Automatic Amortized Resource Analysis with Regular Recursive Types

Jessie Grosen
jgrosen@cs.cmu.edu
Carnegie Mellon University

David M. Kahn
davidkah@andrew.cmu.edu
Carnegie Mellon University

Jan Hoffmann
jhoffmann@cmu.edu
Carnegie Mellon University

***Abstract*—The goal of automatic resource bound analysis is to statically infer symbolic bounds on the resource consumption of the evaluation of a program. A longstanding challenge for automatic resource analysis is the inference of bounds that are functions of complex custom data structures. This article builds on type-based automatic amortized resource analysis (AARA) to address this challenge. AARA is based on the potential method of amortized analysis and reduces bound inference to standard type inference with additional linear constraint solving, even when deriving non-linear bounds. Such bounds come from resource functions, which are linear combinations of basic functions of data structure sizes that fulfill certain closure properties.**

Previous work on AARA defined resource functions for many data structures such as lists of lists, but left open whether such functions exist for arbitrary data structures. This work answers this question positively by uniformly constructing resource polynomials for algebraic data structures defined by regular recursive types. These functions are a generalization of all previously proposed polynomial resource functions and can be seen as a general notion of polynomials for values of a given recursive type. A resource type system for FPC, a core language with recursive types, demonstrates how resource polynomials can be integrated with AARA while preserving all benefits of past techniques. The article also proposes the use of new techniques useful for stating the rules of this type system succinctly and proving it sound against a small-step cost semantics. First, multivariate potential annotations are stated in terms of free semimodules, substantially abstracting details of the presentation of annotations and the proofs of their properties. Second, a logical relation giving semantic meaning to resource types enables a proof of soundness by a single induction on typing derivations.

I. Introduction

Programming language support for statically deriving resource (or cost) bounds has been extensively studied. Existing techniques encompass manual and automatic resource analyses and are based on type systems [1], [2], [3], deriving and solving recurrence relations [4], [5], [6], or other static analyses [7], [8], [9]. They can derive (worst-case) upper bounds [10], [11] (best-case) lower bounds [12], [13], and relational bounds on the difference of the cost of two programs [14], considering resources like time or memory.

Most automatic techniques focus on bounds that are functions of integers or sizes of simple data structures like lists of integers. One exception is Automatic Amortized Resource Analysis (AARA) [15], [16], [17], which can automatically derive bounds for complex data structures like lists of lists, taking into account the individual lengths of inner lists. As an example, consider the function sort_lefts_list, which extracts only the left injections from its input list and sorts the result. Assume we are interested in the number of cons cells that are created during the evaluation.

```
let sort_lefts_list (l : (int + bool) list) =
  quicksort (filter_map find_left l)
```

RaML [18], an implementation of AARA, is able to derive the exact worst-case bound of n^2+n cons cell creations where n is only the number of left injections in the list. This small example highlights several key qualities of AARA: it is able to tightly analyze tricky recursion patterns, like those that appear in quicksort; it is compositional, easily handling interprocedural code; it produces exact, not asymptotic, polynomial bounds; and it can derive bounds on functions over tree-like data structures that take into account the shape of the data.

AARA for functional programs is based on a type system and type derivations serve as proof certificates for the derived bounds. Type inference is reduced to efficient linear programming and AARA naturally derives bounds on the high-water mark resource use of non-monotone resources like memory, which can become available during the evaluation. The key innovation that enables inference of non-linear bounds with linear programming is the use of a carefully selected set of resource functions that serve as templates for the potential functions used in the physicist's method of amortized analysis.

Despite its benefits, state-of-the-art AARA still has some limitations to its real-world applicability, including its lack of support for general, regular recursive types.[1] As an example, examine the function sort_lefts_tree, similar to the previous function but with lists swapped for *rose trees*:

```
let quicksort : int list → int list = ...
type 'a tree = Tree of 'a * 'a tree list
let filter_map_tree : ('a → 'b option) → 'a tree → 'b list = ...
let sort_lefts_tree (t : (int + bool) tree) =
  quicksort (filter_map_tree find_left t)
```

Rose trees can have arbitrary and variable branching factors, enabled by defining trees and lists of child trees in a nested fashion. Existing AARA systems cannot derive a bound for this function. AARA's inability to derive bounds that are functions of general algebraic data structures poses a real

979-8-3503-3587-3/23/$31.00

[1] We use the term *regular recursive types* to refer to types that may contain non-trivial nested recursion, but where all recursion occurs at base kind.

deficiency. Extending polynomial AARA to handle regular recursive types has been an open problem since it was introduced in 2010 [19]. The core challenge lies in finding an expressive class of potential functions for these types that is still closed under the operations necessary for typing.

We address this longstanding gap by introducing a notion of resource polynomials for regular recursive types that meets the requirements of AARA. We draw inspiration from past approaches, but ultimately adopt a more algebraic view that we believe better follows the structure of types. In particular, the indices that generate the base polynomials match the values they classify nearly exactly. Our resource polynomials are a generalization of all previously proposed polynomial resource functions of AARA [16], [19], [17], [18] and can be seen as a general notion of polynomials for values of a given recursive type. We give the two constructions, *shifting* and *sharing*, which witness resource polynomials' closure under discrete difference and multiplication, respectively; together, they enable AARA's inference of resource bounds using only linear programming. We describe these and other operations as linear maps on free semimodules in order to abstract away some of the tedious details in previous presentations. Finally, we build a type system for a version of FPC (a call-by-value language with recursive types [20]) enriched with resource usage that makes use of these resource polynomials and prove the system sound via a logical-relations argument.

An extended version of this paper contains the full definitions of our logical relations and sharing constructions [21].

II. Overview

To start with, we review AARA (§II-A), detail its potential functions for lists (§II-B), and present the intuition behind our extension to regular recursive types (§II-C).

A. A quick introduction to AARA

AARA is a type-based technique for automatically inferring worst-case cost bounds for programs that manipulate data structures. It uses a formalization of the physicist's method introduced by Tarjan and Sleator [22] to assign potential functions to data structures that can then be used for amortized analysis. The potential available in a given context is then tracked across the program to ensure that the available potential is sufficient to cover the cost of the next transition and the potential of the resulting state.

To automate the physicist's method, AARA defines a set of fixed potential functions for each type. These potential functions have to satisfy certain (closure) properties that enable a smooth integration of potential tracking with the typing rules. This integration is the key to automation, because the potential tracking can be expressed with linear constraints that can be generated in tandem with type checking or inference. These constraints can then be solved by an LP solver, resulting in a final type annotated with a resource bound.

Example: filter_map: To demonstrate the basics of the AARA approach, we build up the motivating example shown in the introduction. As then, say we are interested in the number of cons cell creations as our cost model. To start, consider the standard list function $\mathsf{filter\_map} : (\tau \to \mathsf{option}(\sigma)) \to \mathsf{list}(\tau) \to \mathsf{list}(\sigma)$, which is implemented as follows:

```
let rec filter_map f l = match l with
  | [] → []
  | x :: l' → match f x with
              | Some y → y :: filter_map f l'
              | None → filter_map f l'
```

The evaluation of the expression filter_map f l applies f to each element of l and collects the Some results into the output list. The cost of the evaluation depends on the cost of the higher-order argument f. If we assume that the cost of f is 0, then the cost of filter_map f l is, at worst, the length $|l|$ of the list l. This bound can be expressed by the following type.

$$\mathsf{filter\_map} : (\langle \mathsf{int}^0 + \mathsf{bool}^0, 0\rangle \to \langle \mathsf{option}^0(\mathsf{int}), 0\rangle) \to \langle \mathsf{list}^1(\mathsf{int}^0 + \mathsf{bool}^0), 0\rangle \to \langle \mathsf{list}^0(\mathsf{int}), 0\rangle$$

The type $\langle int^0 + bool^0, 0\rangle \to \langle \mathsf{option}^0(int), 0\rangle$ of the higher-order argument states that the function does not need any input potential and does not assign any potential to its output. The list type $\mathsf{list}^1(int^0 + bool^0)$ expresses that the list argument carries one potential unit per element of the list, reflecting the bound to be proved. The output potential $\langle \mathsf{list}^0(int), 0\rangle$ is zero in this case but is, in general, important for the compositionality of the analysis. To see how the potential of the result can be used consider the following typing:

$$\mathsf{filter\_map} : (\langle \mathsf{int}^1 + \mathsf{bool}^0, 0\rangle \to \langle \mathsf{option}^1(\mathsf{int}), 0\rangle) \to \langle \mathsf{list}^1(\mathsf{int}^1 + \mathsf{bool}^0), 0\rangle \to \langle \mathsf{list}^1(\mathsf{int}), 0\rangle$$

Here the resulting list carries 1 potential unit per element. To cover this additional potential, the input list now has type $\mathsf{list}^1(int^1 + bool^0)$, which expresses 1 potential unit per element and one additional potential unit for each element of the form $\mathsf{inl}\, n$. The type of the higher-order argument expresses that 1 potential unit is necessary if the argument has the form $\mathsf{inl}\, n$ and otherwise none is needed. After the evaluation there is 1 unit left if the result is Some n and 0 otherwise.

The right type annotation for filter_map depends on the context in which the function is used. The general type can be described with abstract annotations and linear constraints:

$$\mathsf{filter\_map} : (\langle \mathsf{int}^{q_1} + \mathsf{bool}^{q_2}, p_0\rangle \to \langle \mathsf{option}^{q_3}(\mathsf{int}), p_0'\rangle) \to \langle \mathsf{list}^{r_1}(\mathsf{int}^{r_2} + \mathsf{bool}^{r_3}), p_1\rangle \to \langle \mathsf{list}^{r_4}(\mathsf{int}), p_1'\rangle$$

$$r_1 \geq p_0 + 1, r_2 \geq q_1, r_3 \geq q_2, p_1 \geq p_1', q_3 + p_0' \geq r_4$$

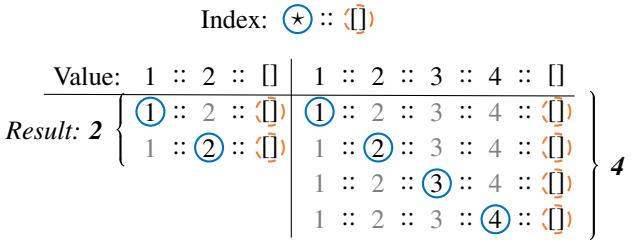
To be clear, this symbolic representation cannot be expressed *within* the type system. However, as part of type inference, this form is derived with the symbolic values as metavariables; the constraints are then solved using linear programming to find a solution that, when substituted in, provides a concrete judgement within the type system. An essential requirement, then, is that the transfer of potential from the list to its head and tail can be expressed with linear constraints. For linear potential functions, this is straightforward since the annotation

of the head is the annotation of the element type and the annotation of the tail is the annotation of the matched list.

