \Downarrow [\,] \mid 0}\ (\text{E:Nil}) \qquad \frac{V \vdash x_1 \Downarrow v \mid 1 \qquad V \vdash x_2 \Downarrow \ell \mid 1}{V \vdash (x_1 :: x_2) \Downarrow (v :: \ell) \mid 3}\ (\text{E:Cons})$$

$$\frac{}{V \vdash \mathsf{fun}\ f\ x = e \Downarrow \mathrm{closure}(V; f, x.e) \mid 1}\ (\text{E:Fun})$$

$$\frac{V(x_1) = \mathrm{closure}(U; f, x.e) \qquad U, f \mapsto \mathrm{closure}(V; f, x.e), x \mapsto V(x_2) \vdash e \Downarrow v \mid q}{V \vdash x_1\ x_2 \Downarrow v \mid 1 + q}\ (\text{E:App})$$

$$\frac{d \in \{\ell, r\} \qquad V(x) = d \cdot v \qquad V, y \mapsto v \vdash e_d \Downarrow v_d \mid q}{V \vdash \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\} \Downarrow v_d \mid 1 + q}\ (\text{E:Case-Sum})$$

$$\frac{V(x) = \langle v_1, v_2\rangle \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e \Downarrow v \mid q}{V \vdash \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\} \Downarrow v \mid 1 + q}\ (\text{E:Case-Product})$$

$$\frac{V(x) = [\,] \qquad V \vdash e_0 \Downarrow v \mid q}{V \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\} \Downarrow v \mid 1 + q}\ (\text{E:Case-Nil})$$

$$\frac{V(x) = v :: \ell \qquad V, x_1 \mapsto v, x_2 \mapsto \ell \vdash e_1 \Downarrow v' \mid q}{V \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\} \Downarrow v' \mid 1 + q}\ (\text{E:Case-Cons})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid q \qquad V, x \mapsto v_1 \vdash e_2 \Downarrow v_2 \mid p}{V \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2 \mid 1 + q + p}\ (\text{E:Let})$$

$$\frac{V(x) = v \qquad V, x_1 \mapsto v, x_2 \mapsto v \vdash e \Downarrow v' \mid q}{V \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e \Downarrow v' \mid q}\ (\text{E:Share})$$

Figure 3 Inference rules defining the running time of RaML.

In (E:Fun), $\mathrm{closure}(V; f, x.e)$ is a function closure where V is an environment and e is code that may mention f and x. The notation $f, x.e$ means that f, x are bound in e.

In (E:Unit) and (E:Nil), the running time is zero. One might argue that these cases could have positive running time in practice. If we are to assign positive running time to (E:Unit) and (E:Nil), we will need to revise the type systems of unvarite and multivariate AARA accordingly. However, it will not fundamentally affect the theorems and their proofs in this article. For instance, consider a primitive recursion $\mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\}$, which can be encoded using (i) general recursion and (ii) lists' pattern matching. Even if the stepping function e_1 has zero running time (e.g. due to (E:Nil)), the total running time of the whole primitive recursion is strictly positive and is proportional to the length of an input list, thanks to (E:Case-Cons). Consequently, even if (E:Unit) and (E:Nil) have zero running time as they do now, the running time of any primitive recursion is non-zero.

We do not have a rule for tick, because it is only necessary for the tick metric.

C Univariate Polynomial AARA

C.1 Type system

This section presents a type system for univariate polynomial AARA. The content and presentation style of this section are attributed to Hoffmann's PhD thesis [15].

The typing judgment for univariate polynomial AARA has the form

$$\Gamma_{\text{anno}}; q \vdash e : A, \tag{C.1}$$

where q is a non-negative rational number and Γ_{anno} is a resource-annotated typing context (i.e. a set of pairs of variable symbols and their resource-annotated types). A has the form $\langle b, p\rangle$, where b is a resource-annotated base type and $p \in \mathbb{Q}_{\geq 0}$ (Section 2.2).

Syntax-Directed Rules

The syntax-directed rules for univariate polynomial AARA are presented in Figure 4.

In (U:Cons), q_1 is not a fresh variable—it denotes the first component of vector $\vec{q}$.

In (U:Fun) and (U:App), $\mathcal{T}$ is a (finite or infinite) set of resource-annotated arrow types of the form $B_1 \to B_2$, where $b_i = \langle b_i, q_i\rangle$. Each function $\mathsf{fun}\ f\ x = e$ is associated with such set $\mathcal{T}$. For instance, the identity function $id : L(\mathbf{1}) \to L(\mathbf{1})$ always consumes one unit of potential. Therefore, the largest possible $\mathcal{T}$ for id is

$$\mathcal{T} = \{\langle L^{\vec{p}}(\mathbf{1}), q + c\rangle \to \langle L^{\vec{p}}(\mathbf{1}), q\rangle \mid q \in \mathbb{Q}_{\geq 0}, c \in \mathbb{Q}_{\geq 1}, \\ \vec{p} \text{ is any valid univariate polynomial annotation}\}.$$

Of course, $\mathcal{T}$ need not be as large as it can—it only needs to be self-sufficient in the sense that every $(B_1 \to B_2) \in \mathcal{T}$ can be derived using $\mathcal{T}$ itself (See (U:Fun)). Since $\mathcal{T}$ is a set of types rather than a single type, the typing judgment $f : \mathcal{T}$ does not conform to (C.1). Nonetheless, we will keep writing $f : \mathcal{T}$ because it conveniently conveys that we are allowed to assign multiple resource annotations to f.

The use of $f : B_1 \to B_2$ to derive a different arrow type $f : C_1 \to C_2$ of f itself, where f is recursively defined (i.e. f mentions itself), is called *resource-polymorphic recursion* [18]. This phenomenon arises in many recursive functions. An example is provided in [18].

It is reasonable to wonder how we can possibly "deterministically" infer resource annotations, given that $\mathcal{T}$ is allowed to be infinite. For instance, we may have $\mathcal{T} = \{B_1 \to C_1, B_2 \to C_2, B_3 \to C_3, \ldots\}$, where in order to type $f : B_i \to C_i$ for each i, we need to use $f : B_{i+1} \to C_{i+1}$. Consequently, resource-polymorphic recursion may induce an infinite chain of (U:Fun). This problem will be discussed in detail in the following section.

The first premise of (U:Fun) states $\Gamma = |\Gamma|$. Informally, $|\cdot|$ deletes all resource annotations inside resource-annotated base types (but does nothing to resource-annotated arrow types).

$$\frac{}{x : \tau; 1 \vdash x : \langle \tau, 0 \rangle} \text{(U:Var)} \qquad \frac{}{\cdot; 0 \vdash \langle\,\rangle : \langle \mathbf{1}, 0 \rangle} \text{(U:Unit)}$$

$$\frac{\Gamma; 1 \vdash x : \langle b_1, 0 \rangle}{\Gamma; 2 \vdash \ell \cdot x : \langle b_1 + b_2, 0 \rangle} \text{(U:SumL)} \qquad \frac{\Gamma; 1 \vdash x : b_2}{\Gamma; 2 \vdash r \cdot x : \langle b_1 + b_2, 0 \rangle} \text{(U:SumR)}$$

$$\frac{}{x_1 : b_1, x_2 : b_2; 3 \vdash \langle x_1, x_2 \rangle : \langle b_1 \times b_2, 0 \rangle} \text{(U:Pair)} \qquad \frac{}{\cdot; 0 \vdash [\,] : \langle L^{\vec{q}}(b), 0 \rangle} \text{(U:Nil)}$$

$$\frac{}{x_1 : b, x_2 : L^{\lhd(\vec{q})}(b); 3 + q_1 \vdash x_1 :: x_2 : \langle L^{\vec{q}}(b), 0 \rangle} \text{(U:Cons)}$$

$$\frac{\Gamma = |\Gamma| \qquad \forall (\langle b, q \rangle \to B_2) \in \mathcal{T}. |\Gamma|, f : \mathcal{T}, x : b; q \vdash e : B_2}{\Gamma; 1 \vdash \mathsf{fun}\ f\ x = e : \langle \mathcal{T}, 0 \rangle} \text{(U:Fun)}$$

$$\frac{(B_1 \to B_2) \in \mathcal{T} \qquad B_1 = \langle b, q \rangle}{f : \mathcal{T}, x : b; 1 + q \vdash f\ x : B_2} \text{(U:App)}$$

$$\frac{\Gamma, y : b_1; q \vdash e_\ell : B \qquad \Gamma, y : b_2; q \vdash e_r : B}{x : b_1 + b_2, \Gamma; 1 + q \vdash \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r\} : B} \text{(U:Case-Sum)}$$

$$\frac{\Gamma, x_1 : b_1, x_2 : b_2; q \vdash e : B}{x : b_1 \times b_2, \Gamma; 1 + q \vdash \mathsf{case}\ x\ \{\langle x_1, x_2 \rangle \hookrightarrow e\} : B} \text{(U:Case-Product)}$$

$$\frac{\Gamma; q \vdash e_0 : B \qquad \Gamma, x_1 : b, x_2 : L^{\lhd(\vec{p})}(b); q + p_1 \vdash e_1 : B}{x : L^{\vec{p}}(b), \Gamma; 1 + q \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid x_1 :: x_2 \hookrightarrow e_1\} : B} \text{(U:Case-List)}$$

$$\frac{\Gamma_1; q \vdash e_1 : \langle \tau, p \rangle \qquad \Gamma_2, x : \tau; p \vdash e_2 : B}{\Gamma_1 \cup \Gamma_2; 1 + q \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B} \text{(U:Let)}$$

$$\frac{\tau \curlyvee (\tau_1, \tau_2) \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : \langle \tau, q \rangle}{\Gamma, x : \tau; q \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e : \langle \tau, q \rangle} \text{(U:Share)}$$

Figure 4 Syntax-directed rules of univariate AARA.

