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 inter-procedural 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

1We 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.
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 filter_map : (τ → option(σ)) → list(τ) → list(σ), which is implemented as follows:

```ocaml
let rec filter_map f l = match l with
  | [] → []
  | x :: l' → match 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:

\[
\text{filter_map} : \langle \text{int}^0 + \text{bool}^0, 0 \rangle \rightarrow \langle \text{option}^0(\text{int}), 0 \rangle \rightarrow \langle \text{list}^1(\text{int}^0 + \text{bool}^0), 0 \rangle \rightarrow \langle \text{list}^3(\text{int}), 0 \rangle
\]

The type \langle \text{int}^0 + \text{bool}^0, 0 \rangle \rightarrow \langle \text{option}^0(\text{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 \langle \text{list}^1(\text{int}^0 + \text{bool}^0) \rangle expresses that the list argument carries one potential unit per element of the list, reflecting the bound to be proved. The output potential \langle \text{list}^3(\text{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:

\[
\text{filter_map} : \langle \text{int}^1 + \text{bool}^0, 0 \rangle \rightarrow \langle \text{option}^1(\text{int}), 0 \rangle \rightarrow \langle \text{list}^1(\text{int}^1 + \text{bool}^0), 0 \rangle \rightarrow \langle \text{list}^3(\text{int}), 0 \rangle
\]

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

\[
\text{filter_map} : \langle \text{int}^q_1 + \text{bool}^{q_2}, p_0 \rangle \rightarrow \langle \text{option}^{q_3}(\text{int}), p'_0 \rangle \rightarrow \langle \text{list}^{r_1}(\text{int}^{r_2} + \text{bool}^{r_2}), p_1 \rangle \rightarrow \langle \text{list}^{r_4}(\text{int}), p'_1 \rangle
\]

\[
r_1 \geq p_0 + 1, \ r_2 \geq q_1, \ r_3 \geq q_2, \ r_1 \geq p'_1, \ q_3 + p'_0 \geq 0, \ r_4
\]

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 \( L^{q_1}(A), q_0 \) to \( L^{q_0,q_1,...,q_m}(A) \), where \( q_i \) 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. One answer can be made elegant with a clever choice of basis: the coefficients \( \langle q_i \rangle \) correspond to a basis of binomial coefficients \( \binom{n}{i} \), rather than monomials \( n^i \), due to their possession of an additive shift function \( \langle q_0, \ldots, q_m \rangle = (q_0 + q_1, \ldots, q_{m-1} + q_m, q_m) \). This is a linear function that specifies how to preserve potential, i.e., evaluating \( \langle q \rangle \) on \( n \) is equal to evaluating \( \langle q \rangle \) 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 \( m,n \), 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 \( \binom{n}{i} \) or \( \binom{m}{i} \). 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 \([*] = \star :: \text{nil} \) on lists, which counts the number of ways that an element \( \star \) can be followed by \text{nil}, i.e., the length of the list. Visually consider evaluating this index on two lists of different lengths:

$$
\text{Index: } \star :: [1] \\
\text{Value: } 1 :: 2 :: [1] \quad 1 :: 2 :: 3 :: 4 :: [1]
$$

$$
\text{Result: } 2 \\
\begin{cases}
1 :: 2 :: [1] \\
1 :: 2 :: 3 :: 4 :: [1]
\end{cases}
$$

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

$$
\phi_{i_1, \ldots, i_n}(v :: u s) = \phi_i(v) \cdot \phi_{i_1, \ldots, i_n}(u s) + \phi_{i_1, \ldots, i_n}(u s)
$$

Here evaluating \( \phi \) 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: \( \langle i :: i s \rangle = (i, i s) + (\star, i :: i s) \), 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 \( \langle \text{list}\text{\{int\} + bool\text{\{0\}}}, 0 \rangle \rightarrow \langle \text{list}\text{\{int\}}, 0 \rangle \). Expressed using indices, this function requires its argument to have potential \( 2 \cdot \langle \text{inl}\star \rangle + 1 \cdot \langle \text{inr}\star \rangle \) and returns a value with potential \( 1 \cdot \langle \star \rangle \).

Building toward our desire to type quicksort, first consider some evaluations of the index \([*;\star]::[1]:\)

\[\text{Index: } \star :: [2] :: [1] \]

\[\text{Value: } 1 :: 2 :: [1] \quad 1 :: 2 :: 3 :: 4 :: [1]
\]

\[\text{Result: } 1 \begin{cases}
1 :: 2 :: [1] \\
1 :: 2 :: 3 :: 4 :: [1]
\end{cases}
\]

As expected, we find that this index corresponds to \( \binom{2}{1} \).