B. Potential functions of lists

To go beyond linear potential, polynomial AARA extends the notation $\langle L^{q_1}(A), q_0\rangle$ to $L^{(q_0,q_1,\ldots,q_m)}(A)$, where $\vec{q}$ is a vector of coefficients that specify a polynomial [19]. What is less clear is how to maintain the requirement that only linear constraints come of destructing a list. The answer can be made elegant with a clever choice of basis: the coefficients (q_i) correspond to a basis of binomial coefficients $\binom{n}{i}$, rather than monomials n^i, due to their posession of an *additive shift* function $\triangleleft(q_0,\ldots,q_m) = (q_0+q_1,\ldots,q_{m-1}+q_m,q_m)$. This is a *linear* function that specifies how to preserve potential, i.e., evaluating $\triangleleft(\vec{q})$ on n is equal to evaluating $\vec{q}$ on $n+1$. This concept of a linear shift function turns out to be a key guiding abstraction that guarantees the generation of only linear constraints in the typing rule for pattern matching.

This principle carries over when AARA is extended to multivariate annotations–including terms like $m{\cdot}n$, as might be required when computing the Cartesian product of two lists–but the coefficient vector notation does not. To address this, multivariate AARA introduces the use of *indices* to form a basis of potential functions [17]. Intuitively, they generalize the notion of giving names to "monomials" like $\binom{n}{2}$ or $\binom{n}{3}\binom{m}{2}$. List indices have the form $[i_1,\ldots,i_n]$, where each i_j is an index for the list elements' type. Such a list index refers to counting the number of combinations of elements of the list that match the inner indices. It's perhaps best illustrated with some examples; we'll stick with univariate examples for simplicity's sake, but it is easily extended to the multivariate case. For starters, take the index $[\star] = \star :: \mathsf{nil}$ on lists, which counts the number of ways that an element $\star$ can be followed by nil, i.e., the length of the list. Visually consider evaluating this index on two lists of different lengths:



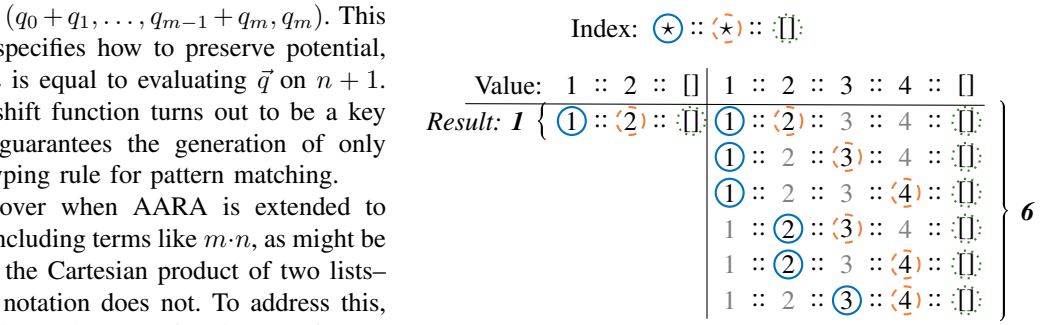
Note that, as demonstrated by the two circles in each evaluation, there are two matches in each: a cons cell, and the ending nil. Let ϕ denote the function that evaluates an index on a value, whose definition we now develop. The critical aspect of computing ϕ is that it can be phrased purely locally in terms of the heads and tails of the index and list elements:

$$\phi_{i::is}(v :: vs) = \phi_i(v)\cdot\phi_{is}(vs) + \phi_{i::is}(vs)$$

Here evaluating ϕ on a cons node first counts the combinations that include the head element, then adds the combinations that don't. From this presentation, an analogous shift function falls out: $\triangleleft(i :: is) = (i, is) + (\star, i :: is)$, where the result is evaluated on (v, vs) given a list $v :: vs$. Note just how similar this is to the definition for binomial coefficients!

As an example of how these indices are used in types, return to the second type of filter_map f we presented, namely $\langle \mathsf{list}^1(\mathsf{int}^1 + \mathsf{bool}^0), 0\rangle \to \langle \mathsf{list}^1(int), 0\rangle$. Expressed using indices, this function requires its argument to have potential $2\cdot[\mathsf{inl}\,\star] + 1\cdot[\mathsf{inr}\,\star]$ and returns a value with potential $1\cdot[\star]$.

Building toward our desire to type quicksort, first consider some evaluations of the index $[\star;\star]$:



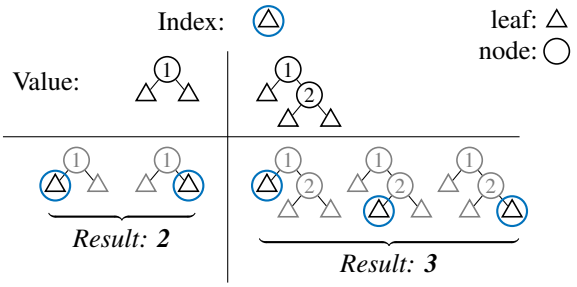
As expected, we find that this index corresponds to $\binom{n}{2}$. Thus, given that we know quicksort has cost n^2, we can express the required potential of its argument using indices as $2\cdot[\star;\star] + 1\cdot[\star]$. Finally, we can consider our original function, sort_lefts_list. Here we can see that it must require an input potential of $2\cdot[\mathsf{inl}\,\star;\mathsf{inl}\,\star] + 2\cdot[\mathsf{inl}\,\star]$–filter_map consumes the $1\cdot[\mathsf{inl}\,\star]$ part of it and passes on the rest to quicksort.

C. Extending to regular recursive types

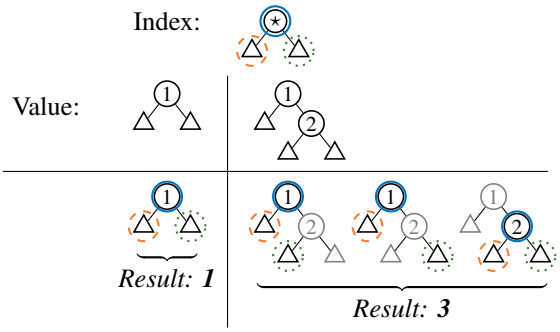
However, these indices do not obviously generalize to regular inductive types. Jost et al. [16] handle potential on regular inductives, but only in the very restricted setting of univariate linear potential, which amounts to just counting constructors. Hoffmann et al. [17] and their successor works handle more expressive potential functions, but don't support regular inductives and treat even just binary trees as lists for potential purposes. Tree indices are identical to list indices, and tree values are just list versions of themselves flattened by a preorder traversal. This results in the combinatorial *structure* of trees being completely lost.

Let's explore a different design. For one, we know we absolutely must preserve some sort of linear shift function. Another hint comes from Hoffmann et al. [17], who observe in passing that their indices for a type τ essentially follow the structure of values of type τ. We find that they were on to something after all. We consider indices that correspond almost *exactly* to the values of the type they describe. To build intuition, we'll first give some examples on specific data types before we get to describing the general case.

Stepping stone: binary trees: We'll start by looking at the case of binary trees. In the following diagrams, tree nodes are circles while leaves are triangles. Consider evaluating the "leaf" index on two different trees:



This counts the number of leaves in the tree, just as the first list index example (consisting of a cons node) counted the number of cons nodes in a list. Now let's look at the next simplest index, a node connecting two leaves:



The evaluation on the right may be confusing at first–isn't there only one subtree that matches the index? The answer may be seen in analogy with the combinatorial evaluation on lists presented earlier: all possible combinations of constructors are considered, subject to the ordering imposed by the index.

These examples are instances of the rules for binary trees, again defined purely locally:

$$\phi_{\triangle}(\triangle) = 1$$

$$\phi_{\overset{i}{\diagup\ \diagdown}}(\triangle) = 0$$

$$\phi_{\triangle}\left(\overset{v}{\diagup\ \diagdown}\right) = \phi_{\triangle}(t_1) + \phi_{\triangle}(t_2)$$

$$\phi_{\overset{i}{\diagup\ \diagdown}}\left(\overset{v}{\diagup\ \diagdown}\right) = \phi_i(v) \cdot \phi_{i_1}(t_1) \cdot \phi_{i_2}(t_2) + \phi_{\overset{i}{\diagup\ \diagdown}}(t_1) + \phi_{\overset{i}{\diagup\ \diagdown}}(t_2)$$

Eureka!: The key insight is to notice this correspondence between the rules for lists and the rules for binary trees:

$$\phi_{i::is}(v :: vs) = \phi_i(v) \cdot \phi_{is}(vs) + \phi_{i::is}(vs)$$

$$\phi_{\overset{i}{\diagup\ \diagdown}}\left(\overset{v}{\diagup\ \diagdown}\right) = \phi_i(v) \cdot \phi_{i_1}(t_1) \cdot \phi_{i_2}(t_2) + \phi_{\overset{i}{\diagup\ \diagdown}}(t_1) + \phi_{\overset{i}{\diagup\ \diagdown}}(t_2)$$

Key Intuition. *To evaluate an index at a constructor, first evaluate it at the immediate constructor, then add that to the evaluation of the original index at all direct children.*

Note that it satisfies our desired properties: it is multivariate through the use of multiplication at the immediate constructor evaluation; it is structure-dependent by evaluating recursively only at direct children; and, critically, it suggests a shift function that exactly mirrors this construction. Having observed that, we will leave it to §III-A to define this formally, but we will at least address one ambiguity in that specification: what are the "direct children" of a constructor?

The prize: rose trees: The direct children of a cons cell or tree node are readily apparent, but they are less obvious for the motivating rose tree. Let us turn to the examples in fig. 1 for intuition. To specify the direct children of a rose tree node, we piggyback off of the list's notion of direct children: a rose tree node's direct child is any node that appears in its list. This notion of pushing the problem of recursive evaluation of the outer type down to the inner type is precisely the solution. Speaking anthropomorphically, the rose tree can identify, in any given list node, the one possible occurrence of a tree (in a cons cell); the list can then use that information to look through the recursive occurences of the list. This intuition is formalized and explained once again in §III-A.

Calling back to our motivating filter_map_tree, specifying a required potential for the same function as the second typing of filter_map is now as simple as $2 \cdot \mathsf{Tree}(\mathsf{inl}\,\star, []) + 1 \cdot \mathsf{Tree}(\mathsf{inr}\,\star, [])$, i.e., $2m + 1n$ where m is the number of nodes with ints and n is the number of nodes with bools. For the overall sort_lefts_tree, it is the similarly natural $2 \cdot \mathsf{Tree}(\mathsf{inl}\,\star, [\mathsf{Tree}(\mathsf{inl}\,\star, [])]) + 2 \cdot \mathsf{Tree}(\mathsf{inl}\,\star, [])$, for much the same reasons as the list case. Incredibly, these indices look nearly as simple as the indices for the equivalent list functions, which we believe is a strong suggestion of elegance.

III. Resource Polynomials

As in previous AARA type systems, *resource polynomials* serve as our language's mechanism to assign potential to typed values. Our core contribution to their theory is a generalization of past systems' bounded-branching tree types to more general algebraic, possibly-mutually recursive types. In this section, we first formally define these potential functions, then give manipulations of them necessary for the type system, continuing our use of running examples to illustrate the definitions.

A. Resource polynomial definitions

Types and values: To show to what exactly resource polynomials assign potential, we first give the types and values over which the resource polynomials are defined in fig. 2. The types presented are standard, save the arrow type–the details of which are irrelevant to the resource polynomials and explained in §IV. We also give, in fig. 2, inference rules for the set of syntactically valid values $\mathcal{V}(\tau)$ for a given type τ. The notation $[\sigma/\alpha]\tau$ refers to the capture-avoiding substitution of σ for α in τ. Note that these typing rules do not guarantee anything for the purposes of language semantics–in particular, the function type here is practically unrestricted–but are instead just to guarantee that the potential function can be evaluated on indices and values of matching types.