Formally, $|\tau|$, where τ is a resource-annotated simple type, is defined as follows:

$$\begin{aligned} |\mathbf{1}| &:= \mathbf{1} & |L^{\vec{q}}(b)| &:= L^0(|b|) \\ |b_1 + b_2| &:= |b_1| + |b_2| & |\langle b_1, q_1 \rangle \to \langle b_2, q_2 \rangle| &:= \langle b_1, q_1 \rangle \to \langle b_2, q_2 \rangle \\ |b_1 \times b_2| &:= |b_1| \times |b_2|. \end{aligned}$$

This can be generalized to typing contexts as follows: if $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, we have $|\Gamma| = x_1 : |\tau_1|, \ldots, x_n : |\tau_n|$.

The reason why we require $\Gamma = |\Gamma|$ in the first premise of (U:Fun) is that we want function closures (i.e. semantic values closure$(V; f, x.e)$) to store zero potential. If a function f is defined in the presence of a positive-potential base-type variable, it means f consumes potential every time it is invoked. However, in AARA, we do not keep track of how many times f is invoked—instead, we let it be invoked freely. Of course, we could modify AARA such that it computes an upper bound on the number of times f is invoked. However, for simplicity, we do not adopt this approach.

In (U:Share), $\tau \curlyvee (\tau_1, \tau_2)$ means the resource-annotated type τ can be split into τ_1 and τ_2. This is defined in Figure 5. When we split a variable $x : B_1 \to B_2$ into x_1 and x_2 (i.e. $\mathsf{share}\ x\ \mathsf{as}\ x_1, x_2$), x_1 and x_2 are allowed to have distinct annotations as long as these annotations can be legally derived using (U:Fun). Thus, it is possible to use a different resource annotation for a function f each time f is invoked. In Figure 5, $\mathcal{T}_f$ denotes a set of resource-annotated arrow types associated with function f that we consider now.

$$\frac{}{\mathbf{1} \curlyvee (\mathbf{1},\mathbf{1})} \qquad \frac{b_1 \curlyvee (b_{1,1}, b_{1,2}) \qquad b_2 \curlyvee (b_{2,1}, b_{2,2})}{(b_1 + b_2) \curlyvee (b_{1,1} + b_{2,1}, b_{1,2} + b_{2,2})} \qquad \frac{b_1 \curlyvee (b_{1,1}, b_{1,2}) \qquad b_2 \curlyvee (b_{2,1}, b_{2,2})}{(b_1 \times b_2) \curlyvee (b_{1,1} \times b_{2,1}, b_{1,2} \times b_{2,2})}$$

$$\frac{\vec{p} = \vec{p_1} + \vec{p_2} \qquad b \curlyvee (b_1, b_2)}{L^{\vec{p}}(b) \curlyvee (L^{\vec{p_1}}(b_1), L^{\vec{p_2}}(b_2))} \qquad \frac{(B_1 \to B_2), (C_1 \to C_2), (D_1 \to D_2) \in \mathcal{T}_f}{(B_1 \to B_2) \curlyvee (C_1 \to C_1, D_1 \to D_2)}$$

Figure 5 Definition of the sharing relation $\tau \curlyvee (\tau_1, \tau_2)$.

In (U:Cons) and (U:Case-List), we use the univariate additive shift operator (denoted by $\lhd$) that splits the potential of a list between the head element and the tail. This is formally defined below.

▶ **Definition 13** (Additive shift of potential vectors). *Given a potential vector $\vec{q} = (q_1, q_2, \cdots, q_k)$, its additive shift is defined as*

$$\lhd(\vec{q}) := (q_1 + q_2, q_2 + q_3, \cdots, q_{k-1} + q_k, q_k).$$

Structural Rules

The structural rules of univariate AARA are displayed in Figure 6. Also, the subtyping relation $\tau_1 <: \tau_2$ is defined in Figure 7.

$$\frac{\Gamma; q \vdash e : \langle \tau, p \rangle \qquad \tau <: \tau'}{\Gamma; q \vdash e : \langle \tau'.p \rangle}\ (\text{U:Sub}) \qquad \frac{\Gamma, x : \tau; q \vdash e : B \qquad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B}\ (\text{U:Sup})$$

$$\frac{\Gamma_1; q \vdash e : B}{\Gamma_1, \Gamma_2; q \vdash e : B}\ (\text{U:Weak}) \qquad \frac{\Gamma; p_1 \vdash e : \langle \tau, p_2 \rangle \qquad q_1 \geq p_1 \qquad q_1 - q_2 \geq p_1 - p_2}{\Gamma; q_1 \vdash e : \langle \tau, q_2 \rangle}\ (\text{U:Relax})$$

Figure 6 Structural rules of univariate AARA.

$$\frac{}{\mathbf{1} <: \mathbf{1}} \qquad \frac{b_1 <: c_1 \qquad b_2 <: c_2}{b_1 + b_2 <: c_1 + c_2} \qquad \frac{b_1 <: c_1 \qquad b_2 <: c_2}{b_1 \times b_2 <: c_1 \times c_2}$$

$$\frac{b_1 <: b_2 \qquad \vec{p_1} \geq \vec{p_2}}{L^{\vec{p_1}}(b_1) <: L^{\vec{p_2}}(b_2)} \qquad \frac{C_1 <: B_1 \qquad B_2 <: C_2}{B_1 \to B_2 <: C_1 \to C_2} \qquad \frac{b_1 <: b_2 \qquad q_1 \geq q_2}{\langle b_1, q_1 \rangle <: (b_2, q_2)}$$

Figure 7 Definition of the subtyping relation $\tau_1 <: \tau_2$ in univariate AARA.

C.2 Practical Type Inference

As mentioned above, resource-polymorphic recursion may induce an infinite chain of (U:Fun), causing type checking/inference to continue indefinitely. To work around this problem, the type inference algorithm of AARA in [18, 17] uses the following heuristic. Suppose we are to derive a resource-annotated arrow type $f : B_1 \to B_2$, where f is recursively defined. To derive $f : B_1 \to B_2$, suppose we need to use a distinct type $f : C_1 \to C_2$. Now consider the type $(C_1 - B_1) \to (C_2 - B_2)$, where $C_i - B_i$ denotes annotation-wise subtraction. For instance, if $C_i \equiv L^{\vec{p}_C}(L^{\vec{q}_C}(\mathbf{1}))$ and $B_i \equiv L^{\vec{p}_B}(L^{\vec{q}_B}(\mathbf{1}))$, then $C_i - B_i \equiv L^{\vec{p}_C - \vec{p}_B}(L^{\vec{q}_C - \vec{q}_B}(\mathbf{1}))$, where $\vec{p}, \vec{q}$ are vectors of $\mathbb{Q}_{\geq 0}$.

AARA's type inference algorithm requires the following:

- $f : (C_1 - B_1) \to (C_2 - B_2)$ can be derived under the *cost-free* metric;
- $(C_1 - B_1) \to (C_2 - B_2)$ has a strictly lower degree than $B_1 \to B_2$ and $C_1 \to C_2$.

In the first condition, the cost-free metric means any computational cost (e.g. function application, pattern matching, and variable lookups) is zero. The cost-free metric (as opposed to the cost metric of the running time) is used to derive $f : (C_1 - B_1) \to (C_2 - B_2)$ because the derivations of $B_1 \to B_2$ and $C_1 \to C_2$ already consider the actual computational cost (i.e. running time).

To paraphrase the second condition, $B_1 \to B_2$ and $C_1 \to C_2$ must share the same coefficients for the maximum degree. For example, consider $B_1 \equiv (6, 6, 1)$, which represents $6 \cdot \binom{n}{3} + 6 \cdot \binom{n}{2} + 1 \cdot \binom{n}{1} = n^3$. The degree of B_1 is 3 because B_1 is a vector of size (that is, B_1 represents a cubic polynomial function). Therefore, C_1 must have the shape $C_1 \equiv (6, p_2, p_3)$ for some rational numbers p_2, p_3 so that $C_1 - B_1$ has a strictly lower degree than B_1 and C_1. Thanks to this restriction, AARA's inference algorithm is guaranteed to terminate. However, it is not complete with respect to the type system in Figure 4: some polynomial-time recursive functions require an infinite set $\mathcal{T}$ in (U:Fun).

One might wonder whether the derivation of $f : (C_1 - B_1) \to (C_2 - B_2)$ under the cost-free metric requires resource-polymorphic recursion; that is, whether we will have a cascade/chain of resource-polymorphic recursion. Interestingly enough, according to [15], the derivations of cost-free types (e.g. $f : (C_1 - B_1) \to (C_2 - B_2)$) in univariate AARA only need resource-monomorphic recursion; i.e. $\mathcal{T}$ in (U:Fun) is a singleton set. In other words, in univariate AARA, the type inference algorithm only applies resource-polymorphic recursion once, which will then be followed by resource-monomorphic recursion. There is no justification for using resource-monomorphic recursion instead of resource-polymorphic recursion in the derivations of cost-free types. This approach probably works well for all programs that (i) are tested in [15] and (ii) are typable in univariate AARA even if we use a cascade of resource-polymorphic recursion. By contrast, in multivariate AARA, we sometimes have a cascade of resource-polymorphic recursion under the cost-free metric.

Finally, it is necessary to clarify what we mean by typability in this article. In univariate (and multivariate) AARA, we have two notions of typability:

1. A proof based on Figure 4, where $\mathcal{T}$ may be an infinite set, exists;
2. The type inference algorithm of AARA actually terminates and produces a polynomial cost bound, provided that the user specifies a sufficiently high degree.

In this article, we will use the latter, stronger notion of typability. For example, Theorem 6 and Theorem 9 use the stronger notion of typability: after each invocation of (U:Fun) (actually, the multivariate version (M:Fun)), the degree strictly decreases.

