Semantically Sound Resource Analysis with Nested Recursive Types

JESSIE GROSEN, DAVID M. KAHN, and JAN HOFFMANN

The goal of automatic resource bound analysis is to statically infer symbolic bounds on the resource consumption of the evaluation of a program. Two longstanding challenges for automatic resource analysis are the inference of bounds that are functions of complex custom data structures and smooth integration with manual bound analysis. This article builds on type-based automatic amortized resource analysis (AARA) to address these two challenges. 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. A key component of AARA are resource functions that generate the space of possible bounds for values of a given type while enjoying necessary closure properties.

Existing work on AARA defined such functions for many data structures such as lists of lists but the question of whether such functions exist for arbitrary data structures remained open. This work answers this questions positively by uniformly constructing resource polynomials for general algebraic data structures defined by possibly-nested 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 the techniques.

To integrate automatic and manual bound analysis, the article proposes a novel notion of semantic type soundness for AARA that relates type judgements—and thus bounds—with a small-step cost semantics. First, semantic type soundness is used to modularly prove the soundness of the individual type rules for FPC. Second, it is shown how semantic type soundness can be used to prove a polynomial bound that is then available in AARA type derivations. Semantic type soundness, resource polynomials, the AARA type system, the cost semantics, and the soundness proof have been mechanized in the Coq Proof Assistant.

ACM Reference Format:
Jessie Grosen, David M. Kahn, and Jan Hoffmann. 2022. Semantically Sound Resource Analysis with Nested Recursive Types. 1, 1 (March 2022), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 [Dal Lago and Gaboardi 2011; Dal Lago and Petit 2013; Rajani et al. 2021], deriving and solving recurrence relations [Albert et al. 2007; Cutler et al. 2020; Kavvos et al. 2019], or other static analysis [Avanzini et al. 2015; Chatterjee et al. 2019; Gulwani et al. 2009]. They can derive (worst-case) upper bounds [Kincaid et al. 2017a; Wang et al. 2017] (best-case) lower bounds [Albert et al. 2013; Ngo et al. 2017], and relational bounds on the difference of the cost of two programs [Radiček et al. 2017], considering resources like time or memory.

Authors' address: Jessie Grosen, jgrosen@cs.cmu.edu; David M. Kahn, davidkah@cs.cmu.edu; Jan Hoffmann, jhoffmann@cmu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
XXXX-XXXX/2022/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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) [Hoffmann et al. 2011; Hofmann and Jost 2003; Jost et al. 2010], which has emerged as resource analysis that 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 [Hoffmann et al. 2017], 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 (like left and right injections).

AARA for functional programs is based on a type system and type derivations serve a 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 that can become available during the evaluation. The key innovation that enables inference of non-linear bounds with linear programming is the use of the potential method of amortized analysis with a carefully selected set of resource functions that serve as templates for the potential functions for values of a given type.

Despite its benefits, state-of-the-art AARA still has some fundamental limitations that limit its applicability to real-world code:

(1) Its lack of support for general, potentially nested recursive types.[1]
(2) Its inability to integrate resource bounds that cannot be automatically inferred.

As an example of the former, examine the function sort_lefts_tree, similar to the function above but with lists swapped for *rose trees*:

```
type 'a tree = Tree of 'a * 'a tree 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 deficiency. Extending polynomial AARA to handle nested recursive types has been an open problem since it was introduced by Hoffmann and Hofmann [2010]. The core challenge lies in finding a class of potential functions for these types that is expressive but constrained enough to be closed under the operations necessary for typing.

We address this longstanding gap by introducing a notion of resource polynomials for recursive types that meets the requirements to be used in AARA. We draw inspiration from past approaches, but ultimately adopt a more algebraic view that we believe better follows the structure of the types. In particular, the indices that generate the base polynomials match the values they classify

[1] Nested recursive types may also be instead defined *mutually*: Bekić's lemma says the latter can be transformed into the former.

nearly exactly. Our resource polynomials are a generalization of all previously proposed polynomial resource functions of AARA [Hoffmann et al. 2011, 2017; Hoffmann and Hofmann 2010; Jost et al. 2010] 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.

The second aforementioned issue with AARA, its inability to integrate manually derived bounds, leads to the requirement that the entire program be automatically analyzable. This is made evident by the function bubblesort_lefts_list, which is the same as the first function save its replacement of quicksort for bubblesort, which recurses until *the list does not change*:

```
let bubblesort_lefts_list (l : (int + bool) list) =
  bubblesort (filter_map find_left l)
```

AARA cannot derive a bound for bubblesort and thus cannot derive a bound for bubblesort_lefts_list either. This is because bubblesort's resource usage depends deeply on its semantic behavior as opposed to just its structural behavior. While an SMT solver could perhaps help solve this problem in the particular case of bubblesort, it would be unable to address it in the case of more complex programs requiring more insight to prove. Unfortunately, this means that any program that uses one of these routines anywhere cannot take advantage of AARA to derive a bound, even if the bound for those functions were expressible within AARA's type system.

We tackle this problem by introducing a *semantic typing* relation, showing its adequacy, and proving that our typing rules are compatible with it, allowing the composition of syntactic and semantic proofs of typing for programs. To do this, we develop a type-indexed unary logical relation that generalizes the usual notion of soundness to our setting but is still simple enough to reasonably prove functions against. We also mechanically prove these results in Coq, offering a clear setting in which semantic resource bound proofs for functions can be verified.

In summary, our contributions include:

- We solve the longstanding problem of defining resource polynomials for general recursive types (§3) and use them to develop a state-of-the-art multivariable AARA type system that can derive bounds that are functions of general algebraic data structures (§4).
- We construct a notion of semantic type soundness of an expression using a logical relation (§5.1), then prove our type system to be compatible with that definition (§5.2), enabling the sound integration of manually proven bounds.
- We discuss the Coq mechanization of semantic type soundness, resource polynomials, the AARA type system, the cost semantics, and the soundness proof (§5.4).

To assist the reader in keeping track of notation, a glossary is provided in Appendix A (in the supplementary material).

2 OVERVIEW

To start with, we review AARA (§2.1), its potential functions for lists (§2.2), present the intuition behind our extension to nested inductive types (§2.3), and discuss the possibilities for semantically-proven function bounds (§2.4).

2.1 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 [1985] 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 function 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 gradually build up the motivating example shown in the introduction. As then, say we are interested in tracking the number of cons cell creations as our cost model. To start, consider the standard list function filter_map : $(\tau \to \mathsf{option}(\sigma)) \to \mathsf{list}(\tau) \to \mathsf{list}(\sigma)$, which can be 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. First 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.

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

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 in general important for the compositionality of the analysis. To see how the potential of the result can be used consider the following typing.

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