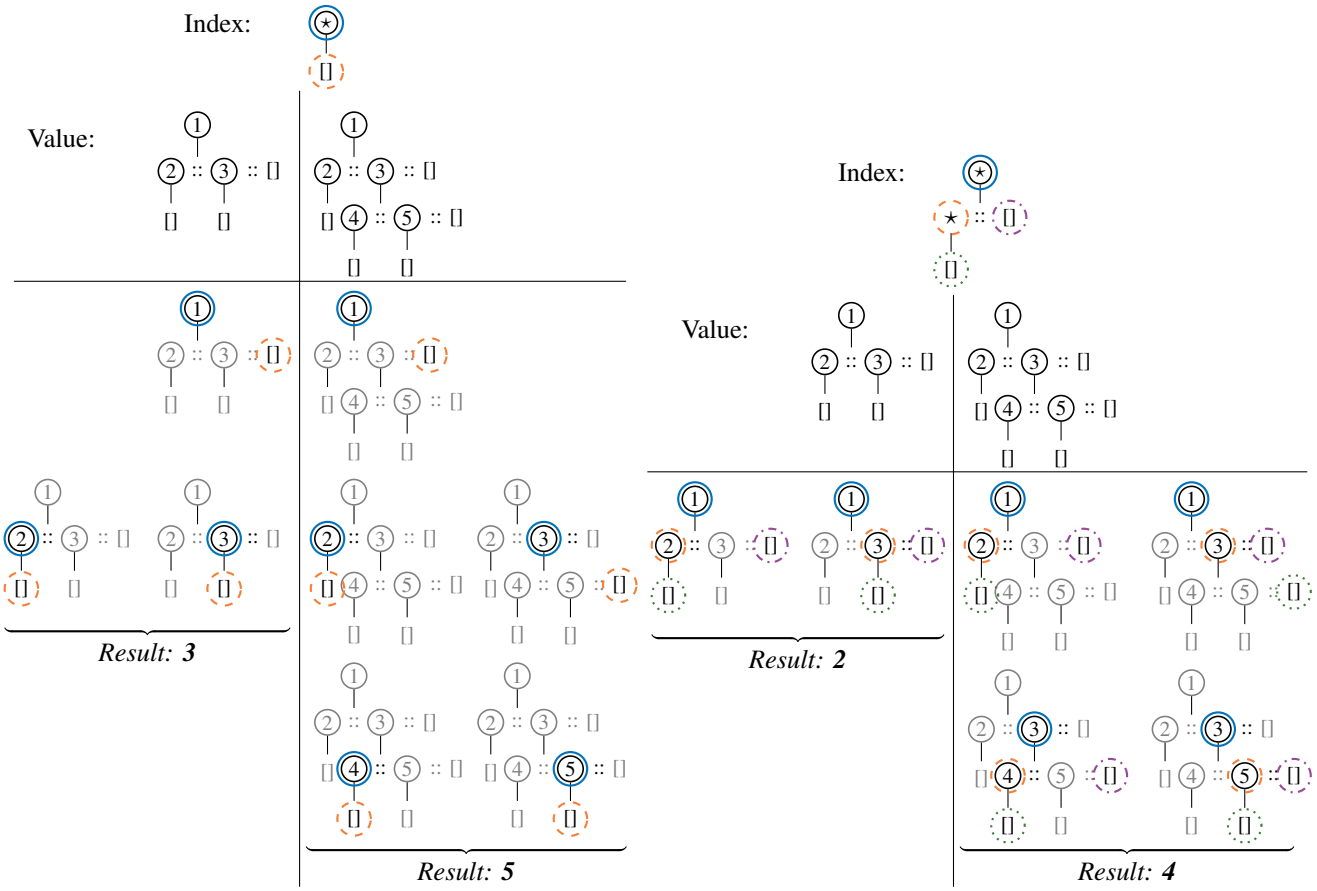


Fig. 1: Rose tree index examples

$$
\begin{array}{rrcll}
\text{Types} & \tau & ::= & \alpha & \text{Type variable} \\
& & | & \mathbf{1} & \text{Unit} \\
& & | & \tau_1 \times \tau_2 & \text{Product} \\
& & | & \tau_1 + \tau_2 & \text{Sum} \\
& & | & \langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathsf{cf}} \rangle & \text{Arrow} \\
& & | & \mu\alpha.\,\tau & \text{Isorecursive}
\end{array}
$$

$$
\begin{array}{rrcllll}
\text{Values} & v & ::= & \mathsf{tt} & | & \mathsf{pair}(v_1; v_2) \\
& & | & \mathsf{inl}\, v & | & \mathsf{inr}\, v \\
& & | & \mathsf{fun}(f, x.\, e) & | & \mathsf{fold}\, v
\end{array}
$$

$$\frac{}{\mathsf{tt} \in \mathcal{V}(\mathbf{1})} \qquad \frac{}{\mathsf{fun}(f, x.\, e) \in \mathcal{V}(\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathsf{cf}} \rangle)}$$

$$\frac{v_1 \in \mathcal{V}(\tau_1) \qquad v_2 \in \mathcal{V}(\tau_2)}{\mathsf{pair}(v_1; v_2) \in \mathcal{V}(\tau_1 \times \tau_2)} \qquad \frac{v_1 \in \mathcal{V}(\tau_1)}{\mathsf{inl}\, v_1 \in \mathcal{V}(\tau_1 + \tau_2)}$$

$$\frac{v_2 \in \mathcal{V}(\tau_2)}{\mathsf{inr}\, v_2 \in \mathcal{V}(\tau_1 + \tau_2)} \qquad \frac{v \in \mathcal{V}([\mu\alpha.\,\tau/\alpha]\tau)}{\mathsf{fold}\, v \in \mathcal{V}(\mu\alpha.\,\tau)}$$

Fig. 2: Types, values, and typing of values

Following our running examples, we may define the types $\mathsf{bool} \triangleq \mathbf{1} + \mathbf{1}$, $\mathsf{list}(\tau) \triangleq \mu\alpha.\,\mathbf{1} + \tau \times \alpha$, and $\mathsf{tree}(\tau) \triangleq \mu\beta.\,\tau \times \mathsf{list}(\beta) = \mu\beta.\,\tau \times (\mu\alpha.\,\mathbf{1} + \beta \times \alpha)$, with value constructors $\mathsf{True} \triangleq \mathsf{inl}\,\mathsf{tt}$ and $\mathsf{False} \triangleq \mathsf{inr}\,\mathsf{tt}$, $\mathsf{Nil} \triangleq \mathsf{fold}\,(\mathsf{inl}\,\mathsf{tt})$ and $\mathsf{Cons}(h, t) \triangleq \mathsf{fold}\,(\mathsf{inr}\,(\mathsf{pair}(h; t)))$, and $\mathsf{Tree}(x, t) \triangleq \mathsf{fold}\,(\mathsf{pair}(x; t))$, respectively. We use the notation $[v_1, \ldots, v_n]$ to refer to $\mathsf{Cons}(v_1, \ldots (\mathsf{Cons}(v_n, \mathsf{Nil})))$.

Indices: Resource polynomials consist of a sum of "monomial" base polynomials with rational coefficients. We use indices to name those base polynomials. Figure 3 shows inference rules for the set of indices $\mathcal{I}(\tau)$ for a given type τ. They nearly exactly mirror the syntactic values $\mathcal{V}(\tau)$, with the addition of an "end" index for recursive types.[2] One possible intuition for an index is to view it like a pattern in a pattern match specifying a shape that values are compared against. However, matching a pattern is a binary decision, whereas an index *counts* occurrences in a value.

Following our running examples, both True and False are indices for bool that match those values exactly; Nil and end are indices for $\mathsf{list}(\tau)$ that match against any list value exactly once; $\mathsf{Cons}(\mathsf{tt}, \mathsf{Nil})$ matches against any $\mathsf{list}(\mathbf{1})$ value as many times as the length of the list; and $\mathsf{Node}(\mathsf{tt}, \mathsf{Nil})$ matches

[2]Several parts of the type system rely on describing constant potential; we thus add end to do so for recursive types otherwise lacking such an index.

Base polynomial indices $\boxed{i \in \mathcal{I}(\tau)}$

$$\frac{}{\mathsf{tt} \in \mathcal{I}(\mathbf{1})} \qquad \frac{i_1 \in \mathcal{I}(\tau_1) \qquad i_2 \in \mathcal{I}(\tau_2)}{\mathsf{pair}(i_1;\, i_2) \in \mathcal{I}(\tau_1 \times \tau_2)}$$

$$\frac{i_1 \in \mathcal{I}(\tau_1)}{\mathsf{inl}\, i_1 \in \mathcal{I}(\tau_1 + \tau_2)} \qquad \frac{i_2 \in \mathcal{I}(\tau_2)}{\mathsf{inl}\, i_2 \in \mathcal{I}(\tau_1 + \tau_2)}$$

$$\frac{}{\lambda \in \mathcal{I}(\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathsf{cf}} \rangle)} \qquad \frac{i \in \mathcal{I}([\mu\alpha.\,\tau/\alpha]\tau)}{\mathsf{fold}\, i \in \mathcal{I}(\mu\alpha.\,\tau)}$$

$$\frac{}{\mathsf{end} \in \mathcal{I}(\mu\alpha.\,\tau)}$$

Recursive occurrence indices $\boxed{\mathcal{M}\{\alpha.\tau\}(i)}$

$$\begin{aligned}
\mathcal{M}\{\alpha.\alpha\}(i) &= \{i\} \\
\mathcal{M}\{\alpha.\top_\mu\}(i) &= \emptyset \\
\mathcal{M}\{\alpha.1\}(i) &= \emptyset \\
\mathcal{M}\{\alpha.\tau_1 + \tau_2\}(i) &= \{\mathsf{inl}\, j \mid j \in \mathcal{M}\{\alpha.\tau_1\}(i)\} \cup \\
&\quad \{\mathsf{inr}\, j \mid j \in \mathcal{M}\{\alpha.\tau_2\}(i)\} \\
\mathcal{M}\{\alpha.\tau_1 \times \tau_2\}(i) &= \{\mathsf{pair}(j;\, c) \mid j \in \mathcal{M}\{\alpha.\tau_1\}(i), \\
&\quad\qquad c \in \mathcal{C}(\tau_2)\} \cup \\
&\quad \{\mathsf{pair}(c;\, j) \mid c \in \mathcal{C}(\tau_1), \\
&\quad\qquad j \in \mathcal{M}\{\alpha.\tau_2\}(i)\} \\
\mathcal{M}\{\alpha.\tau_1 \to \tau_2\}(i) &= \emptyset \\
\mathcal{M}\{\alpha.\mu\beta.\,\tau\}(i) &= \{\mathsf{fold}\, j \mid j \in \mathcal{M}\{\alpha.[\top_\mu/\beta]\tau\}(i)\}
\end{aligned}$$