C.3 Examples

Section 2.2 presents univariate resource annotations of *append* and *quicksort*. They are implemented as follows:

$$\begin{aligned}
&\quad \mathit{append} : \langle L(\mathbf{1}), L(\mathbf{1})\rangle \to L(\mathbf{1})\\
&\quad \mathit{append} := \mathsf{fun}\ f\ \langle \ell_1, \ell_2\rangle = \mathsf{case}\ \ell_1\ \{[\,] \hookrightarrow \ell_2 \mid (y :: \mathit{ys}) \hookrightarrow y :: (f\ \langle \mathit{ys}, \ell_2\rangle)\}\\
&\mathit{quicksort} : L(b) \to L(b)\\
&\mathit{quicksort} := \mathsf{fun}\ f\ \ell = \mathsf{case}\ \ell\ \{[\,] \hookrightarrow [\,] \mid (y :: \mathit{ys}) \hookrightarrow e_1\}\\
&\qquad e_1 \equiv \mathsf{let}\ \langle \ell_1, \ell_2\rangle = \mathit{split}\ \langle y, \mathit{ys}\rangle, x_1 = f\ \ell_1, x_2 = f\ \ell_2\ \mathsf{in}\ \mathit{append}\ \langle x_1, y :: x_2\rangle.
\end{aligned}$$

For readability, we have abbreviate $\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ e_3$ to $\mathsf{let}\ x_1 = e_1, x_2 = e_2\ \mathsf{in}\ e_3$. Also, we use the notation $\mathsf{let}\ \langle x_1, x_2\rangle = e_1$ as syntactic sugar for $\mathsf{let}\ x = e_1\ \mathsf{in}\ \mathsf{case}\ e_1\ \{\langle x_1, x_2\rangle \hookrightarrow \cdots\}$.

Here, $\mathit{split}\ \langle y, \mathit{ys}\rangle$ classifies each element in the list ys according to whether it is smaller than y or not. *split* is defined as

$$\begin{aligned}
&\mathit{split} : (b \times L(b)) \to (L(b) \times L(b))\\
&\mathit{split} := \mathsf{fun}\ f\ \langle x, \ell\rangle = \mathsf{case}\ \ell\ \{[\,] \hookrightarrow \langle [\,], [\,]\rangle \mid (y :: \mathit{ys}) \hookrightarrow e_1\}\\
&\quad e_1 \equiv \mathsf{let}\ \langle \ell_1, \ell_2\rangle = f\ \langle x, \mathit{ys}\rangle\ \mathsf{in}\ \mathsf{if}\ y < x\ \mathsf{then}\ \langle y :: \ell_1, \ell_2\rangle\ \mathsf{else}\ \langle \ell_1, y :: \ell_2\rangle.
\end{aligned}$$

D Multivariate Polynomial AARA

This section describes multivariate polynomial AARA. As before, the content and presentation style of this section are attributed to Hoffmann's PhD thesis [15].

D.1 Notation

We will introduce the projection and extension operators on multivariate annotations. To define projection, suppose we are given a base-type typing context $\Gamma_1 \cup \Gamma_2$. If $j \in \mathcal{I}(\Gamma_2)$, the projection $\pi_j^{\Gamma_1}(Q)$ is defined as

$$\pi_j^{\Gamma_1}(Q)(i) := Q(i, j). \tag{D.1}$$

Next, to define the extension operator, let Q be a multivariate annotation over Γ_1. Given $r \in \mathcal{I}(\Gamma_2)$, the extension $\eta_r^{\Gamma_1 \cup \Gamma_2}(Q)$ is defined as

$$\eta_r^{\Gamma_1 \cup \Gamma_2}(Q)(i, j) := \begin{cases} Q(i) & \text{if } j = r;\\ 0 & \text{otherwise.}\end{cases} \tag{D.2}$$

Denoted by $\lhd$, additive shift specifies how to split the potential of a list between the head element and the tail. We now explain additive shift of multivariate AARA. Suppose that we are given a base-type context $\Gamma_1 = \Gamma \cup \{\ell : L(b)\}$, where ℓ is the last element without loss of generality. Let Q be a multivariate annotation over $\Gamma \cup \{\ell : L(b)\}$. $\ell : L(b)$ is split into the head element $x : b$ and the tail $\mathit{xs} : L(b)$, yielding a new typing context $\Gamma_2 = \Gamma \cup \{x : b, \mathit{xs} : L(b)\}$. The resource annotation for the new context is given by the (multivariate) additive shift of Q with respect to $\ell : L(b)$. This is denoted by $\lhd_\ell Q$ and is defined as

$$\lhd(Q)(i, j, k) := \begin{cases} Q(i, 0_b :: k) + Q(i, k) & \text{if } j = 0_b;\\ Q(i, j :: k) & \text{otherwise.}\end{cases} \tag{D.3}$$

Here, i, j, and k are indexes of Γ, $x : b$, and $xs : L(b)$, respectively. Recall that $j = 0_b$ refers to the base polynomial $\lambda v.1$ for type b. The notation $j :: k$ denotes the concatenation of j, which is an index for type b, to k, which is an index for type $L(b)$.

Lastly, for variable sharing, it is sufficient to know that for any multivariate resource annotation Q for a base-type typing context $\Gamma \cup \{x_1 : b, x_2 : b\}$, we have a way to produce P for the unified typing context $\Gamma \cup \{x : b\}$ such that no potential from Q is lost. The relation between Q and P will be denoted by $P \curlyvee_{x_1,x_2} Q$. The details of variable sharing are deferred to [15, 16].

D.2 Type System

We will consider both (i) the type system with the cost metric being the running time and (ii) the type system under the cost-free metric.

A typing judgment in multivariate AARA has the form

$$\Sigma_{\text{anno}}; \Gamma; Q \vdash e : \langle \tau, P \rangle.$$

Σ_{anno} is a resource-annotated arrow-type typing context, where each variable binding is of the form $f : \langle b_1, Q_1 \rangle \to \langle b_2, Q_2 \rangle$. Γ is a base-type typing context without resource annotations, and Q is a multivariate resource annotation/polynomial for Γ. Likewise, P is a multivariate resource annotation for τ. When τ is a resource-annotated arrow type, P will just be a constant.

Syntax-Directed Rules

Syntax-directed rules of multivariate AARA are presented in Figure 8. Anything enclosed by square brackets should be ignored if we use the cost-free metric. For instance, in (M:Var-Base), the resource annotation of the typing context in the conclusion is Q if we use the cost-free metric. Otherwise, if the cost metric is the running time, the resource annotation should be $1 + Q$.

In (M:Var-Base), $c + Q$, where $c \in \mathbb{Q}_{\geq 0}$ and Q is a multivariate resource annotation, denotes the addition of x to $Q(0)$; i.e. the constant potential of Q. The coefficients of all the other indexes, i.e. $i \neq 0 \in \mathcal{I}(\Gamma)$, remain unchanged.

In (M:Case-Sum), $Q[(\ell \cdot y)/x]$ denotes the result of discarding the resource annotations within Q for $x = r \cdot y$, which has a wrong tag.

In (M:Case-List), $\pi_i^{\Gamma}(Q)$ denotes the projection of Q on Γ, which is defined in (D.1).

According to the first premise of (M:Let), e_1 must carry $\pi_0^{x:b_1}(R)$ much potential after computation. This potential is part of R that only concerns size variables inside $x : b_1$. However, being multivariate in nature, R may involve a product of both size variables outside x and those inside x. How can we squeeze such multivariate potential into Q such that it becomes available in R? To achieve this, we additionally impose this requirement: for each $j \in \mathcal{I}(\Gamma_2)$ such that $j \neq 0_{\Gamma_2}$, we have

$$\Sigma_1; \Gamma_1; \pi_j^{\Gamma_1} Q \vdash e_1 : \langle b_1, \pi_j^{x:b_1}(R) \rangle$$

under the cost-free metric. We could not place this requirement inside the rule (M:Let) simply due to the shortage of space.

$Q \curlyvee_{x_1,x_2} P$ in (M:Share-Base) is a multivariate extension of the sharing relation $\tau \curlyvee (\tau_1, \tau_2)$ in Figure 5. In the interest of space, $Q \curlyvee_{x_1,x_2} P$ is not formally defined in this article.

$$\frac{Q(0_b) = 0}{\cdot; x : b; [1+]Q \vdash x : \langle b, Q\rangle} \text{ (M:Var-Base)} \qquad \frac{\tau \equiv B_1 \to B_2}{f : \tau; \cdot; [1] \vdash f : \langle \tau, 0\rangle} \text{ (M:Var-Arrow)}$$

$$\frac{Q(0_b) = 0}{\cdot; x : b_1; [1+]Q \vdash \ell \cdot x : \langle b_1 + b_2, Q\rangle} \text{ (M:SumL)} \qquad \frac{Q(0_b) = 0}{\cdot; x : b_2; [1+]Q \vdash r \cdot x : \langle b_1 + b_2, Q\rangle} \text{ (M:SumR)}$$

$$\frac{}{\cdot; \cdot; 0 \vdash \langle\rangle : \langle \mathbf{1}, 0\rangle} \text{ (M:Unit)} \qquad \frac{Q(0_b) = 0}{\cdot; x_1 : b_1, x_2 : b_2; [3+]Q \vdash \langle x_1, x_2\rangle : \langle b_1 \times b_2, Q\rangle} \text{ (M:Pair)}$$

$$\frac{Q(0_b) = 0}{\cdot; \cdot; 0 \vdash [\,] : \langle L(b), Q\rangle} \text{ (M:Nil)} \qquad \frac{Q = \lhd_{x_1 :: x_2}(Q')}{\cdot; x_1 : b, x_2 : L(b); [3+]Q \vdash x_1 :: x_2 : \langle L(b), Q'\rangle} \text{ (M:Cons)}$$

$$\frac{\Gamma = |\Gamma| \qquad \forall(\langle b, Q\rangle \to B_2) \in \mathcal{T}.\Sigma, f : \mathcal{T}; \Gamma, x : b; \eta_0^{\Gamma, x:b}(Q) \vdash e : B_2}{\Sigma; \Gamma; 1 \vdash \mathsf{fun}\ f\ x = e : \langle \mathcal{T}, 0\rangle} \text{ (M:Fun)}$$