Thus, given that we know quicksort has cost \( n^2 \), we can express the required potential of its argument using indices as \( 2 \cdot \langle [*;\star] \rangle + 1 \cdot \langle \star \rangle \). Finally, we can consider our original function, sort_lefts_list. Here we can see that it must require an input potential of \( 2 \cdot \langle \text{inl}\star \rangle + 2 \cdot \langle \text{inl}\star \rangle \)–filter_map consumes the \( 1 \cdot \langle \text{inl}\star \rangle \) 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 \( \tau \) essentially follow the structure of values of type \( \tau \). 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:

Value:  

\[
\begin{array}{c}
\text{Index:} \\
\begin{array}{c}
\text{leaf: } \triangle \\
\text{node: } \circ \\
\text{Result: } 2 \\
\end{array}
\end{array}
\]

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:

\[
\begin{align*}
\phi_\Delta(\Delta) &= 1 \\
\phi_\Delta(t_1 \triangleright t_2) &= \phi_\Delta(t_1) + \phi_\Delta(t_2) \\
\phi_{i_1 i_2}(t_1 \triangleright t_2) &= \phi_{i_1}(v) \cdot \phi_{i_2}(t_1) \cdot \phi_{i_2}(t_2) \\
&\quad + \phi_{i_1}(v) \cdot \phi_{i_2}(t_1) \cdot \phi_{i_2}(t_2)
\end{align*}
\]

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

\[
\begin{align*}
\phi_{i_1 i_2}(v :: u s) &= \phi_{i_1}(v) \cdot \phi_{i_2}(u s) \\
&\quad + \phi_{i_1}(v) \cdot \phi_{i_2}(u s)
\end{align*}
\]

\text{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?

\text{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 occurrences of this 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 \text{Tree}(\text{inl}*, []) + 1 \cdot \text{Tree}(\text{inr}*, []), 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_list_tree, it is the similarly natural 2 \cdot \text{Tree}(\text{inl}*, [\text{Tree}(\text{inl}*, [])]) + 2 \cdot \text{Tree}(\text{inl}*, []), 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.

\text{A. Resource polynomial definitions}

\text{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 \(V(\tau)\) for a given type \(\tau\). The notation \(\sigma/\alpha/\tau\) refers to the capture-avoiding substitution of \(\sigma\) for \(\alpha\) in \(\tau\). 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.
Following our running examples, we may define the types bool \( \triangleq 1 + 1 \), list(\( \tau \)) \( \triangleq \mu \alpha.1 + \tau \times \alpha \), and tree(\( \tau \)) \( \triangleq \mu \beta.\tau \times \text{list}(\beta) \times (\mu \alpha.1 + \beta \times \alpha) \), with value constructors True \( \triangleq \text{inl tt} \) and False \( \triangleq \text{inr tt} \), Nil \( \triangleq \text{fold (inl tt)} \) and Cons(\( h, t \)) \( \triangleq \text{fold (inr (pair(h; t))} \), and Tree(\( x, t \)) \( \triangleq \text{fold (pair(x; t)} \), respectively. We use the notation \([v_1, \ldots, v_n]\) to refer to \( \text{Cons}(v_1, \ldots, \text{Cons}(v_n, \text{Nil})) \).

**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 \( I(\tau) \) for a given type \( \tau \). They nearly exactly mirror the syntactic values \( V(\tau) \), with the addition of an “end” index for recursive types. 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 list(\( \tau \)) that match against any list value exactly once; Cons(\( \text{tt}, \text{Nil} \)) matches against any list(\( 1 \)) value as many times as the length of the list; and Node(\( \text{tt}, \text{Nil} \)) matches

---

2Several 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

<table>
<thead>
<tr>
<th>Base polynomial indices</th>
<th>( i \in \mathcal{I}(\tau) )</th>
<th>Recursive occurrence indices</th>
<th>( \mathcal{M}{\alpha.\tau}(i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{tt} \in \mathcal{I}(1) )</td>
<td>( i_1 \in \mathcal{I}(\tau_1), \quad i_2 \in \mathcal{I}(\tau_2) )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = {i} )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = \emptyset )</td>
</tr>
<tr>
<td>( \text{pair}(i_1; i_2) \in \mathcal{I}(\tau_1 \times \tau_2) )</td>
<td>( i_1 \in \mathcal{I}(\tau_1), \quad i_2 \in \mathcal{I}(\tau_2) )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = \emptyset )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = \emptyset )</td>
</tr>
<tr>
<td>( \text{inl} \ i_1 \in \mathcal{I}(\tau_1 + \tau_2) )</td>
<td>( \text{inr} \ i_2 \in \mathcal{I}(\tau_1 + \tau_2) )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = {\text{inr} \ j \mid j \in \mathcal{M}{\alpha.\tau}(i)} )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = {\text{inr} \ j \mid j \in \mathcal{M}{\alpha.\tau}(i)} )</td>