Constant index set $\boxed{\mathcal{C}(\tau)}$

$$\begin{aligned}
\mathcal{C}(\mathbf{1}) &= \{\mathsf{tt}\} \\
\mathcal{C}(\tau_1 \times \tau_2) &= \{\mathsf{pair}(i_1;\, i_2) \mid i_1 \in \mathcal{C}(\tau_1), \\
&\quad\qquad i_2 \in \mathcal{C}(\tau_2)\} \\
\mathcal{C}(\tau_1 + \tau_2) &= \{\mathsf{inl}\, i_1 \mid i_1 \in \mathcal{C}(\tau_1)\} \cup \\
&\quad \{\mathsf{inr}\, i_2 \mid i_2 \in \mathcal{C}(\tau_2)\} \\
\mathcal{C}(\tau_1 \to \tau_2) &= \{\lambda\} \\
\mathcal{C}(\mu\alpha.\,\tau) &= \{\mathsf{end}\} \\
\mathcal{C}(\top_\mu) &= \{\mathsf{end}\}
\end{aligned}$$

Base polynomial evaluation $\boxed{\phi_i(v : \tau)}$

$$\begin{aligned}
\phi_{\mathsf{tt}}(\mathsf{tt} : \mathbf{1}) &= 1 \\
\phi_{\mathsf{pair}(i_1;\, i_2)}(\mathsf{pair}(v_1;\, v_2) : \tau_1 \times \tau_2) &= \phi_{i_1}(v_1 : \tau_1) \cdot \phi_{i_2}(v_2 : \tau_2) \\
\phi_{\mathsf{inl}\, i_1}(\mathsf{inl}\, v_1 : \tau_1 + \tau_2) &= \phi_{i_1}(v_1 : \tau_1) \\
\phi_{\mathsf{inl}\, i_1}(\mathsf{inr}\, v_2 : \tau_1 + \tau_2) &= 0 \\
\phi_{\mathsf{inr}\, i_2}(\mathsf{inl}\, v_1 : \tau_1 + \tau_2) &= 0 \\
\phi_{\mathsf{inr}\, i_1}(\mathsf{inr}\, v_2 : \tau_1 + \tau_2) &= \phi_{i_2}(v_2 : \tau_2) \\
\phi_\lambda(\mathsf{fun}(f, x.\, e) : \langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathsf{cf}} \rangle) &= 1 \\
\phi_{\mathsf{fold}\, i}(\mathsf{fold}\, v : \mu\alpha.\,\tau) &= \phi_i(v : [\mu\alpha.\,\tau/\alpha]\tau) \\
&\quad + \textstyle\sum_{k \in \mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, i)} \phi_k(v : [\mu\alpha.\,\tau/\alpha]\tau) \\
\phi_{\mathsf{end}}(\mathsf{fold}\, v : \mu\alpha.\,\tau) &= 1
\end{aligned}$$

Fig. 3: Fundamental base polynomial index constructions.

against any $\mathsf{tree}(\mathbf{1})$ value as many times as nodes in the tree.

Constant index set: A function is given in fig. 3 that defines a set of indices $\mathcal{C}(\tau)$ for any type τ such that the sum of their evaluation on any value of type τ is exactly 1. The definition proceeds easily from the definition of index evaluation, which will be given shortly. We also include a definition of $\mathcal{C}$ for $\top_\mu$, a piece of syntax used in the course of the evaluation of $\mathcal{M}$ that is substituted for the bound type variable when unfolding a recursive type, in order to only unfold each recursive type once.

Following our running examples, we have $\mathcal{C}(\mathsf{bool}) = \{\mathsf{True}, \mathsf{False}\}$, which indeed encompasses all possible values of type bool, and $\mathcal{C}(\mathsf{list}(\tau)) = \mathcal{C}(\mathsf{tree}(\tau)) = \{\mathsf{end}\}$, which forms the set of constant indices for any recursive type.

Recursive occurrence index set: This is the key insight that enables the extension to more general algebraic, mutually inductive types. The function $\mathcal{M}\{\alpha.\tau\}(i)$ defined in fig. 3, where τ is a type with no free type variables except for α and i is an index for the type that α represents, returns a set of indices that correspond to placing i at every occurrence of α in τ. In more detail, here are the function's cases:

- α. We have found an occurence of α, so i goes here.
- $\top_\mu$. This represents some occurrence of a recursive type *other* than the one α refers to, having been substituted in during unfolding of said recursive type. Any occurrence of α within that recursive type has already been handled by the fold that is applied at the place of unfolding.
- $\mathbf{1}$. No occurences of α to be found here.
- $\tau_1 + \tau_2$. No matter whether the value turns out to be a left or right injection, there could be a value of type α within either, so we consider both cases.
- $\tau_1 \times \tau_2$. Here α could occur inside *both* projections of the pair, but we only want to consider one at a time, so we consider finding values in the first projection with arbitrary contents in the second, or vice versa.
- $\tau_1 \to \tau_2$. We treat functions opaquely, with no α values.
- $\mu\beta.\,\tau$. Here is the case critical for handling nested recursive types. As observed in §II-C, introducing a fold in the index here will cause this recursive process to happen over again during the *evaluation* of the index, but for β instead of α. This sort of

"delaying" of the recursive unrolling is what enables the nested recursive evaluation without having this process generate an infinite number of indices.

Following our running examples, we have

$$\mathcal{M}\{\alpha.\mathsf{bool}\}(i) = \emptyset,$$

because α is not free in bool;

$$\mathcal{M}\{\alpha.\mathbf{1} + \mathsf{bool} \times \alpha\}(i) = \{\mathsf{inr}\,(\mathsf{pair}(\mathsf{True};\, i)), \mathsf{inr}\,(\mathsf{pair}(\mathsf{False};\, i))\},$$

where $\mathbf{1} + \mathsf{bool} \times \alpha$ is $\mathsf{list}(\mathsf{bool})$ with the recursive binder stripped, because all recursive occurrences in a list of bools are at the tail of a cons cell with a bool as the head; and

$$\mathcal{M}\{\beta.\mathsf{bool} \times \mathsf{list}(\beta)\}(i) = \{\mathsf{pair}(\mathsf{False};\, \mathsf{fold}\,(\mathsf{inr}\,(\mathsf{pair}(i;\, \mathsf{end})))),\ \mathsf{pair}(\mathsf{True};\, \mathsf{fold}\,(\mathsf{inr}\,(\mathsf{pair}(i;\, \mathsf{end}))))\},$$

where $\mathsf{bool} \times \mathsf{list}(\beta)$ is $\mathsf{tree}(\mathsf{bool})$ with the recursive binder stripped, because all recursive occurrences in a rose tree of bools are in *some* cons cell of the list of children. It's worth examining the last example a little more closely to grok the intuition for how this works for mutually inductive types: though the number of direct recursive occurrences of rose trees is unbounded and thus at first glance might require infinite indices to represent, the fold corresponding to the list *itself* finds all of its recursive occurrences, allowing a finite number of indices to capture any number of descendants.

Index evaluation: Finally, we reach the definition of the index evaluation function $\phi_i(v : \tau)$ in fig. 3, which evaluates the index i for type τ on value v. This gives the result of "counting" the number of matches of i in v. The definition is straightforward except when evaluating an index $\mathsf{fold}\, i$, so we will just explain that rule in more detail. When evaluating index $\mathsf{fold}\, i$ on a value $\mathsf{fold}\, v$ of type $\mu\alpha.\,\tau$, we want to find all possible matches of i in v. The first place those could occur is directly at the value v, which the term $\phi_i(v : [\mu\alpha.\,\tau/\alpha]\tau)$ accounts for. However, we also want to consider matches in the recursive positions of the type within v; as explained above, these positions are exactly what $\mathcal{M}\{\alpha.\tau\}$ identifies, and we want to continue looking for all matches of $\mathsf{fold}\, i$ at those positions, so we sum the results of evaluating each index in $\mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, i)$ to count the recursive occurrences.

The results of evaluating indices on a few values of our example types are illustrated in fig. 4.

With index evaluation defined, we may characterize the key property of the constant index set by induction on τ.

Lemma 1 (Constant indices sum)**.** *For all types τ and values $v \in \mathcal{V}(\tau)$, $\sum_{i \in \mathcal{C}(\tau)} \phi_i(v : \tau) = 1$.*

Note that distinct indices may sometimes refer to the same base polynomial. For example, the two indices end and Nil for the type $\mathsf{list}(\tau)$ both represent the constant function.

Type τ	Index $i \in \mathcal{I}(\tau)$	Value $v \in \mathcal{V}(\tau)$	Result $\phi_i(v : \tau)$
bool	False	False	1
		True	0
	True	False	0
		True	1
list($\mathbf{1}$)	[]	[tt; tt]	1
		[tt; tt; tt; tt]	1
	[tt]	[tt; tt]	2
		[tt; tt; tt; tt]	4
	[tt; tt]	[tt; tt]	1
		[tt; tt; tt; tt]	6
tree($\mathbf{1}$)	end		1
			1
	Tree(tt, [])		3
			6
	Tree(tt, [Tree(tt, [])])		2
			7

Fig. 4: Example index evaluation results.

Resource polynomials, proper: We have up to this point described the base polynomials by way of specification of their syntactic indices; the set of *resource polynomials* $\mathcal{R}(\tau)$ for a type τ are then the linear combinations of base polynomials with nonnegative rational coefficients.

B. Semimodules Overview

This section gives a brief overview of *semimodules* and the operations on and properties of them that are relevant in this work. Semimodules share the same definition as vector spaces, except that scalars over which they are defined must only be a semiring rather than a field. Because of this difference, they lack much of the structure of vector spaces; however, *free* semimodules can be thought to behave fairly similarly because they have bases. Since we only work with free semimodules in this work, it is safe to carry over intuition from vector spaces.

An R-semimodule M is a generalization of a vector space that works over a semiring R instead of a field. In this work, we will assume that R is commutative. Just as in vector spaces, there is an addition operation $+ : M \times M \to M$ that forms a commutative monoid and a scalar multiplication operation $\cdot : R \times M \to M$ that respects the various distributive and identity laws. The simplest example of an R-semimodule is R itself, with semimodule operations inherited from the ring structure.

A linear map (known also as a homomorphism) between R-semimodules M and M' is a function from M to M' that respects addition and scalar multiplication. An isomorphism

between semimodules is a bijective linear map between them. A bilinear map $M \times M' \to M''$ is a function that is linear in both arguments.

A free R-semimodule is one that is finitely generated by a basis E over the semimodule operations. Equivalently, the free R-semimodule generated by E is the set of finite maps from E to nonzero elements of R, with addition and scalar multiplication defined pointwise. (Unlike vector spaces, not every semimodule is isomorphic to a free semimodule, but that is not relevant for this paper.)

If $f, g : M \to M'$ are linear maps where M is the free R-semimodule generated by E, then $f = g$ iff $f(e) = g(e)$ for all $e \in E$, as can be derived using the properties of linear maps by inducting over an M-element's finite generation.