$$\frac{(B_1 \to B_2) \in \mathcal{T} \qquad B_1 = \langle b, Q\rangle}{f : \mathcal{T}; x : b; [1+]Q \vdash f\ x : B_2} \text{ (M:App)}$$

$$\frac{\Sigma; y : b_1, \Gamma; Q[(\ell \cdot y)/x] \vdash e_\ell : B \qquad \Sigma; y : b_2, \Gamma; Q[(r \cdot y)/x] \vdash e_r : B}{\Sigma; x : b_1 + b_2, \Gamma; [1+]Q \vdash \mathsf{case}\ x\ \{\ell \cdot y \hookrightarrow e_\ell \mid r \cdot y \hookrightarrow e_r : B\}} \text{ (M:Case-Sum)}$$

$$\frac{\Sigma; x_1 : b_1, x_2 : b_2, \Gamma; Q \vdash e : B}{\Sigma; x : b_1 \times b_2, \Gamma; [1+]Q \vdash \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\} : B} \text{ (M:Case-Prod)}$$

$$\frac{\Sigma; \Gamma; \pi_0^\Gamma(Q) \vdash e_0 : B \qquad \Sigma; \Gamma, y : b, ys : L(b); \lhd_x(Q) \vdash e_1 : B}{\Sigma; x : L(b), \Gamma; [1+]Q \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys) \hookrightarrow e_1\} : B} \text{ (M:Case-List)}$$

$$\frac{\Sigma_1; \Gamma_1; \pi_0^{\Gamma_1}(Q) \vdash e_1 : \langle b_1, \pi_0^{x:b_1}(R)\rangle \qquad \Sigma_2; x : b_1, \Gamma_2; R \vdash e_2 : B}{\Sigma_1 \cup \Sigma_2; \Gamma_1 \cup \Gamma_2; [1+]Q \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B} \text{ (M:Let)}$$

$$\frac{\Sigma; \Gamma, x_1 : b, x_2 : b; P \vdash e : B \qquad Q \curlyvee_{x_1, x_2} P}{\Sigma; \Gamma, x : b; Q \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e : B} \text{ (M:Share-Base)}$$

$$\frac{\Sigma, f_1 : \tau_1, f_2 : \tau_2; \Gamma; Q \vdash e : B \qquad \tau, \tau_1, \tau_2 \in \mathcal{T}_f}{\Sigma, f : \tau; \Gamma; Q \vdash \mathsf{share}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ e : B} \text{ (M:Share-Arrow)}$$

Figure 8 Syntax-directed rules of multivariate AARA. Anything enclosed by square brackets is ignored if the cost-free metric is used. For instance, if we have $[1+]Q$, it represents Q in case of the cost-free metric and $1 + Q$ in case the cost metric is the running time.

Finally, in the premise of (M:Share-Arrow), τ_i must be a valid resource annotation of f_i derived using (M:Fun). However, τ_1 does not need to be identical to τ_2.

Like in univariate AARA (Appendix C), the type system of multivariate AARA in Figure 8 can give rise to an infinite chain of (M:Fun). To circumvent this, the type inference algorithm of multivariate AARA requires adopts the heuristic described in Appendix C.2. The only difference is that, under the cost-free metric, multivariate AARA's type inference creates a cascade of (M:Fun) where each invocation of (M:Fun) reduces the degree by one. On the other hand, under the cost-free metric, univariate AARA's type inference uses resource-monomorphic recursion rather than resource-polymorphic recursion.

Structural Rules

Structural rules are presented in Figure 9.

$$\frac{\Sigma; \Gamma; Q \vdash e : \langle \tau, P_1 \rangle \qquad P_1 <: P_2}{\Sigma; \Gamma; Q \vdash e : \langle \tau, P_2 \rangle} \text{ (M:SUB)} \qquad \frac{\Sigma; \Gamma; Q_1 \vdash e : B \qquad Q_2 <: Q_1}{\Sigma; \Gamma; Q_2 \vdash e : B} \text{ (M:SUP)}$$

$$\frac{\Sigma_1 \subseteq \Sigma_2 \qquad \Sigma_1; \pi_0^{\Gamma}(Q) \vdash e : B}{\Sigma_2; \Gamma; Q \vdash e : B} \text{ (M:WEAK)}$$

$$\frac{\Sigma; \Gamma; P \vdash e : \langle \tau, P' \rangle \qquad Q = P + c \qquad Q' = P' + c}{\Sigma; \Gamma; Q \vdash e : \langle \tau, Q' \rangle} \text{ (M:RELAX)}$$

Figure 9 Structural rules of multivariate AARA.

The subtyping relationship is determined by the point-wise inequality of coefficients of resource polynomials: $Q <: P$ if and only if $\forall i \in \mathcal{I}(\Gamma).Q(i) \geq P(i)$.

E Proof of the Typability Theorem

This section provides detailed proofs of Theorem 6 and Theorem 9. The inductive proof of Theorem 6 makes use of Theorem 9, while the inductive proof of Theorem 9 is self-contained; that is, its statement is strong enough for an inductive proof to go through.

First of all, in Figure 10, we present three inference rules for inherently polynomial time that are missing from Figure 1.

$$\frac{\Delta_1; \cdot \vdash e_1 \ \mathsf{time} \qquad \Delta_2, x \ \mathsf{time}; \Gamma, x : b \vdash e_2 \ \mathsf{inhpoly}(V)}{\Delta_1 \cup \Delta_2; \Gamma \vdash \mathsf{let} \ x = e_1 \ \mathsf{in} \ e_2 \ \mathsf{inhpoly}(V)} \text{ (IP:LET-ARROW)}$$

$$\frac{\Delta[x \ \mathsf{time} \mapsto x_1 \ \mathsf{time}, x_2 \ \mathsf{time}]; \Gamma, x_1 : b_1 \to b_2, x_2 : b_1 \to b_2 \vdash e \ \mathsf{inhpoly}(V)}{\Delta; \Gamma, x : b_1 \to b_2 \vdash \mathsf{share} \ x \ \mathsf{as} \ x_1, x_2 \ \mathsf{in} \ e \ \mathsf{inhpoly}(V)} \text{ (IP:SHARE-ARROW)}$$

$$\frac{\Delta_1; \Gamma_1 \vdash e \ \mathsf{inhpoly}(V_1) \qquad \Delta_1 \subseteq \Delta_2 \qquad \Gamma_1 \subseteq \Gamma_2 \qquad V_1 \subseteq V_2}{\Delta_2; \Gamma_2 \vdash e \ \mathsf{inhpoly}(V_2)} \text{ (IP:WEAKEN-BASE)}$$

Figure 10 Remaining rules for inherently polynomial time

In (IP:WEAKEN-BASE), we require V_2 to be a subset of base-type variables in $\mathrm{dom}(\Gamma)$.

▶ **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.

Proof. The proof proceeds by structural induction on $\Delta; \Sigma; \Gamma \vdash e \ \mathsf{inhpoly}(V)$.

For base cases, we have (IP:BASE), (IP:ARROW), (IP:UNIT), and (IP:NIL). In all of them, since the running time is constant, we only need constant potential. Thus, the theorem is indeed true.

Moving on to inductive cases, let us consider (IP:SumL):

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

Again, $\ell \cdot x$ runs in constant time. Therefore, it is easy to see that the theorem holds. The same reasoning applies to (IP:SumR), (IP:Pair), and (IP:Cons).

(IP:Const) and (IP:Poly) are straightforward.

Next, we consider (IP:App-Const):

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

From the inductive hypothesis, it is given that $x_1 : b_1 \to b_2$ can be typed as $\langle b_1, P\rangle \to \langle b_2, Q\rangle$, where P contains constant potential. Therefore, $x_1\ x_2$ can be annotated in such a way that the annotation for x_2 contains zero potential; that is, only constant potential is needed. Thus, the theorem holds. The inductive case for (IP:App-Poly) can be proved in the same manner.

The next case we consider is (IP:Case-Sum):

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

Applying the inductive hypothesis to the two premises, we obtain that both e_ℓ and e_r are typable in multivariate AARA. Let P_1 and P_2 be the multivariate annotations for the two premises. We can derive a multivariate annotation for the conclusion by taking $\max\{P_1(i), P_2(i)\}$ for each index/base polynomial $i \in \mathcal{I}(\Gamma \cup \{y : b_{1,2}\})$. Furthermore, it follows from the inductive hypothesis that any variable $v \in \mathrm{dom}(\Gamma) \cup \{y\} \setminus V[x \mapsto y]$ has zero potential in both P_1 and P_2. Consequently, $\max\{P_1(i), P_2(i)\} = \max\{0, 0\} = 0$ if index i involves a size variable from v. This establishes the theorem. (IP:Case-Prod) can be proved in the same fashion.

We next consider (IP:Case-List):

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

Assume that the inductive hypothesis gives us annotations P_0 and P_1 for the two premises of (IP:Case-List). From P_1, we can construct an annotation P_2 over the typing context $\Gamma \cup \{x : L(b)\}$ such that $P_1 = \lhd(P_2)$. It follows from the definition of $\lhd$ that

$$\lhd(P_2)(i, j, k) := \begin{cases} P_2(i, 0_b :: k) + P_2(i, k) & \text{if } j = 0_b; \\ P_2(i, j :: k) & \text{otherwise.} \end{cases} \tag{E.1}$$

We will define P_2 as follows. For any variable $v \in (\mathrm{dom}(\Gamma) \cup \{x\}) \setminus V$, the coefficient of a base polynomial in P_2 that involves v's size variables is set to 0. All the other coefficients are set to the largest coefficient that appears in P_1. Consequently, by construction, every $v \in (\mathrm{dom}(\Gamma) \cup \{x\}) \setminus V$ contains zero potential in P_2. Let M denote the largest coefficient in P_1.