</tr>
<tr>
<td>( \lambda \in \mathcal{I}((\tau_1 \rightarrow \tau_2, \Theta, \Theta_\delta)) )</td>
<td>( \lambda \in \mathcal{I}(\mu\alpha.\tau) )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = {\text{fold} \ j \mid j \in \mathcal{M}{\alpha.\tau}(i)} )</td>
<td>( \mathcal{M}{\alpha.\tau}(i) = {\text{fold} \ j \mid j \in \mathcal{M}{\alpha.\tau}(i)} )</td>
</tr>
</tbody>
</table>

Constant index set

<table>
<thead>
<tr>
<th>Constant index set</th>
<th>( \mathcal{C}(\tau) )</th>
<th>Base polynomial evaluation</th>
<th>( \phi_i(v : \tau) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{C}(1) = {\text{tt}} )</td>
<td>( \phi_{\text{tt}}(1) = 1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mathcal{C}(\tau_1 \times \tau_2) = {\text{pair}(i_1; i_2) \mid i_1 \in \mathcal{C}(\tau_1), \ i_2 \in \mathcal{C}(\tau_2)} )</td>
<td>( \phi_{\text{pair}(i_1; i_2)}(\text{pair}(v_1; v_2) : \tau_1 \times \tau_2) = \phi_{\text{pair}(v_1)}(v_1 : \tau_1) \cdot \phi_{\text{pair}(v_2)}(v_2 : \tau_2) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mathcal{C}(\tau_1 + \tau_2) = {\text{inl} \ i_1 \mid i_1 \in \mathcal{C}(\tau_1)} \cup {\text{inr} \ i_2 \mid i_2 \in \mathcal{C}(\tau_2)} )</td>
<td>( \phi_{\text{inl} i_1}(\text{inl} v_1 : \tau_1 + \tau_2) = \phi_{\text{inl} i_1}(v_1 : \tau_1) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mathcal{C}(\tau_1 \rightarrow \tau_2) = {\lambda} )</td>
<td>( \phi_{\text{inr} i_2}(\text{inr} v_2 : \tau_1 + \tau_2) = 0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mathcal{C}(\mu\alpha.\tau) = {\text{end}} )</td>
<td>( \phi_{\text{inl} i_2}(\text{inr} v_2 : \tau_1 + \tau_2) = \phi_{\text{inr} i_2}(v_2 : \tau_2) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \mathcal{C}(\top_\mu) = {\text{end}} )</td>
<td>( \phi_{\text{fun}(f, x, e)}((\tau_1 \rightarrow \tau_2, \Theta, \Theta_\delta)) = 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3: Fundamental base polynomial index constructions.

against any tree(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 \( \tau \) such that the sum of their evaluation on any value of type \( \tau \) 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 \( \mathcal{C}(\text{bool}) = \{\text{True}, \text{False}\} \), which indeed encompasses all possible values of type bool, and \( \mathcal{C}(\text{list}(\tau)) = \mathcal{C}(\text{tree}(\tau)) = \{\text{end}\} \), 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 \( \tau \) is a type with no free type variables except for \( \alpha \) and \( i \) is an index for the type that \( \alpha \) represents, returns a set of indices that correspond to placing \( i \) at every occurrence of \( \alpha \) in \( \tau \). In more detail, here are the function’s cases:

\( \alpha \). We have found an occurrence of \( \alpha \), so \( i \) goes here.

\( \top_\mu \). This represents some occurrence of a recursive type other than the one \( \alpha \) refers to, having been substituted in during unfolding of said recursive type. Any occurrence of \( \alpha \) within that recursive type has already been handled by the fold that is applied at the place of unfolding.

1. No occurrences of \( \alpha \) 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 \( \alpha \) within either, so we consider both cases.

\( \tau_1 \times \tau_2 \). Here \( \alpha \) 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.

\( \tau_1 \rightarrow \tau_2 \). We treat functions opaquely, with no \( \alpha \) 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 \( \beta \) instead of \( \alpha \). 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

\[ \mathcal{M}\{\alpha.\text{bool}\}(i) = \emptyset, \]

because \( \alpha \) is not free in bool;

\[ \mathcal{M}\{\alpha.\mathbb{1} + \text{bool} \times \alpha\}(i) = \{\text{inr} (\text{pair}(\text{True}; i)), \text{inr} (\text{pair}(\text{False}; i))\}, \]

where \( \mathbb{1} + \text{bool} \times \alpha \) is list(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

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

where \( \text{bool} \times \text{list}(\beta) \) is tree(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 unbound 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 \( \tau \) 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 fold \( i \), so we will just explain that rule in more detail. When evaluating index fold \( i \) on a value 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 fold \( i \) at those positions, so we sum the results of evaluating each index in \( \mathcal{M}\{\alpha.\tau\}(\text{fold} i) \) to count the recursive occurrences.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Index</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{1} )</td>