The tensor product of two R-semimodules M_1 and M_2 is the semimodule $M_1 \otimes M_2$ with canonical linear map $\otimes : M_1 \times M_2 \mapsto m_1 \otimes m_2$ satisfying the universal property that all bilinear maps $f : M_1 \times M_2 \to M'$ have a unique linear map $\tilde{f} : M_1 \otimes M_2 \to M'$ such that $f = \tilde{f} \circ \otimes$. (Its existence can be shown with a quotient construction.) The tensor product has symmetric monoidal structure, so $M_1 \otimes M_2 \cong M_2 \otimes M_1$ and $M_1 \otimes (M_2 \otimes M_3) \cong (M_1 \otimes M_2) \otimes M_3$. A tensoring operation $f \otimes g$ on functions $f : M_1 \to M_1'$ and $g : M_2 \to M_2'$ can be defined such that $(f \otimes g)(m_1 \otimes m_2) = f(m_1) \otimes g(m_2)$ for any $m_1 \in M_1$ and $m_2 \in M_2$. By convention, for a function $f : M_1 \to M_1'$, the notation $f \otimes M_2$ denotes the map $f \otimes g : M_1 \otimes M_2 \to M_1' \otimes M_2$ where g is the identity map on M_2.

C. Annotations

Though resource polynomials are the objects we really care about for analysis, the most useful representation of resource polynomials is a reified form we call *annotations* $\mathcal{A}(\tau)$.

Definition 1 (Annotation)**.** *Let $\mathcal{A}(\tau)$ denote the free $\mathbb{Q}_{\geq 0}$-semimodule with $\mathcal{I}(\tau)$ as a basis. Then an* annotation *for type τ is an element of $\mathcal{A}(\tau)$.*

We denote annotations as P or Q and define p_i to be the coefficient corresponding to index i. Then we can recover the resource polynomial as the potential function

$$\Phi(v : \langle \tau; P \rangle) \triangleq \textstyle\sum_{i \in \mathcal{I}(\tau)} p_i \cdot \phi_i(v : \tau).$$

Note that $P \mapsto \Phi(\cdot : \langle \tau; P \rangle)$ is a linear map from annotations to resource polynomials and that $P \mapsto \Phi(v : \langle \tau; P \rangle)$ is a linear form. In addition to basic operations on semimodules, we also use the following notions on annotations:

- We implicitly coerce finite sets of indices $B \subseteq \mathcal{I}(\tau)$ to annotations $B \in \mathcal{A}(\tau)$, where $b_i = 1$ if $i \in B$ and 0 otherwise.
- We use the preorder $P \leq Q$, defined by $p_i \leq q_i$ for all i. Note $P \leq Q$ matches the extension order, i.e. $\exists R.\, Q = P + R$, and thus $\Phi(v : \langle \tau; \cdot \rangle)$ respects the order.
- Also note that because $\mathcal{I}(\tau_1 \times \tau_2) \cong \mathcal{I}(\tau_1) \times \mathcal{I}(\tau_2)$, we have $\mathcal{A}(\tau_1 \times \tau_2) \cong \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2)$. Then $\Phi(\mathsf{pair}(v_1; v_2) : \langle \tau_1 \times \tau_2; P \otimes Q \rangle) = \Phi(v_1 : \langle \tau_1; P \rangle) \cdot \Phi(v_2 : \langle \tau_2; Q \rangle)$. We will use the notations $\mathsf{pair} : \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2) \to \mathcal{A}(\tau_1 \times \tau_2)$ and $\mathsf{pair}^{-1} : \mathcal{A}(\tau_1 \times \tau_2) \to \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2)$ to explicitly note the two sides of this isomorphism.
- Similarly, we have linear maps $\mathsf{inl} : \mathcal{A}(\tau_1) \to \mathcal{A}(\tau_1 + \tau_2)$ and its retraction $\mathsf{inl}^{-1} : \mathcal{A}(\tau_1 + \tau_2) \to \mathcal{A}(\tau_1)$, and similarly for inr. Explicitly, inl^{-1} is the linear map with defining equations $\mathsf{inl}^{-1}(\mathsf{inl}\, i) = i$ and $\mathsf{inl}^{-1}(\mathsf{inr}\, i) = 0$.
- We use the notation $\mathrm{id}_{\mathcal{A}(\tau)}$ to refer to the identity linear map $P \mapsto P : \mathcal{A}(\tau) \to \mathcal{A}(\tau)$.

Then, if we want, say, n^2 potential for a unit list ℓ of length n, we can use the annotation $P = 1 \cdot [\mathsf{tt}] + 2 \cdot [\mathsf{tt}; \mathsf{tt}]$, so that

$$\begin{aligned}\Phi(\ell : \langle \mathsf{list}(\mathbf{1}); P \rangle) &= 1 \cdot \phi_{[\mathsf{tt}]}(\ell : \mathsf{list}(\mathbf{1})) + 2 \cdot \phi_{[\mathsf{tt};\mathsf{tt}]}(\ell : \mathsf{list}(\mathbf{1})) \\ &= n + 2\tbinom{n}{2} = n^2.\end{aligned}$$

Shifting: A key requirement for our resource polynomials is the ability to fold and unfold recursive values while maintaining potential. The *additive shift* operator accomplishes this.

Definition 2 (Additive shift operator)**.** *Let $\mu\alpha.\, \tau$ be a recursive type. Then the* additive shift operator $\lhd$ *is the linear map $\lhd : \mathcal{A}(\mu\alpha.\, \tau) \to \mathcal{A}([\mu\alpha.\, \tau/\alpha]\tau)$ corresponding to the function from basis elements $\mathcal{I}(\mu\alpha.\, \tau)$ to $\mathcal{A}([\mu\alpha.\, \tau/\alpha]\tau)$ defined by*

$$\lhd \mathsf{end} \triangleq \mathcal{C}([\mu\alpha.\, \tau/\alpha]\tau) \quad \lhd(\mathsf{fold}\, i) \triangleq i + \mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, i).$$

In words, end is the constant index, and $\mathsf{fold}\, i$ refers to evaluation at both the current constructor (the i term) as well as all immediate children (the $\mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, i)$ term). The key property we desire for this operator is as follows:

Theorem 1 (Shift preserves potential)**.** *For any $P \in \mathcal{A}(\mu\alpha.\, \tau)$ and $v \in \mathcal{V}([\mu\alpha.\, \tau/\alpha]\tau)$,*

$$\Phi(\mathsf{fold}\, v : \langle \mu\alpha.\, \tau; P \rangle) = \Phi(v : \langle [\mu\alpha.\, \tau/\alpha]\tau; \lhd P \rangle).$$

Proof. This is equivalent to the statement of equality of linear forms $\Phi(\mathsf{fold}\, v : \langle \mu\alpha.\, \tau; \cdot \rangle) = \Phi(v : \langle [\mu\alpha.\, \tau/\alpha]\tau; \lhd \cdot \rangle)$. By linearity, it suffices to show this on basis elements $\mathcal{I}(\mu\alpha.\, \tau)$, at which point the property follows directly. □

It can additionally be shown that shifting is in fact a linear isomorphism $\mathcal{A}(\mu\alpha.\, \tau) \cong \mathcal{A}([\mu\alpha.\, \tau/\alpha]\tau)$.

Sharing: Since we need to be able to use a value multiple times, we need to be able to split its potential across multiple uses. Though this may sound simple at first, subtleties arise due to the multivariate setting: what if the potential across the uses ends up intertwined? For this we need the sharing operator, a bilinear map $\curlyvee : \mathcal{A}(\tau) \to \mathcal{A}(\tau) \to \mathcal{A}(\tau)$. Similarly to shifting, it suffices to define this just on basis elements. Here is the definition for the critical case of sharing two fold indices – the full definition is given in the extended version [21].

$$\begin{aligned}\curlyvee_{\mu\alpha.\, \tau}(\mathsf{fold}\, i, \mathsf{fold}\, j) \triangleq\ & \mathsf{fold}\left(\curlyvee_{[\mu\alpha.\, \tau/\alpha]\tau}(i, j)\right) \\ &+ \mathsf{fold}\left(\curlyvee_{[\mu\alpha.\, \tau/\alpha]\tau}(\mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, i), j)\right) \\ &+ \mathsf{fold}\left(\curlyvee_{[\mu\alpha.\, \tau/\alpha]\tau}(i, \mathcal{M}\{\alpha.\tau\}(\mathsf{fold}\, j))\right) \\ &+ \mathsf{fold}\left(\mathcal{N}\{\alpha.\tau\}(\mathsf{fold}\, i, \mathsf{fold}\, j)\right)\end{aligned}$$

Here $\mathcal{N}$ is like $\mathcal{M}$, but places *two* indices at *two* occurrences of α. Intuitively, this says that a pair of indices can apply in

the same value in any of these four categories of places: both at the current value, the left at a child and the right at the current value, the left at the current value and the right at a child, or both at a child.

The sharing operator satisfies the key property stated below:

Theorem 2 (Share preserves potential)**.** *For $P, Q \in \mathcal{A}(\tau)$ and $v \in \mathcal{V}(\tau)$, $\Phi(v : \langle \tau; P \rangle) \cdot \Phi(v : \langle \tau; Q \rangle) = \Phi(v : \langle \tau; P \curlyvee Q \rangle)$.*

D. Comparison to Hoffmann et al. [17]

It might not be clear whether our new resource polynomials are a *generalization* of previous multivariate AARA potentials[17], or are simply *different*. In fact it is the former:

Theorem 3. *All resource polynomials representable in Hoffmann et al. [17] are also representable in our system.*

Proof sketch. We show this for base polynomials by induction over types; the only nontrivial case is for binary trees. After a further induction over the length of the index list (the index for such a binary tree), we can consider all possible splittings of the list, inductively obtain annotations for each splitting, and construct nodes with those annotations on either side. □

IV. Language & Type System

Our language is essentially eager FPC [20], [23, Chapter 20] with a $\mathsf{tick}\{q\}$ expression to express cost and an explicit let construct. tick expressions are the only sources of cost in our language, but any given cost metric based on syntactic forms can be desugared into a language with no cost other than explicit tick expressions; additionally, such forms offer more flexibility for the programmer to specify particular kinds of costs.

A. Semantics

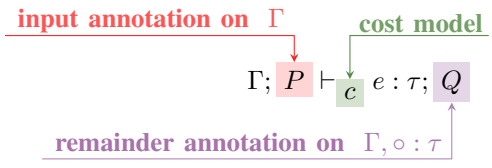
The cost semantics of the language is a standard small-step operational semantics. The judgement $(e, q) \mapsto (e', q')$ says the expression e, starting with $q \geq 0$ resources, transitions in a single step to expression e', with $q' \geq 0$ resources remaining. The only reduction added compared to a pure call-by-value language is that for $\mathsf{tick}\{q\}$, as follows:

$$\frac{q \geq q_0}{(\mathsf{tick}\{q_0\}, q) \mapsto (\mathsf{tt}, q - q_0)}$$

The judgement $(e, q) \mapsto^* (e', q')$ is then the transitive reflexive closure of the single step relation, with the constraint that only nonnegative resources may be considered. For use in cost-free derivations, we also define a "pure" semantics $e \to e' \triangleq \exists q, q'.\, (e, q) \mapsto (e', q')$ and denote the transitive reflexive closure of that as $e \to^* e'$.

B. Type judgements

Type judgements in our system are as follows:



This reads "in the context Γ with P resources, under cost model c, e has type τ with Q resources left over." We now describe each of these constructs that we have yet to explain: context annotations, remainder contexts, and cost models.