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 no potential is needed. After the evaluation there is 1 unit left if the result is of the form 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.

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

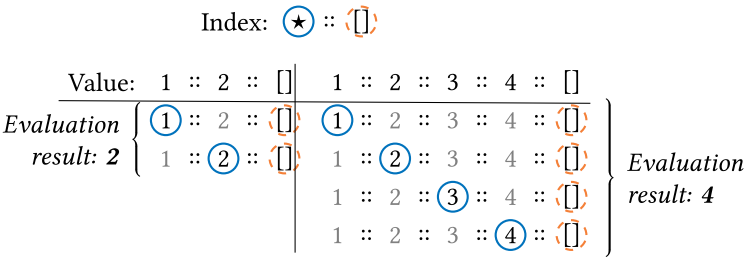
$$r_1 \geq p_0 + 1, r_2 \geq q_1, r_3 \geq q_2, p_1 \geq p_1', q_3 + p_0' \geq r_4$$

An equivalent constraint set can be automatically derived with local syntax-directed type rules. An essential requirement is that the transfer of potential from the list to its head and tail can be expressed with linear constraints. In the case of linear potential functions, this is straightforward since the annotation of the head is the annotation of the element type the annotation of the tail is the annotation of the matched list.

2.2 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 [Hoffmann and Hofmann 2010]. What is less clear is how to maintain the aforementioned requirement for only linear constraints to come of destructing a list. The answer turns out to be 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 $\lhd(q_0,\ldots,q_m) = (q_0+q_1,\ldots,q_{m-1}+q_m,q_m)$. This is a *linear* function that specifies how to preserve potential–that is, evaluating $\lhd(\vec{q})$ 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–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 [Hoffmann et al. 2011]. 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 list index $[\star]$ on lists, referring to linear potential n, and consider evaluating it on two lists of different lengths:



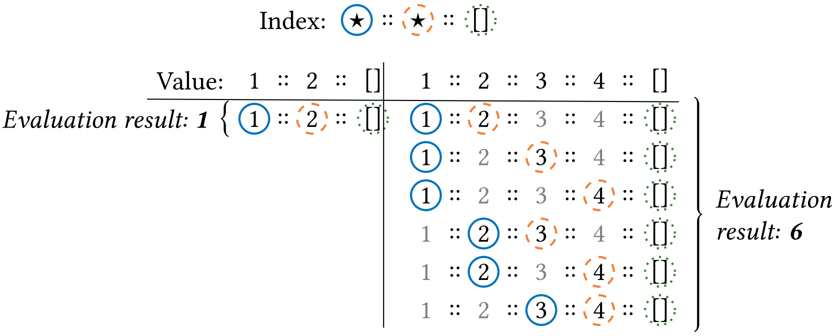
Note that, as demonstrated by the two circles in each evaluation, there are two matches in each: a cons cell, and the ending nil. The critical aspect of list indices' evaluation is that it can be phrased purely locally in terms of the heads and tails of the index and list elements:

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

which 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: $\lhd(i :: is) = (i, is) + (\star, i :: is)$, 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 $\langle \mathsf{list}^1(int^1 + bool^0), 0\rangle \longrightarrow \langle \mathsf{list}^1(int), 0\rangle$. 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, the structure of these evaluations demonstrate that this index corresponds to $\binom{n}{2}$. Thus, given that we know quicksort has cost n^2, we can express its 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.

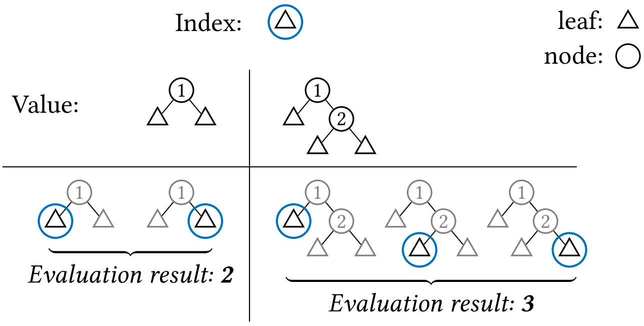
2.3 Extending to nested recursive types

However, this technique does not obviously generalize to nested inductive types. Jost et al. [2010] handles potential on nested inductives, but only in the very restricted setting of univariate linear potential, which amounts to just counting constructors. Hoffmann et al. [2011] and its successors handle more expressive potential functions, but don't support nested inductives and treat even just binary trees as lists for potential purposes, more or less–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.

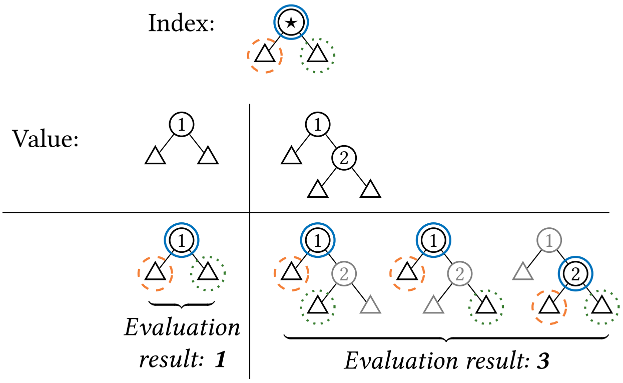