<td>False</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>list(1)</td>
<td>[ ]</td>
<td>[tt; tt]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[tt]</td>
<td>[tt; tt; tt]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[tt; tt]</td>
<td>[tt; tt]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>[tt; tt; tt]</td>
<td>[tt; tt]</td>
<td>4</td>
</tr>
<tr>
<td>tree(1)</td>
<td>end</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tree(tt, [ ] )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tree(tt, [Tree(tt, [ ] )] )</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \tau \) 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 \( \mathcal{R} \)-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 \( + : \mathcal{R} \times M \to M \) 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 \( \mathcal{R} \)-semimodule is \( R \) itself, with semimodule operations inherited from the ring structure.

A linear map (known also as a homomorphism) between \( \mathcal{R} \)-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 \( \tau \cdot (m_1 \otimes m_2) \) defined such that \( \tau \otimes (m_1 \otimes m_2) = \tau \cdot (m_1 \otimes m_2) \) for any \( m_1 \in M_1 \) and \( m_2 \in M_2 \). By convention, for a function \( f : M_1 \to M_3 \) and \( g : M_2 \to M_4 \), the notation \( f \otimes g \) denotes the map \( f \otimes g : M_1 \otimes M_2 \to M_3 \otimes M_4 \) 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 refined form we call annotations \( \mathcal{A}(\tau) \).

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

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

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 \( B \subseteq I(\tau) \) 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 \( \Phi(v : \langle \tau ; P \rangle) \) respects the order.
- Also note that because \( I(\tau_1 \times \tau_2) \cong I(\tau_1) \times I(\tau_2) \), we have \( \mathcal{A}(\tau_1 \times \tau_2) \cong \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2) \). Then \( \Phi(\text{pair}(v_1 ; v_2 : \langle \tau_1 \times \tau_2 ; P \otimes Q \rangle) \triangleq \Phi(v_1 : \langle \tau_1 ; P \rangle) \cdot \Phi(v_2 : \langle \tau_2 ; Q \rangle) \). We will use the notations pair : \( \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2) \to \mathcal{A}(\tau_1 \times \tau_2) \) and 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 \( \text{inl} : \mathcal{A}(\tau_1) \to \mathcal{A}(\tau_1 + \tau_2) \) and its retraction \( \text{inr}^{-1} : \mathcal{A}(\tau_1 + \tau_2) \to \mathcal{A}(\tau_1) \), and similarly for \( \text{inr} \). Explicitly, \( \text{inr}^{-1} \) is the linear map with defining equations \( \text{inr}^{-1}(\text{inl} x) = x \) and \( \text{inr}^{-1}(\text{inr} x) = 0 \).

- We use the notation \( \text{id}_{\mathcal{A}(\tau)} \) to refer to the identity linear map \( P \mapsto \Phi(\tau) \to \Phi(\tau) \).

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

\[
\Phi(\ell : (\text{list}(1); P)) = 1 \cdot \phi_{[tt]}(\ell : \text{list}(1)) + 2 \cdot \phi_{[tt;tt]}(\ell : \text{list}(1)) = n + 2 \binom{n}{2} = n^2.
\]

**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 \( \triangleleft \) is the linear map \( \mathcal{A}(\mu \alpha.\tau) \to \mathcal{A}(\mu \alpha.\tau / \alpha) \) corresponding to the function from basis elements \( I(\mu \alpha.\tau) \) to \( I(\mu \alpha.\tau / \alpha) \) defined by

\[
\triangleleft_{\text{end}} \triangleleft \mathcal{C}(\mu \alpha.\tau / \alpha) \triangleleft (\text{fold i}) \triangleleft i + \mathcal{M}(\alpha.\tau)(\text{fold i}).
\]

In words, \( \triangleleft \text{end} \) is the constant index, and \( \text{fold i} \) refers to evaluation at both the current constructor (the \( i \) term) as well as all immediate children (the \( \mathcal{M}(\alpha.\tau)(\text{fold i}) \)) 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 \( v \in V(\mu \alpha.\tau) \),

\[
\Phi(\text{fold } v : \langle \mu \alpha.\tau ; P \rangle) = \Phi(v : \langle \mu \alpha.\tau / \alpha ; \triangleleft P \rangle).
\]

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