Context annotations: Define $\mathcal{I}(\Gamma) \triangleq \prod_{x:\tau \in \Gamma} \mathcal{I}(\tau)$ and $\mathcal{A}(\Gamma)$ and $\mathcal{C}(\Gamma)$ as the corresponding extensions of $\mathcal{A}(\tau)$ and $\mathcal{C}(\tau)$; note that $\mathcal{A}(\Gamma) \cong \bigotimes_{x:\tau \in \Gamma} \mathcal{A}(\tau)$. We may notationally elide coercions between isomorphic semimodules, such as $\mathcal{A}(\Gamma_1, \Gamma_2) \cong \mathcal{A}(\Gamma_1) \otimes \mathcal{A}(\Gamma_2)$ and $\mathcal{A}(x : \tau) \cong \mathcal{A}(\tau)$. Other annotation operations (all of which are linear maps) include:

- The "projection operator" $\pi_i : \mathcal{A}(\Gamma_1, \Gamma_2) \to \mathcal{A}(\Gamma_1)$, where $i \in \mathcal{I}(\Gamma_2)$, which takes a slice of the annotation at the index i. $Q = \pi_i(P)$ is defined by $q_j = p_{(j,i)}$.
- The "pairing operator" $\mathsf{pair}_y^{x_1,x_2} : \mathcal{A}(\Gamma, x_1 : \tau_1, x_2 : \tau_2) \to \mathcal{A}(\Gamma, y : \tau_1 \times \tau_2)$ defined just by reassociating, because $\mathcal{A}(\Gamma, x_1 : \tau_1, x_2 : \tau_2) \cong \mathcal{A}(\Gamma) \otimes \mathcal{A}(\tau_1 \times \tau_2) \cong \mathcal{A}(\Gamma, y : \tau_1 \times \tau_2)$.
- An extension of the shift operator to context annotations, $\lhd_x = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \lhd) : \mathcal{A}(\Gamma, x : \mu\alpha.\tau) \to \mathcal{A}(\Gamma, x : [\mu\alpha.\tau/\alpha]\tau)$, i.e., shifting is applied to x.
- An extension of the sharing operator to context annotations, ${}^x\curlyvee_z^y = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \curlyvee) : \mathcal{A}(\Gamma, x : \tau, y : \tau) \to \mathcal{A}(\Gamma, z : \tau)$, i.e., the variables x and y are shared together into z

Remainder contexts: To more accurately track potential, we make use of annotations on *remainder contexts*, which were inspired by IO-contexts from linear logic proof search [24], [25] but introduced in the programmatic AARA setting by Kahn and Hoffmann [26]. Remainder contexts contain both the typing context Γ with the additional pseudovariable "$\circ : \tau$" representing the result. Annotations on remainder contexts then represent the potential left on the whole context after the expression being typed has terminated.

Many of the benefits of remainder contexts noted in [26] extend to our setting. Such advantages include functions' ability to return potential back to their arguments after being called and the elimination of explicit sharing.

Cost models: We have two different cost models: "cost-paid", denoted cp, and "cost-free", denoted cf. The former refers to an actual, cost-relevant execution ($\mapsto^*$), while the latter refers to a pure execution ($\to^*$). While we aren't concerned with pure executions directly, understanding them is necessary to transform mixed potential contexts across abstraction boundaries (in our case, function calls).

C. Types

We explained of most of our language's types in §III-A, but discuss recursive types and function types in more detail here.

Support for more general recursive types are the core novel feature we add to AARA. As put forth in sections §II-C and §III, we make substantial contributions in generalizing resource polynomials to them. We chose the word "recursive" to describe these types because that is indeed how they are constructed. However, we wish to stress that the distinction between recursive types and *inductive* types is not very meaningful in our setting: for one, our functions have built-in general recursion, so recursive types do not grant

Resource typing $\boxed{\Gamma; P \vdash_c e : \tau; Q}$

T:TickPos
$$\frac{q \geq 0 \quad P \otimes \mathcal{C}(\circ : \mathbf{1}) = Q + q \cdot \mathcal{C}(\Gamma, \circ : \mathbf{1})}{\Gamma; P \vdash_{\mathsf{cp}} \mathsf{tick}\{q\} : \mathbf{1}; Q}$$

T:TickNeg
$$\frac{q \geq 0 \quad P \otimes \mathcal{C}(\circ : \mathbf{1}) + q \cdot \mathcal{C}(\Gamma, \circ : \mathbf{1}) = Q}{\Gamma; P \vdash_{\mathsf{cp}} \mathsf{tick}\{-q\} : \mathbf{1}; Q}$$

T:InR
$$\frac{P = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes (\curlyvee \circ (\mathrm{id}_{\mathcal{A}(\tau_r)} \otimes \mathsf{inr}^{-1})))(Q)}{\Gamma, x : \tau_r; P \vdash_c \mathsf{inr}\, x : \tau_l + \tau_r; Q}$$

T:TickFree
$$\frac{P \otimes \mathcal{C}(\circ : \mathbf{1}) = Q}{\Gamma; P \vdash_{\mathsf{cf}} \mathsf{tick}\{q\} : \mathbf{1}; Q}$$

T:Var
$$\frac{{}^{\circ}\curlyvee_x^x(Q) = P}{\Gamma, x : \tau; P \vdash_c x : \tau; Q}$$

T:Unit
$$\frac{P \otimes \mathcal{C}(\circ : \mathbf{1}) = Q}{\Gamma; P \vdash_c \mathsf{tt} : \mathbf{1}; Q}$$

T:InL
$$\frac{P = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes (\curlyvee \circ (\mathrm{id}_{\mathcal{A}(\tau_l)} \otimes \mathsf{inl}^{-1})))(Q)}{\Gamma, x : \tau_l; P \vdash_c \mathsf{inl}\, x : \tau_l + \tau_r; Q}$$

T:Let
$$\frac{\begin{array}{c}\Gamma; P \vdash_c e_1 : \tau'; R \\ (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathrm{id}_{\mathcal{A}(\tau')})(R) = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathrm{id}_{\mathcal{A}(\tau')})(S) \\ \Gamma, x : \tau'; S \vdash_c e_2 : \tau; Q \otimes \mathcal{C}(x : \tau')\end{array}}{\Gamma; P \vdash_c \mathsf{let}(e_1;\, x.\, e_2) : \tau; Q}$$

T:CaseSum
$$\frac{\begin{array}{c}x' \text{ fresh} \qquad {}^{x'}\curlyvee_x^x(P') = P \qquad {}^{x'}\curlyvee_x^x(Q') = Q \qquad \forall n \in \{\mathsf{l}, \mathsf{r}\}.\, \big(\\ (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathsf{in}_n^{-1} \otimes \mathsf{in}_n^{-1})(P') = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathsf{in}_n^{-1} \otimes \mathrm{id}_{\mathcal{A}(\tau_n)})(R_n) \\ \Gamma, x : \tau_l + \tau_r, x_n : \tau_n; R_n \vdash_c e_n : \tau; S_n \\ (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathsf{in}_n^{-1} \otimes \mathrm{id}_{\mathcal{A}(\tau_n)} \otimes \mathrm{id}_{\mathcal{A}(\tau)})(S_n) = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes \mathsf{in}_n^{-1} \otimes \mathsf{in}_n^{-1} \otimes \mathrm{id}_{\mathcal{A}(\tau)})(R_n)\big)\end{array}}{\Gamma, x : \tau_l + \tau_r; P \vdash_c \mathsf{cases}(x;\, x_l.\, e_l;\, x_r.\, e_r) : \tau; Q}$$

T:Prod
$$\frac{P = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes (\mathsf{pair}^{-1} \circ \curlyvee \circ (\mathsf{pair} \otimes \mathrm{id}_{\mathcal{A}(\circ : \tau_1 \times \tau_2)})))(Q)}{\Gamma, x_1 : \tau_1, x_2 : \tau_2; P \vdash_c \mathsf{pair}(x_1;\, x_2) : \tau_1 \times \tau_2; Q}$$

T:CaseProd
$$\frac{\begin{array}{c}P = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes (\curlyvee \circ (\mathrm{id}_{\mathcal{A}(\tau_1 \times \tau_2)} \otimes \mathsf{pair})))(R) \\ \Gamma, x : \tau_1 \times \tau_2, x_1 : \tau_1, x_2 : \tau_2; R \vdash_c e : \tau; S \\ Q = (\mathrm{id}_{\mathcal{A}(\Gamma)} \otimes (\curlyvee \circ (\mathrm{id}_{\mathcal{A}(\tau_1 \times \tau_2)} \otimes \mathsf{pair})) \otimes \mathrm{id}_{\mathcal{A}(\tau)})(S)\end{array}}{\Gamma, x : \tau_1 \times \tau_2; P \vdash_c \mathsf{casep}(x;\, x_1, x_2.\, e) : \tau; Q}$$

T:Fold
$$\frac{P = {}^{\circ}\curlyvee_x^x(\lhd_{\circ}(Q))}{\Gamma, x : [\mu\alpha.\, \tau/\alpha]\tau; P \vdash_c \mathsf{fold}\, x : \mu\alpha.\, \tau; Q}$$

T:Unfold
$$\frac{P = {}^{\circ}\curlyvee_x^x(\lhd_{\circ}^{-1}(Q))}{\Gamma, x : \mu\alpha.\, \tau; P \vdash_c \mathsf{unfold}\, x : [\mu\alpha.\, \tau/\alpha]\tau; Q}$$

T:Fun
$$\frac{\begin{array}{c}\forall (R, S) \in \Theta_{\mathsf{cp}}.\, \Gamma, f : \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, x : \tau_1; \mathcal{C}(\Gamma) \otimes R \vdash_{\mathsf{cp}} e : \tau_2; \mathcal{C}(\Gamma) \otimes S \\ \forall (R, S) \in \Theta_{\mathsf{cf}}.\, \Gamma, f : \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, x : \tau_1; \mathcal{C}(\Gamma) \otimes R \vdash_{\mathsf{cf}} e : \tau_2; \mathcal{C}(\Gamma) \otimes S\end{array}}{\Gamma; P \vdash_c \mathsf{fun}(f, x.\, e) : \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle; Q}$$

T:Weaken
$$\frac{\begin{array}{c}\Gamma; R \vdash_c e : \tau; S \\ P = R \otimes \mathcal{C}(x : \sigma) \qquad Q = S \otimes \mathcal{C}(x : \sigma)\end{array}}{\Gamma, x : \sigma; P \vdash_c e : \tau; Q}$$

T:App
$$\frac{\begin{array}{c}\forall i \in \mathcal{C}(\Gamma).\, (\pi_{\Gamma \mapsto i}(P), \pi_{\Gamma \mapsto i}(Q)) \in \Theta_{\mathsf{cp}} \\ \forall i \notin \mathcal{C}(\Gamma).\, (\pi_{\Gamma \mapsto i}(P), \pi_{\Gamma \mapsto i}(Q)) \in \Theta_{\mathsf{cf}}\end{array}}{\Gamma, f : \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, x : \tau_1; P \vdash_c \mathsf{app}(f;\, x) : \tau_2; Q}$$

T:Relax
$$\frac{\begin{array}{c}\Gamma'; R \vdash_c e : \tau'; S \\ R \leq P \qquad Q \leq S\end{array}}{\Gamma; P \vdash_c e : \tau; Q}$$

T:Augment
$$\frac{\begin{array}{c}\Gamma; R \vdash_c e : \tau; S \\ P = R + T \qquad Q = S + T \otimes \mathcal{C}(\tau)\end{array}}{\Gamma; P \vdash_c e : \tau; Q}$$

Fig. 5: Resource typing inference rules. See §IV-D for explanations of selected rules.

any more expressive power there than inductive types would; furthermore, because we have only trivial (constant) resource polynomials over function values themselves, the landscape of resource polynomials is also unaffected by the distinction between recursive and inductive types.