Let's try something else. For one, we know we absolutely must preserve some sort of linear shift function. Another hint comes from Hoffmann et al. [2011], who observe that their indices for a type τ are "[i]nterestingly, but as we find coincidentally, [...] essentially the meaning of $[\tau]$ with every atomic type replaced by unit." We find that they were on to something after all. We consider indices that correspond *exactly* to the values of the type they describe.[2] To build intuition, we'll first give show some examples on specific data types before as we get to describing the general case.

2.3.1 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:

[2] Well, *almost* exactly, as we will see in §3.1.



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; so far so good. 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:

$$\phi_{\triangle}(\triangle) = 1 \qquad \phi_{\underset{i_1\quad i_2}{\textcircled{i}}}(\triangle) = 0 \qquad \phi_{\triangle}\left(\underset{t_1\quad t_2}{\textcircled{i}}\right) = \phi_{\triangle}(t_1) + \phi_{\triangle}(t_2)$$

$$\phi_{\underset{i_1\quad i_2}{\textcircled{i}}}\left(\underset{t_1\quad t_2}{\textcircled{v}}\right) = \phi_i(v)\cdot\phi_{i_1}(t_1)\cdot\phi_{i_2}(t_2) + \phi_{\underset{i_1\quad i_2}{\textcircled{i}}}(t_1) + \phi_{\underset{i_1\quad i_2}{\textcircled{i}}}(t_2)$$

2.3.2 *Eureka!*

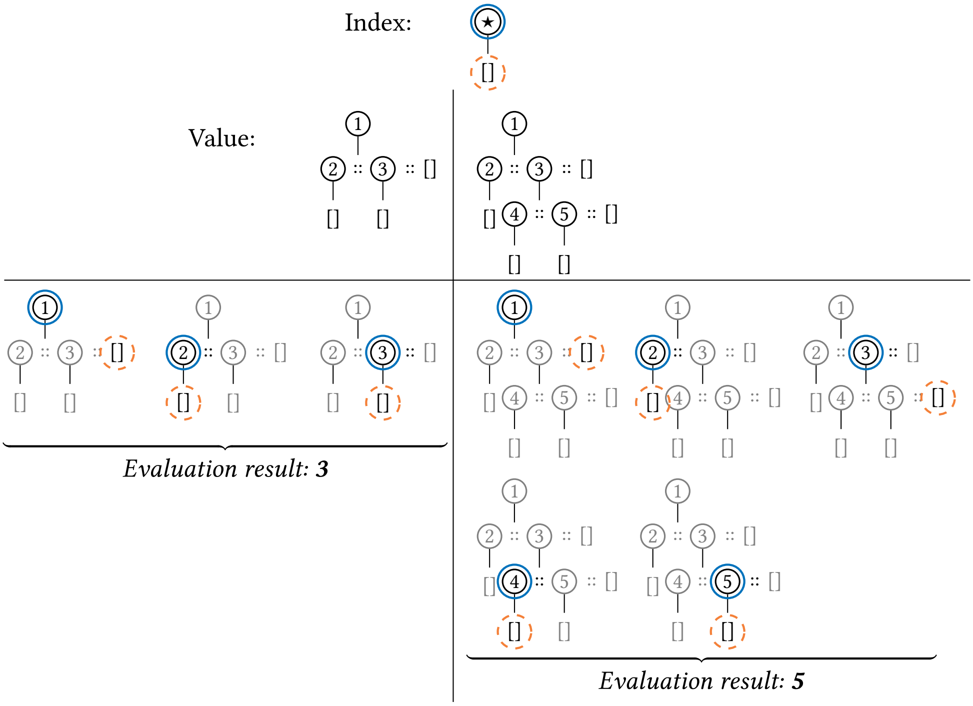
The insight for the general case, then, is to notice this correspondence between the rules for lists and the rules for binary trees:

$$\begin{aligned}
\phi_{i::is}(v :: vs) \quad &= \quad \phi_i(v)\cdot\phi_{is}(vs) \quad &&+ \quad \phi_{i::is}(vs) \\
\phi_{\underset{i_1\quad i_2}{\textcircled{i}}}\left(\underset{t_1\quad t_2}{\textcircled{v}}\right) \quad &= \quad \phi_i(v)\cdot\phi_{i_1}(t_1)\cdot\phi_{i_2}(t_2) \quad &&+ \quad \left[\phi_{\underset{i_1\quad i_2}{\textcircled{i}}}(t_1) + \phi_{\underset{i_1\quad i_2}{\textcircled{i}}}(t_2)\right]
\end{aligned}$$

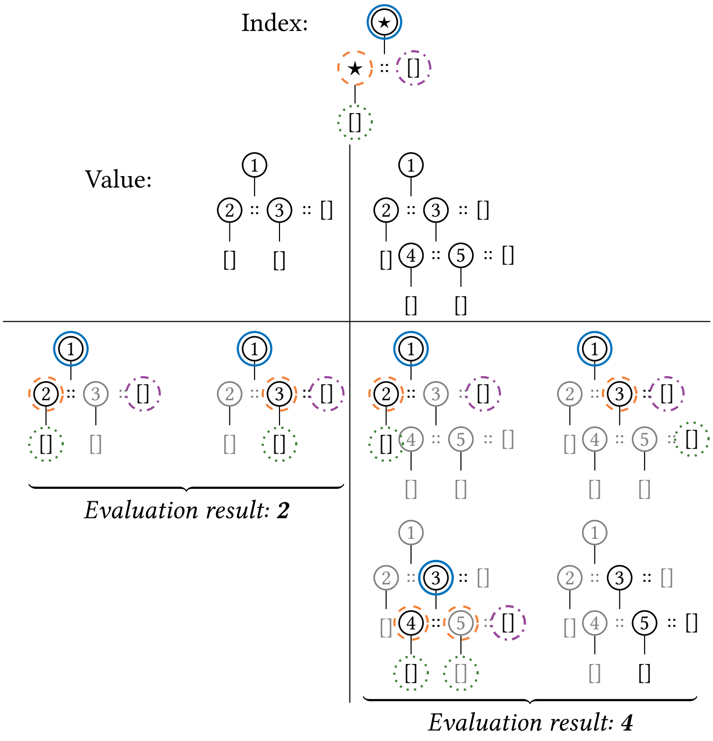
Key Intuition. To evaluate an index at a constructor, first evaluate it at the immediate constructor, then add that to the evaluation of the original index at all direct children.

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 §3.1 to define this formally, but we will at least address one ambiguity in that specification: what are the "direct children" of a constructor?

2.3.3 The prize: rose trees. The direct children of a cons cell or tree node are readily apparent, but they are less obvious for our original motivating data type, the rose tree. Let us again turn to examples, starting with the simplest index:



And a more complex index:



Posed again, how do we specify the direct children of a rose tree node? The wonderful trick is to piggyback off of the list's notion of direct children: a rose tree node's direct children is any node that appears in its list. That sounds almost tautological, but what else do you expect from mutual recursion?

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 §3.1.

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 $2 \cdot \mathsf{Tree}(\mathsf{inl}\,\star, []) + 1 \cdot \mathsf{Tree}(\mathsf{inr}\,\star, [])$. 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.

2.4 Integrating manually-proven functions

While we now have the intuition to be able to type filter_map_tree, we are no closer to typing bubblesort!