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

**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 \( \gamma : \mathcal{A}(\tau) \to \mathcal{A}(\tau) \to \mathcal{A}(\tau) \). 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].

\[
\gamma_{\mu \alpha.\tau}(\text{fold i}, \text{fold j}) \triangleq \text{fold } \gamma(\mu \alpha.\tau / \alpha, \mu \alpha.\tau / \alpha)(i, j) + \text{fold } \gamma(\mathcal{M}(\alpha.\tau)(\text{fold i}), \mu \alpha.\tau / \alpha)(j) + \text{fold } \gamma(\mathcal{N}(\alpha.\tau)(\text{fold i}, \text{fold j})).
\]

Here \( \mathcal{N} \) is like \( \mathcal{M} \), but places two indices at two occurrences of \( \alpha \). 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 \( P, Q \in A(\tau) \) and \( v \in \nu(\tau), \Phi(v : (\tau; P)) \cdot \Phi(v : (\tau; Q)) = \Phi(v : (\tau; P \cdot Q)) \).

**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
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 cost annotations on either side. □

IV. LANGUAGE & TYPE SYSTEM

Our language is essentially eager FPC [20], [23] Chapter 20] with a 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 \( (e, q) \rightarrow (e', q') \) 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 reduced added compared to a pure call-by-value
language is that for tick \( \{q\} \), as follows:

\[
q \geq q_0 \\
\text{tick}\{q_0\}, q \rightarrow (tt, q - q_0)
\]

The judgement \( (e, q) \rightarrow^* (e', q') \) 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
\( e \rightarrow e' \triangleq \exists q, q'. (e, q) \rightarrow (e', q') \) and denote the transitive
reflexive closure of that as \( e \rightarrow^* e' \).

B. Type judgements

Type judgements in our system are as follows:

Input annotation on \( \Gamma \):

\[
\Gamma; P \vdash c : \tau ; Q
\]

Cost model:

\[
\Gamma ; \cdot \vdash e : \tau
\]

Remainder annotation on \( \Gamma \):

\[
\Gamma ; \cdot \vdash e : \tau
\]

This reads “in the context \( \Gamma \) with \( P \) resources, under cost
model \( c \), \( e \) has type \( \tau \) 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 \( \mathcal{I}(\Gamma) \triangleq \prod_{\tau \in \Gamma} \mathcal{I}(\tau) \) and \( A(\Gamma) \) and \( C(\Gamma) \) as the corresponding extensions of \( A(\tau) \) and
\( C(\tau) \); note that \( A(\Gamma) \cong \mathcal{T}_{x \cdot \cdot \cdot \cdot \cdot} A(\tau) \). We may notationally
elide coercions between isomorphic semimodules, such as
\( A(\Gamma_1, \Gamma_2) \cong A(\Gamma_1) \otimes A(\Gamma_2) \) and \( A(x : \tau) \cong A(\tau) \). Another
annotation operations (all of which are linear maps) include:

- The “projection operator” \( \pi_i : A(\Gamma_1, \Gamma_2) \rightarrow A(\Gamma_1) \),
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” \( \pi_{y,x}^*(\cdot, \cdot) : A(\Gamma_1, \tau_1, \tau_2) \rightarrow A(\Gamma, y : \tau_1 \times \tau_2) \) defined just by reassociating,
because \( A(\Gamma_1, \tau_1, \tau_2) \cong A(\Gamma) \otimes A(\tau_1 \times \tau_2) \).

- An extension of the shift operator to context annotations,
\( \theta_x = (id_{A(\tau)} \otimes \langle \cdot \rangle) : A(\Gamma, x) : \alpha \cdot \tau \rightarrow A(\Gamma, x) : \langle \mu \cdot x, \alpha \cdot \tau \rangle \), i.e., shifting is applied to \( x \).

- An extension of the sharing operator to context annotations,
\( \times_y \theta_y^*(\cdot, \cdot) : A(\Gamma, y) : \tau \times \cdot \rightarrow A(\Gamma, z : \tau) \), 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],
but introduced in the programmatic AARA setting by Kahn and Hoffmann [26]. Remainder contexts contain both
the typing context \( \Gamma \) 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 \( (\cdot \rightarrow^*) \), while the
latter refers to a pure execution \( (\cdot \rightarrow^*) \). 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

\[ \vec{\Gamma}; P \vdash_c e : \tau; Q \]

\[
\begin{array}{ll}
\text{T:TickPos} & q \geq 0 \quad P \otimes C(o : 1) = Q + q \cdot C(\vec{\Gamma}, o : 1) \\
\vec{\Gamma}; P \vdash_c \text{tick}(q) : 1; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:TickNeg} & q \geq 0 \quad P \otimes C(o : 1) + q \cdot C(\vec{\Gamma}, o : 1) = Q \\
\vec{\Gamma}; P \vdash_c \text{tick}(-q) : 1; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Var} & \forall_{\tau}(Q) = P \\
\vec{\Gamma}, x : \tau; P \vdash_c x : \tau; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Unit} & P \otimes C(o : 1) = Q \\