Function types have the form $\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathsf{cf}} \rangle$, where $\Theta, \Theta_{\mathsf{cf}} \subseteq \mathcal{A}(\tau_1) \times \mathcal{A}(\tau_1 \times \tau_2)$. Θ and Θ_{cf} are *resource specifications*: they denote how much potential is required on a function's argument and how much potential is returned on the argument and result. A function may have many different resource specifications in order to handle resource polymorphic recursion. Intuitively, for any specification $(P, Q) \in \Theta$, to call the function on an argument v, at least $\Phi(v : \langle \tau_1; P \rangle)$ resources are needed; once it returns with value v', $\Phi(\mathsf{pair}(v;\, v') : \langle \tau_1 \times \tau_2; Q \rangle)$ resources are returned as well. The argument v is mentioned in the output potential to enable the remainder contexts discussed in §IV-B. Θ_{cf} serves a similar purpose for the cost-free model discussed in §IV-B: $(P, Q) \in \Theta_{\mathsf{cf}}$ implies that $\Phi(\mathsf{pair}(v;\, v') : \langle \tau_1 \times \tau_2; Q \rangle) \geq \Phi(v : \langle \tau_1; P \rangle)$, with no mention of resources gained or spent during execution.

D. Typing rules

The typing rules for our language are available in fig. 5. All rules are syntax-directed, except for the final three, which are structural. All rules apply to programs in *let-normal form* to allow more precise accounting of potential. Of course, a preprocessing step may be added before typing to hide this restriction from the viewpoint of the user.

We now overview some *interesting* typing rules of fig. 5.

T:Let: This rule is primarily interesting because of how simple it is compared to past work. In previous multivariate AARA works, this rule had to consider cost-free derivations of the term being bound, in order to handle mixed potential between the result and other variables in the context [17]. However, our remainder contexts remove this requirement because the remainder annotation may simply mention these mixed potentials. This improvement does not come for free;

instead, cost-free derivations are needed for functions, but we believe that that is a more appropriate abstraction boundary.

T:PAIR*:* Some rules use tensor constructions that look complex but are really just victims of cumbersome bookkeeping; this rule is one of them. In words, the condition says that Q is just P, but with x_1 and x_2 shared and then packed into $\circ$.

T:FOLD *and* T:UNFOLD*:* These rules are the core of the potential annotation-based type system, taking advantage of the definition of the additive shift operator in §III-C.

T:FUN*:* The function abstraction rule requires showing that each annotation pair in the function type is justified by a type derivation. For convenience, this derivation gives direct access to using the function recursively. However, this is redundant, as self-reference may be derived from recursive types in the standard manner, defining $\mathsf{self}(\tau) \triangleq \mu\alpha.\, \langle \alpha \to \tau, \{(0,0)\}, \{(0,0)\}\rangle$ with the constructor and eliminator as usual [23, §20.3].

T:APP*:* This function application rule essentially requires enough potential on the argument to use a cost-paid annotation of the function that is then returned into the remainder annotation, and then all of the mixed potential terms must be transformed according to cost-free annotations of the function.

E. Typing example

We now give examples of typings that can be given with this type system. We present the code in a surface language with names for constructors of recursive types.

Imagine implementing a mock filesystem. Each node is either a file, consisting of its name and contents, or a directory, consisting of its name and a list of its contained nodes.

```
type filesystem = File of string * string
                | Dir of string * filesystem list
```

We can then define a function attach to demonstrate that our indices can represent simple bounds like in previous work. Letting foldl : $('\mathsf{b} \times '\mathsf{a} \to '\mathsf{b}) \to '\mathsf{b} \times \mathsf{list}('\mathsf{a}) \to '\mathsf{b}$ be the left fold function on lists, attach can be defined with:

```
let rec attach dname (acc, fs) = match fs with
  | File (fname, _) → [(dname, fname)]
  | Dir (subdname, fss) →
    (dname, subdname) :: foldl (attach dname) (acc, fss)
```

This attach function, given a name and a filesystem, returns a list of all pairs of that name and the name of a directory or file in the filesystem. If we count cons cell creation as cost, then the cost is equal to the length of that output list, which is the number of nodes in the filesystem. We can represent this cost in our type system by annotating the simple type $\mathsf{attach} : \mathbb{S} \to \mathsf{list}(\mathbb{S} \times \mathbb{S}) \times \mathsf{filesystem} \to \mathsf{list}(\mathbb{S} \times \mathbb{S})$ using our indices, where $\mathbb{S}$ is the string type. The only nonzero annotation required is that the second argument must provide $1 \cdot ([], \mathsf{File}(\star, \star)) + 1 \cdot ([], \mathsf{Dir}(\star, []))$ potential, where $\star$ gives the constant potential on strings. This amount is just 1 unit of potential per file or directory, precisely the cell count expected.

We can now define a second function trans[3] to demonstrate that more novel cost bounds are expressible with our indices.

[3]This is adapted from an example given by Hoffmann [27].

```
let rec trans (acc, fs) = match fs with
  | File (_, _) → []
  | Dir (dname, fss) →
    foldl trans ((foldl (attach dname) (acc, fss)), fss)
```

Given a representation of a filesystem, trans computes a list of every pair (d, s) such that d is the name of an ancestor directory of the file or subdirectory named s. This creates cons cells at worst quadratic in the number of nodes in the filesystem, but the exact amount varies depending on the filesystem's shape. Nonetheless, we can represent this number exactly by annotating the simple type $\mathsf{trans} : \mathsf{list}(\mathbb{S} \times \mathbb{S}) \times \mathsf{filesystem} \to \mathsf{list}(\mathbb{S} \times \mathbb{S})$ using our indices. The only nonzero annotation required is that the argument must provide $1 \cdot ([], \mathsf{Dir}(\star, [\mathsf{File}(\star, \star)])) + 1 \cdot ([], \mathsf{Dir}(\star, [\mathsf{Dir}(\star, [])]))$ potential. This amount is 1 unit of potential per ancestral directory relation in the filesystem, precisely as expected. One can also infer that this amount can grow quadratically by noting the twice-nested occurrences of the Dir constructor.

F. Soundness

Soundness here means that a well-typed term in a closed context is safe to execute if starting with enough resources:

Theorem 4 (Soundness)**.** *Assume* $\cdot; p \vdash_{\mathsf{cp}} e : \tau; Q$*. For any* $r \geq 0$ *and execution* $(e, p + r) \mapsto^* (e', q)$*, we have either:*

- $e' \in \mathcal{V}(\tau)$ *and* $q \geq r + \Phi(e' : \langle \tau; Q \rangle)$*, or*
- *there exists some* e'' *and* q' *such that* $(e', q) \mapsto (e'', q')$*.*

Proof. We construct a step-indexed logical relation; the only non-standard case is that for the arrow type, in which, starting at the next step, all of the type's annotations must be satisfied in all states with sufficiently many resources:

$$
\begin{aligned}
&\mathcal{V}'[\![\langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle]\!]_{n+1}(v) \triangleq \\
&\quad \exists f, x, e.\ v = \mathsf{fun}(f, x.\, e) \wedge \forall m \leq n. \\
&\qquad (\forall (P, Q) \in \Theta_{\mathsf{cp}}.\ \forall q \geq 0, v'.\ \mathcal{V}[\![\tau_1]\!]_m(v') \Rightarrow \\
&\qquad\quad \mathcal{WP}_{\mathsf{cp}}[\lambda(m', v'', q').\ \mathcal{V}[\![\tau_2]\!]_{m'}(v'') \wedge \\
&\qquad\qquad q' \geq q + \Phi(\mathsf{pair}(v'; v'') : \langle \tau_1 \times \tau_2; Q \rangle)]_m \\
&\qquad\quad ([v'/x, v/f]e, q + \Phi(v' : \langle \tau_1; P \rangle)) \wedge \\
&\qquad (\forall (P, Q) \in \Theta_{\mathsf{cf}}.\ \forall q \geq 0, v'.\ \mathcal{V}[\![\tau_1]\!]_m(v') \Rightarrow \\
&\qquad\quad \mathcal{WP}_{\mathsf{cf}}[\lambda(m', v'', q').\ \mathcal{V}[\![\tau_2]\!]_{m'}(v'') \wedge \\
&\qquad\qquad q' \geq q + \Phi(\mathsf{pair}(v'; v'') : \langle \tau_1 \times \tau_2; Q \rangle)]_m \\
&\qquad\quad ([v'/x, v/f]e, q + \Phi(v' : \langle \tau_1; P \rangle))
\end{aligned}
$$

where the cost-paid weakest precondition relation is:

$$
\begin{aligned}
&\mathcal{WP}_{\mathsf{cp}}[\Psi]_{n+1}(e, q) \triangleq \\
&\quad (e\ \mathrm{val} \wedge \Psi(n+1, e, q)) \vee \\
&\quad ((\exists e', q'.\ (e, q) \mapsto (e', q')) \wedge \\
&\qquad \forall e', q'.\ (e, q) \mapsto (e', q') \Rightarrow \mathcal{WP}_{\mathsf{cp}}[\Psi]_n(e', q'))
\end{aligned}
$$

and the cost-free relation is defined analogously. The full definitions are available in the extended version [21]. We then prove the fundamental theorem of logical relations by induction on the typing derivation and show that inhabitation in the logical relation implies the conclusion of this theorem. □

G. Automation Discussion

We expect the automation of this type system to be straightforward. Like previous AARA systems, we can reduce type inference to linear programming, because like those systems, our new typing rules generate only linear constraints. Previous work has shown this is sufficient to enable implementing a realistic type checker [27]. The main concern for our system specifically is the increase in number of indices over which our system generates constraints; we expect this to be modest, but heuristics could potentially be used to prune the search space if needed.

V. Related Work

Our work builds upon the literature of the Automatic Amortized Resource Analysis (AARA) type system. AARA was first introduced by Hofmann and Jost [15], using potential method [22] reasoning for the automatic derivation of heap-space bounds *linear* in the sizes of data structures. AARA has been extended to support bounds in the forms of polynomials [19], exponentials [28], logarithms [29], and maxima [26], [30]. However, each of these works bakes their size parameters and resource functions into the inductive data types of trees and lists (or labeled trees [18]). The one exception is the Schopenhauer language [16], which includes support for deriving bounds on programs using nested recursive types, but only for the class of linear functions. Our indices provide the first method of automatically constructing resources functions and size parameters for general recursive types. Our resulting system conservatively extends the bounds given by the *multivariate* polynomial system [17], [18], wherein bounds may depend on the products of the sizes of data structures. And, while our work does not go in these directions, AARA's analysis can also cover other models of computation and analysis, including imperative [31], object-oriented [32], [33], probabilistic [34], [35], and parallel computation [36], digital contract protocols [37], and lower bound costs [38]. Combinatorial species [39] may also relate to AARA's resource polynomials.