It remains to ascertain that $\lhd(P_2)$ is a subtype of P_1. We will first consider the first clause of (E.1). Suppose that $P_1(i, j, k) > 0$ for some i, j, k. This implies that (i, j, k) does not involve any variables from $(\mathrm{dom}(\Gamma) \cup \{x_1, x_2\}) \setminus V[x \mapsto x_1, x_2]$; otherwise, $P_1(i, j, k) = 0$

due to the inductive hypothesis of the second premise. If $j = 0_b$, then $P_2(i, k) = M$ holds because

$$\begin{aligned}&(i, 0_b, k) \text{ contains size variables of some } v \in (\mathrm{dom}(\Gamma) \cup \{x_1, x_2\}) \setminus V[x \mapsto x_1, x_2]\\ \iff &(i, k) \text{ contains size variables of some } v \in (\mathrm{dom}(\Gamma) \cup \{x\}) \setminus V.\end{aligned}$$

Thus, $\lhd(P_2)(i, j, k) \geq P_1(i, j, k)$ in this case. Conversely, if $j \neq 0_b$, $P_2(i, j :: k)$ is guaranteed to be M. Hence, $\lhd(P_2)(i, j, k) \geq P_1(i, j, k)$ is true in this case as well. Therefore, $\lhd(P_2)$ is indeed a subtype of P_1. Finally, we can easily combine P_2 with P_0 to yield a desirable annotation for the conclusion of (IP:Case-List).

Next is (IP:Rec):

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

Appealing to the inductive hypothesis, we know that e_0 and e_1 can be annotated as

$$\Gamma; P_0 \vdash e_0 : \langle b_2, Q_0 \rangle \qquad y : b, ys : L(b), z : b_2; P_1 \vdash e_1 : \langle b_2, Q_1 \rangle,$$

where P_1 assigns zero potential to z because of e_1 $\mathsf{inhpoly}(\{y, ys\})$. From P_0 and P_1, it is possible to construct a multivariate annotation for the conclusion of (IP:Rec). Furthermore, any variable in $\mathrm{dom}(\Gamma) \setminus V$ stores zero potential. The details of this construction are presented in Lemma 14.

The next case we consider is (IP:Let-Base):

$$\frac{\Delta_1; \Sigma_1; \Gamma_1 \vdash e_1 \ \mathsf{inhpoly}(V_1) \qquad \Delta_2; \Gamma_2, x : b_1 \vdash e_2 \ \mathsf{inhpoly}(V_2)}{\Delta_1 \cup \Delta_2; \Sigma_1 \cup \Gamma_1 \cup \Gamma_2 \vdash \mathsf{let}\ x = e_1 \ \mathsf{in}\ e_2 \ \mathsf{inhpoly}(V_3)} \ (\text{IP:Let-Base})$$

The inductive hypothesis tells us that there exists a multivariate annotation

$$\Gamma_2, x : b; P_2 \vdash e_2 : \langle b_2, Q_2 \rangle.$$

Let P be a multivariate annotation of the entire let-binding that we aim to derive. According to (M:Let), P must satisfy the following two conditions:

1. $\Sigma_1; \Gamma_1; \pi_0^{\Gamma_1}(P) \vdash e_1 : \langle b_1, \pi_0^{x:b_1}(P_2) \rangle$ holds under the cost metric of the running time;
2. For all $i \neq 0_b \in \mathcal{I}(\Gamma_2)$, we have $\Sigma_1; \Gamma_1; \pi_i^{\Gamma_1}(P) \vdash e_1 : \langle b_1, \pi_i^{x:b_1}(P_2) \rangle$ under the cost-free metric.

We will now conduct case analysis on whether $x \in V_2$. Assume $x \notin V_2$. It follows from the inductive hypothesis that x contains zero potential in P_2. As a result, $\pi_0^{x:b_1}(P_2)$ in the first condition above is essentially constant potential. Applying the inductive hypothesis to e_1, we obtain a multivariate annotation of e_1 under the cost metric of the running time. If the output of e_1 in this annotation contains less potential than $\pi_0^{x:b_1}(P_2)$, we can always inject constant potential into the annotation of e_1. The resulting annotation will serve as a suitable $\pi_0^{\Gamma_1}(P)$. Likewise, $\pi_i^{x:b_1}(P_2)$ in the second condition above, where $i \neq 0_b$, is constant potential. Therefore, a suitable $\pi_i^{\Gamma_1}(P)$ can be constructed (note that the second condition concerns the cost-free metric).

Lastly, we need to ensure that any $v \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2) \setminus V_3$ has zero potential in P. If $v \in \mathrm{dom}(\Gamma_2) \setminus V_2$, since $\pi_i^{\Gamma_1}(P)$ will be constant potential, it is impossible for any base polynomial with a size variable of v to have a non-zero coefficient. The same reasoning applies to the case of $v \in \mathrm{dom}(\Gamma_1) \setminus V_1$.

Conversely, if $x \in V_2$, e_2 demands potential from $x = e_1$, meaning that we need a multivariate annotation for e_1 with some potential available in the output of e_1. Although the theorem gives us *some* multivariate annotation of e_1, we have no guarantee that the output of e_1 contains a desired arbitrary amount of potential. This is where Theorem 9 comes in. By Theorem 9, we can derive a cost-free annotation of e_1 such that its output stores a desirable amount of potential. Finally, summing this cost-free annotation with the annotation given by Theorem 6, we obtain a suitable $\pi_0^{\Gamma_1}(P)$ in the first condition above. Regarding $\pi_i^{\Gamma_1}(P)$ for $i \neq 0_b$ in the second condition, again, we resort to Theorem 9.

Lastly, we need to ensure that any $v \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2) \setminus V_3$ contains zero potential in P. As we assume $x \in V_2$, we have $V_3 = \mathrm{dom}(\Gamma_1) \cup (V_2 \setminus \{x\})$. Therefore, it is guaranteed that $v \in \mathrm{dom}(\Gamma_2) \setminus V_2$. Due to the inductive hypothesis of the theorem on e_2, v contains zero potential in P_2. Thus, for any $i \in \mathcal{I}(\Gamma)$ that involves a size variable of v, $\pi_i^{x:b_1}(P_2)$ is essentially constant potential. Therefore, $\pi_i^{\Gamma_1}(P)$ will be constant potential as well. Consequently, v will contain zero potential in P, which is the annotation of the whole let-binding.

The final inductive case we consider is (IP:Share-Base):

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

By the inductive hypothesis, the premise of (IP:Share-Base) can be assigned a multivariate annotation. Using this, we can build a multivariate annotation for the conclusion by merging the coefficients for those base polynomials that mention either x_1 or x_2. Moreover, if $x \notin V$, then $x_1, x_2 \notin V[x \mapsto x_1, x_2]$. According to the inductive hypothesis, both x_1 and x_2 store zero potential. As a result, in the multivariate annotation of the rule's conclusion, x stores zero potential as well. This establishes the theorem.

The inductive cases for (IP:Let-Arrow) and (IP:Share-Arrow) are straightforward to prove since they do not affect base-type variables; hence, we will not formally present their proof. Likewise, it is immediate to prove (IP:Weaken-Base). This concludes the proof. ◀

▶ **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.

Proof. Like in the above proof of Theorem 6, this proof will proceed by induction on $\Delta; \Gamma \vdash e$ $\mathsf{inhpoly}(V)$. Although this document does not present inference rules for the cost-free annotations in multivariate AARA, the absence of their formal presentation should not affect the typability proof.

As before, nothing interesting happens in the base cases: (IP:Base), (IP:Arrow), (IP:Unit), and (IP:Nil). Likewise, it is straightforward to prove the inductive cases of (IP:SumL), (IP:SumR), and (IP:Pair).

By contrast, (IP:Cons) is nontrivial:

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

Suppose that we would like $x_1 :: x_2$ to be annotated with Q. If Q is univariate, it is clear that there exists P such that

$$x_1 : b, x_2 : L(b); P \vdash x_1 :: x_2 : \langle L(b), Q \rangle$$

and more importantly, P and Q only differ in lower-degree terms. This is because $\vec{q}$ and $\lhd(\vec{q})$ have the same coefficient of the maximum degree for any potential vector $\vec{q}$. If Q is multivariate, the proof is more complicated and is deferred to Lemma 15.

As before, it is straightforward to prove (IP:Const) and (IP:Poly).

We next consider (IP:App-Const):

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

From the inductive hypothesis of x_1 const, we know that the annotation of x_1's input is identical to that of x_1's output when restricting our attention to base polynomials of degree d. Therefore, the claim holds. (IP:App-Poly) can be proved straightforwardly.

Next is (IP:Case-Sum):

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

From the inductive hypothesis, we have P_1 and P_2 satisfying

$$\Gamma, y : b_1; P_1 \vdash e_\ell : \langle b, Q \rangle \qquad \Gamma, y : b_2; P_2 \vdash e_r : \langle b, Q \rangle.$$

To construct a desirable multivariate annotation for the conclusion, we simply need to integrate P_1 and P_2 by taking the maximum coefficient for each base polynomial. (IP:Case-Prod) can be proved similarly.

Next, we consider (IP:Case-List):

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

Lemma 16 provides details of how to construct an annotation for $\mathsf{case}\ x\ \{[] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\}$.

The next inductive case is (IP:Rec):

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

Suppose that the inductive hypothesis of the second premise yields

$$y : b, ys : L(b), z : b_2; P_1 \vdash e_1 : \langle b_2, Q \rangle,$$

where Q $\mathsf{uniform}(d, n)$ and P_1 $\mathsf{uniform}(d, n, \{y, ys\})$. It follows that $\pi_0^{z:b_2}(P) - Q$, which will be paid by resource-polymorphic recursion, has a strictly lower degree than d. Therefore, resource-polymorphic recursion will derive a multivariate annotation for z such that it has the annotation $\pi_0^{z:b_2}(P) - Q$. To formally prove this, we should have performed strong induction on d as well as structural induction on $\Delta; \Sigma; \Gamma \vdash e$ $\mathsf{inhpoly}(V)$. However, since it might make the proof overly complicated and thereby confuse the readers, we decided to hide this detail until now. Note that if $d = 0$, the theorem clearly holds. If we only demand $\Sigma; \Gamma \vdash e : b$ to have constant potential in the output, it is immediate to type e's context under the cost-free metric.