\vec{\Gamma}; P \vdash_c \text{tt} : 1; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:InR} & P = (\text{id}A(\vec{\Gamma}) \otimes (Y \circ (\text{id}A(\tau_v) \otimes \text{in1})))((Q) \\
\vec{\Gamma}, x : \tau_r; P \vdash_c \text{in} x : \tau_r + \tau_r; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:InL} & P = (\text{id}A(\vec{\Gamma}) \otimes (Y \circ (\text{id}A(\tau_u) \otimes \text{in1})))((Q) \\
\vec{\Gamma}, x : \tau_l; P \vdash_c \text{in} x : \tau_l + \tau_r; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:CaseSum} & x' \text{ fresh} \quad \forall_{\tau}(P') = P \\
\vec{\Gamma}, x : \tau_r; S \vdash_c e_2 : \tau; Q \quad \vec{\Gamma}, x : \tau_r; S \vdash_c e_2 : \tau; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:CaseProd} & P = (\text{id}A(\vec{\Gamma}) \otimes (Y \circ (\text{id}A(\tau_u) \otimes \text{pair})))((Q) \\
\vec{\Gamma}, x : \tau_r \times \tau_2; x_1, x_2 : \tau_2; P \vdash_c \text{case}(x_1, x_2, e) : \tau; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Let} & P = \text{id}A(\vec{\Gamma}) \otimes (\text{pair} \circ \text{pair} \circ \text{id}A(\tau_1 \times \tau_2))((Q) \\
\vec{\Gamma}, x_1 : \tau_1, x_2 : \tau_2; P \vdash_c \text{let}(x_1, x_2) : \tau_1 \times \tau_2; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Prod} & P = \text{id}A(\vec{\Gamma}) \otimes (\text{pair} \circ \text{pair} \circ \text{id}A(\tau_1 \times \tau_2))((Q) \\
\vec{\Gamma}, x : [\mu\alpha.\tau/\alpha] ; P \vdash_c \text{fold} x : \mu\alpha.\tau; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Fold} & P = \forall_{\tau}(Q) \\
\vec{\Gamma}, x : [\mu\alpha.\sigma/\alpha] ; P \vdash_c \text{fold} x : \mu\alpha.\sigma; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Fun} & \forall(S, R) \in T_\theta, \vec{\Gamma}, f : \langle \tau_1 \rightarrow \tau_2, \Theta_{\theta}, \Theta_{\theta} \rangle; x : \tau_1 ; C(\vec{\Gamma}) \otimes R \vdash_c e : \tau_2 ; C(\vec{\Gamma}) \otimes S \\
\forall(S, R) \in T_\theta, \vec{\Gamma}, f : \langle \tau_1 \rightarrow \tau_2, \Theta_{\theta}, \Theta_{\theta} \rangle; x : \tau_1 ; C(\vec{\Gamma}) \otimes R \vdash_c e : \tau_2 ; C(\vec{\Gamma}) \otimes S \\
\vec{\Gamma}; P \vdash_c \text{fun}(f, x, e) : \langle \tau_1 \rightarrow \tau_2, \Theta_{\theta}, \Theta_{\theta} \rangle; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:App} & \forall \in \text{C}(\vec{\Gamma}).(\pi_{\tau_1, \tau_2}(P), \pi_{\tau_1, \tau_2}(Q)) \in \Theta_{\theta} \\
\forall \notin \text{C}(\vec{\Gamma}).(\pi_{\tau_1, \tau_2}(P), \pi_{\tau_1, \tau_2}(Q)) \notin \Theta_{\theta} \\
\vec{\Gamma}, f : \langle \tau_1 \rightarrow \tau_2, \Theta_{\theta}, \Theta_{\theta} \rangle; x : \tau_1 ; P \vdash_c \text{app}(f, x) : \tau_2; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Relax} & R \vdash_c e : \tau; S \\
R \leq P \quad Q \leq S \\
\vec{\Gamma}; P \vdash_c e : \tau; Q \\
\end{array}
\]

\[
\begin{array}{ll}
\text{T:Augment} & R \vdash_c e : \tau; S \\
R = R + T \quad Q = S + T \otimes C(\tau) \\
\vec{\Gamma}; P \vdash_c e : \tau; Q \\
\end{array}
\]

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 \( \langle \tau_1 \rightarrow \tau_2, \Theta_{\theta}, \Theta_{\theta} \rangle \), where \( \Theta_{\theta}, \Theta_{\theta} \subseteq \text{C}(\tau_1) \times \text{C}(\tau_1 \times \tau_2) \). \( \Theta \) and \( \Theta_{\theta} \) 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 \( \Phi(v : \langle \tau_1 \rightarrow \tau_2 \rangle) \) resources are needed; once it returns with value \( v' \), \( \Phi(\text{pair}(v; v') : \langle \tau_1 \times \tau_2 \rangle) \) resources are returned as well. The argument \( \tau_1 \) is mentioned in the output potential to enable the remainder contexts discussed in §IV-B \( \Theta_{\text{cf}} \) serves a similar purpose for the cost-free model discussed in §IV-B \( (P, Q) \in \Theta_{\text{cf}} \) implies that \( \Phi(\text{pair}(v; v') : \langle \tau_1 \times \tau_2 \rangle) \geq \Phi(v : \langle \tau_1 \rangle) \), 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.
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 \( \alpha \).

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 \( \square \).

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 self(\( \tau \)) \( \triangleq \mu \alpha.\langle \alpha \rightarrow \tau, \{(0, 0)\}, \{(0, 0)\}\rangle \) 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.