Aside from AARA, other type-based systems have also been used to analyze the resource usage of programs. These include linear dependent types [2], [1], refinement types [14], [10], modal types [3], sized types [40], [41], annotation-based systems [42], [43], and more. These type systems bookkeep costs using a variety of differing ideas, but they all enjoy the high composability provided by type systems, usually employ some linear features and cost constraints like AARA. Unlike AARA, however, many trade some degree of automatabilitiy for richer features – at the extreme other end from AARA one finds type-based proof logics [44], which require significant user work to prove cost bounds.

The term-rewriting space [45], [8], [46], [47] provides some work that is comparable with ours. Some work has even generalized multivariate potential over arbitrary types using tree automata [48] which may be general enough to contain our work's resource functions. However, their work leaves open how to pick appropriate automata, and how to solve the constraints they induce. There has been much work [49] to even solve simpler cases than the multivariate case.

Recurrence relations are another common approach to resource analysis, especially in a functional setting [11], [50], [5]. Some recent work uses potential-based reasoning for amortization [6]. Usually these methods operate by extracting recurrences from the code and then solving them. While this can be more difficult than solving the linear constraints extracted by AARA, it can allow the expression of bounding functions that AARA cannot yet support.

Techniques from imperative [7], [51] and logic [52] program analyses also can reason about cost in terms of general notions of data structure size. However, that work does so with manually-defined notions of size which are reduced to numerical analysis. While such numerical analysis is common in cost analysis, it does not focus on the the sorts of intrinsic features of data structures that our work does.

Other approaches to cost analysis include abstract interpretation [4], [53], [54], loop analyses [55], [56], relational cost analysis [57], [58], ranking functions [9], and program logics [59], [60], [61]. The field varies broadly and mixes many approaches. In particular, the cited program logics make use of potential-based reasoning like AARA.

VI. Conclusion and Future Work

This work's contributions include the extension of multivariate resource polynomials to regular recursive datatypes and the introduction of semimodules as a helpful formalism in the system's specification. The extended resource polynomials enable the resource analysis of programs using complex, nested data structures like rose trees. These comprise a major step forward in automatable resource analysis through the structural combinatorics of data types. Future work will include the implementation of these resource functions in a fully automated resource analysis typechecker, like RaML [18]. Further, we would like to extend our methods for generating resouce polynomials to also cover resource exponentials [28], which currently are not supported at the multivariate level.

Acknowledgments

We thank the reviewers for their helpful feedback on the initial version of this paper.

This article is based on research supported by the Algorand Centres of Excellence programme managed by the Algorand Foundation and by the National Science Foundation under awards 1801369, 1845514, 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.

References

[1] U. Dal Lago and M. Gaboardi, "Linear dependent types and relative completeness," in *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 2011, pp. 133–142.

[2] U. Dal Lago and B. Petit, "The geometry of types," *ACM SIGPLAN Notices*, vol. 48, no. 1, pp. 167–178, 2013.

[3] V. Rajani, M. Gaboardi, D. Garg, and J. Hoffmann, "A unifying type-theory for higher-order (amortized) cost analysis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–28, 2021.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Costa: Design and implementation of a cost and termination analyzer for java bytecode," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2007, pp. 113–132.

[5] G. Kavvos, E. Morehouse, D. R. Licata, and N. Danner, "Recurrence extraction for functional programs through call-by-push-value," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–31, 2019.

[6] J. W. Cutler, D. R. Licata, and N. Danner, "Denotational recurrence extraction for amortized analysis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.

[7] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," *ACM Sigplan Notices*, vol. 44, no. 1, pp. 127–139, 2009.

[8] M. Avanzini, U. Dal Lago, and G. Moser, "Analysing the complexity of functional programs: higher-order meets first-order," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 152–164.

[9] K. Chatterjee, H. Fu, and A. K. Goharshady, "Non-polynomial worst-case analysis of recursive programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 4, pp. 1–52, 2019.

[10] P. Wang, D. Wang, and A. Chlipala, "Timl: a functional language for practical complexity analysis with invariants," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–26, 2017.

[11] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps, "Compositional recurrence analysis revisited," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 248–262, 2017.

[12] E. Albert, S. Genaim, and A. N. Masud, "On the inference of resource usage upper and lower bounds," *ACM Trans. Comput. Log.*, vol. 14, no. 3, pp. 22:1–22:35, 2013. [Online]. Available: https://doi.org/10.1145/2499937.2499943

[13] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and Synthesizing Constant-Resource Implementations with Types," in *38th IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[14] I. Radiček, G. Barthe, M. Gaboardi, D. Garg, and F. Zuleger, "Monadic refinements for relational cost analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–32, 2017.

[15] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 185–197, 2003.

[16] S. Jost, K. Hammond, H.-W. Loidl, and M. 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*, 2010, pp. 223–236.

[17] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 357–370.

[18] J. Hoffmann, A. Das, and S.-C. Weng, "Towards automatic resource bound analysis for ocaml," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 359–373.

[19] J. Hoffmann and M. Hofmann, "Amortized resource analysis with polynomial potential," in *European Symposium on Programming*. Springer, 2010, pp. 287–306.

[20] M. P. Fiore and G. D. Plotkin, "An axiomatization of computationally adequate domain theoretic models of FPC," in *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 92–102. [Online]. Available: https://doi.org/10.1109/LICS.1994.316083

[21] J. Grosen, D. M. Kahn, and J. Hoffmann, "Automatic amortized resource analysis with regular recursive types: Extended version," 2023, arXiv:2304.13627. [Online]. Available: https://arxiv.org/abs/2304.13627

[22] R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306–318, 1985.

[23] R. Harper, *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016. [Online]. Available: https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

[24] I. Cervesato, J. S. Hodas, and F. Pfenning, "Efficient resource management for linear logic proof search," *Theoretical Computer Science*, vol. 232, no. 1-2, pp. 133–163, 2000.

[25] J. S. Hodas and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," *Information and computation*, vol. 110, no. 2, pp. 327–365, 1994.

[26] D. M. Kahn and J. Hoffmann, "Automatic amortized resource analysis with the quantum physicist's method," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 1–29, 2021.

[27] J. Hoffmann, "Types with potential: Polynomial resource bounds via automatic amortized analysis," Ph.D. dissertation, Ludwig-Maximilians-Universität München, 2011. [Online]. Available: https://edoc.ub.uni-muenchen.de/id/eprint/13955

[28] D. M. Kahn and J. Hoffmann, "Exponential automatic amortized resource analysis," in *International Conference on Foundations of Software Science and Computation Structures*. Springer, Cham, 2020, pp. 359–380.

[29] M. Hofmann, L. Leutgeb, D. Obwaller, G. Moser, and F. Zuleger, "Type-based analysis of logarithmic amortised complexity," *Mathematical Structures in Computer Science*, pp. 1–33, 2021.

[30] B. Campbell, "Amortised memory analysis using the depth of data structures," in *European Symposium on Programming*. Springer, 2009, pp. 190–204.

[31] Q. Carbonneaux, J. Hoffmann, and Z. Shao, "Compositional certified resource bounds," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 467–478.

[32] M. Hofmann and S. Jost, "Type-based amortised heap-space analysis," in *European Symposium on Programming*. Springer, 2006, pp. 22–37.

[33] M. Hofmann and D. Rodriguez, "Automatic type inference for amortised heap-space analysis," in *European Symposium on Programming*. Springer, 2013, pp. 593–613.

[34] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann, "Bounded expectations: resource analysis for probabilistic programs," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 496–512, 2018.

[35] D. Wang, D. M. Kahn, and J. Hoffmann, "Raising expectations: automating expected cost analysis with types," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–31, 2020.

[36] J. Hoffmann and Z. Shao, "Automatic static cost analysis for parallel programs," in *European Symposium on Programming Languages and Systems*. Springer, 2015, pp. 132–157.

[37] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar, "Resource-aware session types for digital contracts," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–16.

[38] M. Dehesa-Azuara, M. Fredrikson, J. Hoffmann *et al.*, "Verifying and synthesizing constant-resource implementations with types," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 710–728.

[39] F. Bergeron, G. Labelle, and P. Leroux, *Combinatorial Species and Tree-like Structures*, ser. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, 1997. [Online]. Available: https://www.cambridge.org/core/books/combinatorial-species-and-treelike-structures/D994A1F2877BDE63FF0C9EDE2F9788A8

[40] P. B. Vasconcelos, "Space cost analysis using sized types," Ph.D. dissertation, University of St Andrews, 2008.

[41] A. Serrano, P. López-García, and M. V. Hermenegildo, "Resource usage analysis of logic programs via abstract interpretation using sized types," *Theory and Practice of Logic Programming*, vol. 14, no. 4-5, pp. 739–754, 2014.

[42] K. Crary and S. Weirich, "Resource bound certification," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000, pp. 184–198.

[43] N. A. Danielsson, "Lightweight semiformal time complexity analysis for purely functional data structures," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 133–144, 2008.

[44] Y. Niu, J. Sterling, H. Grodin, and R. Harper, "A cost-aware logical framework," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–31, 2022.

[45] M. Avanzini and G. Moser, "A combination framework for complexity," *Information and Computation*, vol. 248, pp. 22–55, 2016.

[46] N. Hirokawa and G. Moser, "Automated complexity analysis based on context-sensitive rewriting," in *Rewriting and Typed Lambda Calculi*. Springer, 2014, pp. 257–271.

[47] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl, "Complexity analysis for term rewriting by integer transition systems," in *Interna-*

tional Symposium on Frontiers of Combining Systems. Springer, 2017, pp. 132–150.

[48] M. Hofmann and G. Moser, "Multivariate amortised resource analysis for term rewrite systems," in *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[49] S. Bauer, "Decidability of linear tree constraints for resource analysis of object-oriented programs," Ph.D. dissertation, lmu, 2019.

[50] N. Danner, D. R. Licata, and R. Ramyaa, "Denotational cost semantics for functional languages with inductive types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 140–151.

[51] S. Gulwani, "Speed: Symbolic complexity bound analysis," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 51–62.

[52] J. Navas, E. Mera, P. López-García, and M. V. Hermenegildo, "User-definable resource bounds analysis for logic programs," in *International Conference on Logic Programming*. Springer, 2007, pp. 348–363.

[53] E. Albert, J. C. Fernández, and G. Román-Díez, "Non-cumulative resource analysis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 85–100.

[54] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo, "Interval-based resource usage verification by translation into horn clauses and an application to energy consumption," *Theory and Practice of Logic Programming*, vol. 18, no. 2, pp. 167–223, 2018.

[55] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács, "Abc: algebraic bound computation for loops," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 103–118.

[56] Z. Kincaid, J. Cyphert, J. Breck, and T. Reps, "Non-linear reasoning for invariant synthesis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–33, 2017.

[57] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann, "Relational cost analysis," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 316–329, 2017.

[58] W. Qu, M. Gaboardi, and D. Garg, "Relational cost analysis in a functional-imperative setting," *Journal of Functional Programming*, vol. 31, 2021.

[59] A. Guéneau, A. Charguéraud, and F. Pottier, "A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification," in *European Symposium on Programming*. Springer, 2018, pp. 533–560.

[60] R. Atkey, "Amortised resource analysis with separation logic," in *European Symposium on Programming*. Springer, 2010, pp. 85–103.

[61] G. Mével, J.-H. Jourdan, and F. Pottier, "Time credits and time receipts in iris," in *European Symposium on Programming*. Springer, 2019, pp. 3–29.