Assume that resource-polymorphic recursion yields $P_{2,i}$ for $i \in \mathcal{I}(\{y : b, ys : L(b)\})$ defined by

$$\Gamma, x : L(b); P_{2,0} \vdash e : \langle b_2, \pi_0^{z:b_2}(P) - Q\rangle \qquad \text{if } i = 0;$$
$$\Gamma, x : L(b); P_{2,i} \vdash e : \langle b_2, \pi_i^{z:b_2}(P)\rangle \qquad \text{otherwise,}$$

where e refers to the entire primitive recursion. Note that all of $\pi_0^{z:b_2}(P) - Q$ and $\pi_i^{z:b_2}(P)$ for any $i \neq 0$ have degrees lower than d; hence, the existence of their annotation can be proved by strong induction on d. Our goal is to build an annotation P for e from P_1 and $P_{2,i}$. The details of P's construction are provided in Lemma 17.

Next is (IP:Let-Base):

$$\frac{\Delta_1; \Sigma_1; \Gamma_1 \vdash e_1 \ \mathsf{inhpoly}(V_1) \qquad \Delta_2; \Gamma_2, x : b_1 \vdash e_2 \ \mathsf{inhpoly}(V_2)}{\Delta_1 \cup \Delta_2; \Sigma_1 \cup \Gamma_1 \cup \Gamma_2 \vdash \mathsf{let}\ x = e_1 \ \mathsf{in}\ e_2 \ \mathsf{inhpoly}(V_3)} \ (\text{IP:Let-Base})$$

Suppose that the inductive hypothesis of the second premise yields

$$\Gamma_2, x : b_1; P_2 \vdash e_2 : \langle b_2, Q\rangle,$$

where Q $\mathsf{uniform}(d, n)$. As we currently work with the cost-free metric, the premises of (M:Let) can be simplified to

$$\forall i \in \mathcal{I}(\Gamma_2).\Sigma_1; \Gamma_1; \pi_i^{\Gamma_1}(P) \vdash e_1 : \langle b_1, \pi_i^{x:b_1}(P_2)\rangle \tag{E.2}$$

under the cost-free metric. Here, P is a multivariate annotation of the whole let-binding—it is what we aim to derive in this proof.

We now conduct case analysis. If $x \in V_2$, it is fairly easy to establish the claim. First of all, we fix $i \in \mathcal{I}(\Gamma_2)$. From $\pi_i^{x:b_1}(P_2)$, the inductive hypothesis allows us to create a suitable $\pi_i^{\Gamma_1}(P)$ that satisfies (E.2). If $\pi_i^{x:b_1}(P_2)$ is not uniform at the maximum degree (this is required by the theorem), we can easily create a uniform annotation that is a subtype of $\pi_i^{x:b_1}(P_2)$. It remains to ensure P $\mathsf{uniform}(d, n, V_3)$. The proof of this case is identical to the proof of the next case; hence, we omit it.

Next, assume $x \notin V_2$. As above, we can create a multivariate annotation for the whole let-binding. It remains to ensure that P is a uniform annotation. Let v be a variable drawn from $\mathrm{dom}(\Gamma_1 \cup \Gamma_2) \setminus V_3$, where $V_3 = V_1 \cup V_2$ due to the assumption $x \notin V_2$. If $v \in \mathrm{dom}(\Gamma_2) \setminus V_2$, the inductive hypothesis of e_2 already implies P_2 $\mathsf{uniform}(d, n, V_2)$. It is easy to see that the first condition of Definition 8 holds for P with respect to variable v.

To establish the third condition (and also the second condition) of Definition 8, consider

$$i = (h, g) \in \mathcal{I}(\Gamma_2 \setminus \{v : b\}) \times \mathcal{I}(\{v : b\}),$$

where $h = 0_{\Gamma_2 \setminus \{v:b\}}$ and $\mathsf{deg}(g) = d$. Due to the inductive hypothesis P_2 $\mathsf{uniform}(d, n, V_2)$, $\pi_i^{x:b_1}(P_2)$ must essentially be constant potential of n. Hence, $\pi_i^{\Gamma_1}(P)$ in (E.2) is constant potential of n as well. Therefore, $P(0_{\Gamma_1}, i) = n$ holds, thereby establishing the third condition of P $\mathsf{uniform}(d, n, V)$. The second condition of P $\mathsf{uniform}(d, n, V)$ can be established by the same reasoning. This proof is also applicable to the previous case, where $x \in V_2$ and $v \in \mathrm{dom}(\Gamma_2) \setminus V_2$.

Lastly, let us consider the final case of the case analysis: $x \notin V_2$ and $v \in \mathrm{dom}(\Gamma_1) \setminus V_1$. Due to the inductive hypothesis P_2 $\mathsf{uniform}(d, n, V_2)$, $\pi_i^{x:b_1}(P_2)$ has degree at most d. Therefore, it follows from the inductive hypothesis of e_1 that, for any base polynomial r in $\pi_i^{\Gamma_1}(P)$ with a non-zero coefficient, r's projection on v must have degree at most d. This establishes the first condition of P $\mathsf{uniform}(d, n, V)$ with respect to v.

Furthermore, if $i \neq 0_{\Gamma_2}$, then $\pi_i^{x:b_1}(P_2)$ has degree at most $d-1$. This means that $\pi_i^{\Gamma_1}(P)$ cannot have a base polynomial whose projection on v has degree d. It can only be degree $d-1$ at largest. Consequently, we have established the second condition of P $\mathsf{uniform}(d, n, V)$ with respect to v.

Finally, consider $i = 0_{\Gamma_2}$. $\pi_0^{x:b_1}(P_2)$ satisfies $\mathsf{uniform}(d, n)$. Therefore, by the inductive hypothesis of e_1, we have $\pi_0^{\Gamma_1}(P)$ $\mathsf{uniform}(d, n, V_1)$. This establishes the third condition of Definition 8.

Finally, we will discuss (IP:Share-Base):

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

Assume the inductive hypothesis of the premise yields

$$\Gamma, x_1 : b, x_2 : b; P \vdash e : \langle b_2, Q \rangle,$$

where Q is specified by a user and satisfies Q $\mathsf{uniform}(d, n)$. From P, we can easily obtain a multivariate annotation P' for the context $\Gamma \cup \{x : b\}$. If $x \in V$, the theorem is true for P'. Conversely, if $x \notin V$, we need to establish $\pi_0^{x:b}(P')$ $\mathsf{uniform}(d, n)$. Although the inductive hypothesis gives us $\pi_0^{x:b}(P)$ $\mathsf{uniform}(d, n)$, it is not always the case that $\pi_0^{x:b}(P')$ $\mathsf{uniform}(d, n)$ holds. To circumvent this problem, we impose the restriction that no variable sharing is permitted on those variables that are outside V (or their constituent variables derived by pattern matching).

The remaining cases (i.e. (IP:Let-Arrow), (IP:Share-Arrow), and (IP:Weaken-Base)) are immediate to prove. This concludes the proof. ◀

▶ **Lemma 14** (Construction of a multivariate annotation for primitive recursion). *Consider a primitive recursion e of the form*

$$\Sigma; \Gamma, x : L(b) \vdash \mathsf{rec}\ x\ \{[\,] \hookrightarrow e_0 \mid (y :: ys)\ \mathsf{with}\ z \hookrightarrow e_1\} : b_2.$$

Suppose we have P_i, Q_i for $i \in \{0, 1\}$ such that

$$\Gamma; P_0 \vdash e_0 : \langle b_2, Q_0 \rangle \qquad y : b, ys : L(b), z : b_2; P_1 \vdash e_1 : \langle b_2, Q_1 \rangle$$

under the cost metric of the running time. Here, z contains zero potential in P_1. Then there exists a multivariate annotation R such that $\Sigma; \Gamma, x : L(b); R \vdash e : \langle b_2, Q_1 \rangle$ holds under the cost metric of the running time.

Furthermore, assume $\Delta; \Sigma; \Gamma \vdash e_0$ $\mathsf{inhpoly}(V)$. For any $v \in dom(\Gamma) \setminus V$, if v contains zero potential in P_0, then v contains zero potential in R as well.

Proof. Using general recursion, the primitive recursion e can be written as

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

Here, ys_1 is used in the recursive call, and ys_2 is (possibly) used inside e_1.

It is safe to assume that the potential for $x : L(b)$ and that for Γ are "completely separated" in R. In other words, for any index $(i_x, i_\Gamma) \in \mathcal{I}(x : b) \times \mathcal{I}(\Gamma)$, if $i_x \neq 0_b \wedge i_\Gamma \neq 0_\Gamma$, then $R(i_x, i_\Gamma) = 0$. For example, if we have some variable $(u : L(\mathbf{1})) \in \Gamma$, then R never contains a factor like $|x| \cdot |u|$; i.e. multiplication of size variables from $x : L(b)$ and size variables from

$(u : L(\mathbf{1})) \in \Gamma$. This is a reasonable assumption because (i) variables in Γ are each used only once (namely, the very first iteration of the primitive recursion) and (ii) z is assumed to contain zero potential in P_1. Therefore, intuitively, the multivariate polynomial function represented by R should have the shape $p_x(|x|) + p_\Gamma(|\Gamma|)$, where p_x, p_Γ are some multivariate polynomial functions and $|\cdot|$ is the set of size variables in the input expression, instead of $p_x(|x|) + p_\Gamma(|\Gamma|) + p_{x,\Gamma}(|x|, |\Gamma|)$ which "mixes" the potential from x and the potential from Γ.