```plaintext
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` : \( (b \times 'a \rightarrow 'b) \rightarrow 'b \times \text{list}('a) \rightarrow 'b \) be the left fold function on lists, attach can be defined by:

```plaintext
let rec attach dname (acc, fs) = match fs with
| File (name, _) -> (dname, Iname)
| Dir (subdirname, fss) -> (dname, subdirname) :: 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 `attach` : \( S \rightarrow \text{list}(S \times S) \times \text{filesystem} \rightarrow \text{list}(S \times S) \) using our indices, where \( S \) is the string type. The only nonzero annotation required is that the second argument must provide \( 1 \cdot ([], \text{File}(*, *)) + 1 \cdot ([], \text{Dir}(*, [])) \) potential, where * 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`\( ^4 \) to demonstrate that more novel cost bounds are expressible with our indices.

```plaintext
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 trans : \( \text{list}(S \times S) \times \text{filesystem} \rightarrow \text{list}(S \times S) \) using our indices. The only nonzero annotation required is that the argument must provide \( 1 \cdot ([], \text{Dir}(*, [\text{File}(*, *)])) + 1 \cdot ([], \text{Dir}(*, [\text{Dir}(*, [])])) \) 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 \( ; : p \vdash cP \in \tau; Q \). For any \( r \geq 0 \) and execution \( (e, p + r) \rightarrow^{*} (e', q) \), we either:

- \( e' \in \nu(\tau) \) and \( q \geq r + \Phi(e' : \tau; Q) \), or
- there exists some \( e'' \) and \( q' \) such that \( (e', q') \rightarrow (e'', q') \).

**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:

\[
\nu''(\tau_{1} \rightarrow \tau_{2}, \Theta_{cp}, \Theta_{ct})\|_{n+1}(v) \triangleq \exists f, x, e, v = \text{fun}(f, x, e) \land \forall m \leq n.
\]

\[
(\forall(P, Q) \in \Theta_{cp}, \forall q \geq 0, \nu'. \nu'(\tau_{1})_{m}(v') \Rightarrow WP_{cp}[\lambda(m', v'', q'). \nu''(\tau_{2})_{m}(v'') \land q' \geq q + \Phi(\text{pair}(v', v'') : \langle \tau_{1} \times \tau_{2}; Q \rangle)_{m}]
\]

\[
((v'/x, v'/f)e, q + \Phi(v' : \langle \tau_{1}; P \rangle) \land\]

\[
(\forall(P, Q) \in \Theta_{ct}, \forall q \geq 0, \nu'. \nu'(\tau_{1})_{m}(v') \Rightarrow WP_{ct}[\lambda(m', v'', q'). \nu''(\tau_{2})_{m}(v'') \land q' \geq q + \Phi(\text{pair}(v', v'') : \langle \tau_{1} \times \tau_{2}; Q \rangle)_{m}]
\]

where the cost-paid weakest precondition relation is:

\[
WP_{cp}[\Psi]_{n+1}(e, q) \triangleq \{(e \text{ val} \land \Psi(n + 1, e, q)) \lor ((\exists e', q'. (e, q) \rightarrow (e', q')) \land \forall e', q'. (e, q) \rightarrow (e', q') \Rightarrow WP_{cp}[\Psi]_{n}(e', q'))
\]

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. \( \square \)

---

\(^4\)This is adapted from an example given by Hoffmann [27].
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]. 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], 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 [51], object-oriented [32], probabilistic [34], parallel computation [36], digital contract protocols [38], and lower bound costs [33]. 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], refinement types [14], modal types [3], sized types [40], [41], annotation-based systems [42], 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 automatability for richer features – at the extreme other end from AARA one finds type-based proof logics [43], which require significant user work to prove cost bounds.

The term-rewriting space [45], [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 [35], [39], 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 resource 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