In the second branch of e, $x = (y :: ys)$ is annotated with $\pi_0^{x:L(b)}(R)$, and Γ is annotated with $\pi_0^\Gamma(R)$. For $\pi_0^{x:L(b)}(R)$, it becomes $\lhd(\pi_0^{x:L(b)}(R))$ as a result of pattern matching on x. Because ys_1 is used in the recursive call, it ought to be supplied with the same potential as $x : L(b)$; i.e. $\pi_0^{x:L(b)}(R)$. The remaining potential for y and ys_2, which is given by $\lhd(\pi_0^{x:L(b)}(R)) - \eta_0^{\{y:b, ys:L(b)\}}(\pi_0^{x:L(b)}(R))$, should be equal to Q_1.

Therefore, our goal is to find a suitable $\pi_0^{x:L(b)}(R)$ for Q_1. After this step, $\pi_0^{x:L(b)}(R)$ is combined with P_0, yielding a desirable R. Fortunately, as Q_1 does not need potential from z, we need not be concerned about it.

Let A be a multivariate annotation for $x : L(b)$. A represents $\pi_0^{x:L(b)}(R)$, and we will now work out what A should be. $\lhd(A)$, which is over the typing context $\{y : b, ys : L(b)\}$, is given as

$$\lhd(A)(i, j) := \begin{cases} A(0_b :: j) + A(j) & \text{if } i = 0_b; \\ A(i :: j) & \text{otherwise.} \end{cases}$$

If $i = 0_b$, we have

$$\begin{aligned} \lhd(A)(0_b, j) - A(j) &= A(0_b :: j) + A(j) - A(j) \\ &= A(0_b :: j), \end{aligned}$$

and this should be equal to $\pi_0^{\{y:b, ys:L(b)\}}(Q_1)(0_b, j)$. If $i \neq 0_b$, we should have $\lhd(A)(i, j) = A(i :: j) = \pi_0^{\{y:b, ys:L(b)\}}(Q_1)(i, j)$. In summary, we have

$$A(i :: j) := \pi_0^{\{y:b, ys:L(b)\}}(Q_1)(i, j) \tag{E.3}$$

regardless of whether $i = 0_b$ or not. It is clear that, given Q_1, such A exists. As a sanity check for (E.3), we can check whether it is correct when Q_1 is univariate (and hence A is univariate). However, we will omit the details of the sanity check.

Finally, if $v \in \mathrm{dom}(\Gamma) \setminus V$ contains zero potential in P_0, v also has zero potential in R constructed as above. This is essentially because $\pi_0^{x:L(b)}(R)$ and P_0 are completely separated in R. Note that the assumption that $v \in \mathrm{dom}(\Gamma) \setminus V$ contains zero potential in P_0 is true due to Theorem 6. However, it is treated as an assumption rather than a fact in the present lemma. This concludes the proof. ◀

▶ **Lemma 15** (Preservation of uniformity by list constructors). *Consider a typing judgment $x_1 : b, x_2 : L(b) \vdash x_1 :: x_2 : L(b)$. Fix a multivariate annotation Q of degree d for $x_1 :: x_2$ such that Q $\mathsf{uniform}(d, n)$. There exists a cost-free multivariate annotation P such that*

- $x_1 : b, x_2 : L(b); P \vdash x_1 :: x_2 : \langle L(b), Q \rangle$;
- P $\mathsf{uniform}(d, n, \emptyset)$.

Proof. P is given by $\lhd(Q)$, which is defined as

$$P(i, j) := \begin{cases} Q(0_b :: j) + Q(j) & \text{if } i = 0_b; \\ Q(i :: j) & \text{otherwise.} \end{cases} \tag{E.4}$$

Here, $i \in \mathcal{I}(\{x_1 : b\})$ and $j \in \mathcal{I}(\{x_2 : L(b)\})$.

We will now prove P $\mathsf{uniform}(d, n, \emptyset)$. Firstly, due to the definition of P in (E.4), if the maximum degree of Q is d, so is the maximum degree of P. That is, $P(i, j) > 0$ implies $\mathsf{deg}(i) + \mathsf{deg}(j) \leq d$. Hence, the first and second conditions for P $\mathsf{uniform}(d, n, \emptyset)$ (Definition 8) are met.

Secondly, in P, any base polynomial (i) that has degree d and (ii) that only involves size variables of either x_1 or x_2 (but not both) must have coefficient n. To see this, let us first consider x_2. Fix an arbitrary $j \in \mathcal{I}(\{x_2 : L(b)\})$ such that $\mathsf{deg}(j) = d$. We get

$$\begin{aligned} P(0_b, j) &= Q(0_b :: j) + Q(j) \\ &= 0 + Q(j) \\ &= Q(j). \end{aligned}$$

In the second line, $0_b :: j$ would have degree $d + 1$, which exceeds the highest degree of Q. Hence, we must have $Q(0 :: j) = 0$. As a result, we obtain $P(0_b :: j) = Q(j)$, where both terms have the same degree, namely d.

Next, let us consider x_1. Fix an arbitrary $i \in \mathcal{I}(\{x_1 : b\})$ such that $\mathsf{deg}(i) = d$. This gives

$$\begin{aligned} P(i, 0_{L(b)}) &= Q(i :: 0_{L(b)}) \\ &= 0, \end{aligned}$$

where the last line follows from the fact that $\mathsf{deg}(i :: 0_{L(b)})$ exceeds d and hence $Q(i :: 0_{L(b)}) = 0$ must hold. Thus, (M:Sup) allows us to increase $P(i, 0_{L(b)})$ such that the claim holds. Finally, the case where $i = 0_b$ is equivalent to the case where $d = 0$, and it is immediate to prove this case. This concludes the proof. ◀

▶ **Lemma 16** (Preservation of uniformity by list destructors). *Consider* $e \equiv \mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid (x_1 :: x_2) \hookrightarrow e_1\}$, *where* $\Delta; \Sigma; \Gamma, x : L(b) \vdash e\ \mathsf{inhpoly}(V)$. *Assume that the typing judgments of* e_0 *and* e_1 *are*

$$\Gamma \vdash e_0 : b_2 \qquad \Gamma, x : b, x_2 : L(b) \vdash e_1 : b_2,$$

where b *does not contain list types inside; that is,* $L(b)$ *is a non-nested list type. Additionally, suppose we are given* P_0 *and* P_1 *such that*

$$\Gamma; P_0 \vdash e_0 : \langle b_2, Q \rangle \qquad \Gamma, x_1 : b, x_2 : L(b); P_1 \vdash e_1 : \langle b_2, Q \rangle,$$

where Q $\mathsf{uniform}(d, n)$, P_0 $\mathsf{uniform}(d, n, V \setminus \{x\})$, *and* P_1 $\mathsf{uniform}(d, n, V[x \mapsto x_1, x_2])$. *Then there exists a multivariate annotation* P *for the entire* e *such that (i) the output is annotated with* Q *and (ii)* P $\mathsf{uniform}(d, n, V)$ *holds.*

Proof. From P_1, we will construct an annotation P_2 over the typing context $\Gamma \cup \{x : L(b)\}$. P_2 should satisfy $\lhd(P_2) = P_1$ and P_2 $\mathsf{uniform}(d, n, V)$. $\lhd(P_2)$ is defined as

$$\lhd(P_2)(i, j, k) := \begin{cases} P_2(i, 0_b :: k) + P_2(i, k) & \text{if } j = 0_b; \\ P_2(i, j :: k) & \text{otherwise.} \end{cases}$$

Because $L(b)$ is a non-nested list type by assumption, j is always 0_b. If we set

$$P_2(i, k) := P_1(i, 0_b, k), \tag{E.5}$$

we obtain

$$\begin{aligned}\lhd(P_2)(i, 0_b, k) &= P_2(i, 0_b :: k) + P_2(i, k) && \text{by definition}\\ &= P_1(i, 0_b, 0_b :: k) + P_1(i, 0_b, k) && \text{by (E.5)}\\ &\geq P_1(i, 0_b, k).\end{aligned}$$

Therefore, $\lhd(P_2)$ is a subtype of P_1; hence, $\lhd(P_2)$ can be converted to P_1 by (M:SUB).

It remains to check that P_2 $\mathsf{uniform}(d, n, V)$ holds. Let v be a variable from $\mathrm{dom}(\Gamma) \cup \{x\} \setminus V$. We now conduct case analysis on v. Suppose $v \not\equiv x$; that is, $x \in \mathrm{dom}(\Gamma)$. This gives

$$\begin{aligned}P_2(i, k) > 0 &\implies P_1(i, 0_b, k) > 0 && \text{by (E.5)}\\ &\implies \mathsf{deg}(i) < d \vee (\mathsf{deg}(i) = d \wedge \mathsf{deg}(k) = 0) && \text{by the } \mathsf{uniform} \text{ assumption.}\end{aligned}$$

Therefore, the first and second conditions of Definition 8 are satisfied. Further, if a base polynomial in P_2 only contains size variables of v and has degree d, its coefficient is n due to (E.5) and the assumption P_1 $\mathsf{uniform}(d, n, V[x \mapsto x_1, x_2])$. Hence, the third condition of Definition 8 is true.

Conversely, suppose $v \equiv x$. Consider $k \in \mathcal{I}(\{x : L(b)\})$. This yields

$$\begin{aligned}P_2(i, k) > 0 &\implies P_1(i, 0_b, k) > 0 && \text{by (E.5)}\\ &\implies \mathsf{deg}(k) < d \vee (\mathsf{deg}(k) = d \wedge \mathsf{deg}(i) = 0) && \text{by the } \mathsf{uniform} \text{ assumption.}\end{aligned}$$

This satisfies the first and second conditions of Definition 8. Furthermore, for any $k \in \mathcal{I}(\{x : L(b)\})$, if $\mathsf{deg}(k) = d$, we have $P_2(0_\Gamma, k) = n$. This is because $P_1(0_\Gamma, 0_b, k) = n$ holds due to the inductive hypothesis of P_1. Therefore, P_2 $\mathsf{uniform}(d, n, V)$ holds as required.

Finally, we can merge P_2 and $\eta_0^{\Gamma \cup \{x:L(b)\}}(P_0)$ into P that satisfies P $\mathsf{uniform}(d, n, V)$. This concludes the proof. ◀

The following lemma concerns the construction of a multivariate annotation for a primitive recursion and is more general than Lemma 14.

▶ **Lemma 17** (Resource annotation for resource-polymorphic recursion). *Consider a primitive recursion of the form*

$$f := \lambda x, \Gamma.\mathsf{case}\ x\ \{[\,] \hookrightarrow e_0 \mid y :: ys \hookrightarrow \mathsf{let}\ z = f\ ys\ \Gamma\ \mathsf{in}\ e_1\}.$$

Let P_i and Q_i for $i \in \{0, 1\}$ be multivariate annotations that satisfy

$$\Gamma; P_0 \vdash e_0 : \langle b_2, Q_0 \rangle \qquad y : b, ys : L(b), z : b_2; P_1 \vdash e_1 : \langle b_2, Q_1 \rangle.$$

For each $i \in \mathcal{I}(\{y : b, ys : L(b)\})$, a multivariate annotation $P_{2,i}$ is defined by

$$\begin{aligned}&\Gamma, x : L(b); P_{2,0} \vdash f\ x\ \Gamma : \langle b_2, \pi_0^{z:b_2}(P_1) - Q_1 \rangle && \textit{if } i = 0;\\ &\Gamma, x : L(b); P_{2,i} \vdash f\ x\ \Gamma : \langle b_2, \pi_i^{z:b_2}(P_1) \rangle && \textit{otherwise,}\end{aligned}$$

where $f\ x\ \Gamma$ denotes the whole primitive recursion. From P_1 and $P_{2,i}$ for $i \in \mathcal{I}(\{y : b, ys : L(b)\})$, it is possible to build a multivariate annotation P for the entire primitive recursion such that its final output is annotated with Q_1.

Furthermore, assume $\Delta; \Gamma \vdash e_0$ $\mathsf{inhpoly}(V)$. If we have

- P_0 $\mathsf{uniform}(d, n, V)$*, where $d > 0$, and*
- $P_{2,i}$ $\mathsf{uniform}(d - 1, n_i, V \cup \{x\})$ *for each $i \in \mathcal{I}(\{y : b, ys : L(b)\})$ and some $n_i \in \mathbb{Q}_{>0}$*

then P $\mathsf{uniform}(d, n, V \cup \{x\})$ *is true as well.*

Proof. It is given that each recursive call needs P_1 much potential. This information does not directly tell us the total amount of potential. This is because P_1 involves z, which is the result of the recursive call, and it is not immediately obvious how to relate z back to x, which is the original input. Hence, the first task is to work out a multivariate annotation for each recursive call in terms of $\{y, ys\} \cup \Gamma$ (instead of $\{y, ys, z\}$).

As stated in the theorem, let P be the annotation for the entire primitive recursion that we aim to construct in this proof. In the second branch of the primitive recursion, as a result of pattern matching in x, P becomes $\lhd(P)$ whose domain is $\Gamma \cup \{y : b, ys : L(b)\}$.

Base polynomials in P_1 can be classified into three categories:

- Base polynomials that only concern $\{y, ys\}$. The coefficients of these polynomials are given by $\pi_0^{\{y:b, ys:L(b)\}}(P_1)$.
- Base polynomials that only concern z. The coefficients of these base polynomials are given by $\pi_0^{z:b_2}(P_1)$. Out of this potential, Q_1 will be paid by P because P is exactly what we are trying to establish at the moment and we are allowed to reuse P as an input annotation for Q_1. The remaining potential, $\pi_0^{z:b_2}(P_1) - Q_1$, will be paid by $P_{2,0}$ in the assumption.
- Base polynomials that concern both $\{y, ys\}$ and z. We can replace z with $\{y, ys\}$ in the representation of these base polynomials by using $P_{2,i}$, where $i \neq 0 \in \mathcal{I}(\{y : b, ys : L(b)\})$.

Let ζ be $\Gamma \cup \{y : b, ys : L(b)\}$. The total potential needed by each recursive call is then given by

$$\eta_{0_\Gamma}^{\zeta}(\pi_0^{\{y:b, ys:L(b)\}}(P_1)) + \eta_{0_b}^{\zeta}(P) + \sum_{i \in \mathcal{I}(\{y:b, ys:L(b)\})} P_{2,i} \cdot i,$$

where $P_{2,i} \cdot i$ denotes the product of $P_{2,i}$, whose typing context is $\Gamma \cup \{x : L(b)\}$, and base polynomial i, whose typing context is $\{y : b, ys : L(b)\}$. Notice that the typing context of $P_{2,i}$ and that of i overlap—they have $\{ys : L(b)\}$ in common.

Because this should be equal to $\lhd(P)$, we obtain

$$\lhd(P) - \eta_{0_b}^{\zeta}(P) = \eta_{0_\Gamma}^{\zeta}(\pi_0^{\{y:b, ys:L(b)\}}(P_1)) + \sum_{i \in \mathcal{I}(\{y:b, ys:L(b)\})} P_{2,i} \cdot i. \tag{E.6}$$

From the right hand side, we can construct a desirable P such that (E.6) holds, as done in the proof of Lemma 14. Specifically, $\lhd(P)$ is defined as

$$\lhd(P)(i, j, k) := \begin{cases} P(i, 0_b :: k) + P(i, k) & \text{if } j = 0_b; \\ P(i, j :: k) & \text{otherwise.} \end{cases}$$

Hence, $\lhd(P) - \eta_0^{\zeta}(P)$ is given as follows. If $j = 0_b$, then

$$\begin{aligned} (\lhd(P) - \eta_0^{\zeta}(P))(i, j, k) &= \lhd(P)(i, 0_b, k) - \eta_0^{\zeta}(P)(i, 0_b, k) \\ &= P(i, 0_b :: k) + P(i, k) - P(i, k) \\ &= P(i, 0_b :: k). \end{aligned}$$

Conversely, if $j \neq 0_b$, then

$$\begin{aligned} (\lhd(P) - \eta_0^{\zeta}(P))(i, j, k) &= \lhd(P)(i, j, k) - \eta_0^{\zeta}(P)(i, j, k) \\ &= P(i, j :: k) + P(i, k) - 0 \\ &= P(i, j :: k). \end{aligned}$$

In conclusion, we obtain

$$(\lhd(P) - \eta_0^\zeta(P))(i, j, k) = P(i, j :: k)$$

regardless of whether $j = 0_b$ or not. This must be equal to the right hand side of (E.6), and it is clear that such P is guaranteed to exist. In addition, it must be possible to extract Q_0 from P when x is the empty list. This yields P's definition:

$$\begin{aligned} P(i, j :: k) &:= \eta_{0_\Gamma}^\zeta(\pi_0^{\{y:b, ys:L(b)\}}(P_1))(i, j, k) + \sum_{r \in \mathcal{I}(\{y:b, ys:L(b)\})} (P_{2,r} \cdot r)(i, j, k) \\ P(i, 0_b) &:= Q_0(i). \end{aligned} \tag{E.7}$$

It remains to check that P $\mathsf{uniform}(d, n, V)$ holds. Consider $v \in \mathrm{dom}(\Gamma) \setminus V$. We will first prove the first and second conditions of Definition 8 for P. If $\mathsf{deg}(i) \geq d > 0$ (which is a stronger condition than $\mathsf{deg}(i) > d$), it gives

$$\begin{aligned} P(i, j :: k) &= \eta_{0_\Gamma}^\zeta(\pi_0^{\{y:b, ys:L(b)\}}(P_1))(i, j, k) + \sum_{r \in \mathcal{I}(\{y:b, ys:L(b)\})} (P_{2,r} \cdot r)(i, j, k) \\ &= \sum_{r \in \mathcal{I}(\{y:b, ys:L(b)\})} (P_{2,r} \cdot r)(i, j, k) \\ &= 0. \end{aligned}$$

Here, the second line follows from the definition of the extension operator $\eta_{0_\Gamma}^\zeta$. Because we extend $\pi_0^{\{y:b, ys:L(b)\}}(P_1)$ with 0_Γ, we have $\eta_{0_\Gamma}^\zeta(\pi_0^{\{y:b, ys:L(b)\}}(P_1))(i, j, k) = 0$ whenever $i \neq 0_\Gamma$ (which is the case since $\mathsf{deg}(i) \geq d > 0$). With regard to the third line above, due to the assumption $P_{2,r}$ $\mathsf{uniform}(d-1, n_r, V \cup \{x\})$, $(P_{2,r} \cdot r)(i, j, k) > 0$ only if $\mathsf{deg}(i) \leq d - 1$. Therefore, if $\mathsf{deg}(i) = d$, we have $P(i, j :: k) = 0$ for any $j :: K$, thereby establishing the second condition of Definition 8.

In the case of $P(i, 0_b)$, if $\mathsf{deg}(i) > d$, we have

$$\begin{aligned} P(i, 0_{L(b)}) &= Q_0(i) \\ &= 0, \end{aligned}$$

where the second line follows from the assumption Q_0 $\mathsf{uniform}(d, n, V)$. Therefore, the first and second conditions of Definition 8 are true for P.

Finally, to prove the third condition of Definition 8, consider $i \in \mathcal{I}(\Gamma)$, where $\mathsf{deg}(i) = d$. This gives

$$\begin{aligned} P(i, 0_{L(b)}) &= Q_0(i) \\ &= n \end{aligned}$$

where the second line follows from the assumption Q_0 $\mathsf{uniform}(d, n, V)$.

In summary, all the three conditions of Definition 8 hold for P. Therefore, P $\mathsf{uniform}(d, n, V \cup \{x\})$ is indeed true. This concludes the proof. ◀