```
let rec bubble l =
  match l with
  | x :: y :: l' → if x > y
    then y :: bubble (x :: l')
    else x :: bubble (y :: l')
  | _ → l;;
```

```
let rec bubblesort l =
  let l' = bubble l in
  if l = l' then l else bubblesort l'
```

Considered only structurally, it is not clear that bubblesort has any resource bound at all or that it even terminates. Semantically, by the usual proof, we can see that it allocates at most $2n(n+1)$ cons cells, where n is the length of l. This is easily expressible with AARA indices, as $2 \cdot [\star;\star] + 2 \cdot [\star]$. Thus, if we enriched AARA with some level of semantic reasoning, like with liquid types [Knoth et al. 2020], this function could perhaps have a resource bound verified, if not inferred, without adding any more resource polynomials. However, keeping the type system based purely on structural features confers many benefits–most notably, predicting whether or not a bound can be inferred is much easier. Is there another way to get the best of both worlds?

For this, we look to RustBelt for inspiration, which itself draws on a long line of work [Ahmed et al. 2010; Jung et al. 2017]. RustBelt uses a semantic model of Rust's types not only to prove soundness of Rust's core typing rules, but also to allow "unsafe" code to be verified to be sound against Rust's types using reasoning that goes beyond what the core type system would allow.

We adopt a similar approach in this work. We define a notion of semantic typing based on a logical relation that allows for proofs of well-typedness more involved than the syntactic rules of the language. That way, we can write down a resource-annotated type for bubblesort and use the definition of semantic typing to obtain a logical statement that, if proven by any means, allows us to compose bubblesort with programs otherwise typed using a typical syntactic AARA approach.

3 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' enum-like inductive 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.

3.1 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 Figure 1. The types presented are standard, save the arrow type–the details of which are irrelevant to the resource polynomials and explained in §4. We also give, in Figure 1c, inference rules for the set of syntactically valid values $\mathcal{V}(\tau)$ for a given type τ.

Following our running examples, we may define the types 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 2a shows inference rules for the set of indices $\mathcal{I}(\tau)$ for a given type τ. They nearly exactly mirror the syntactic values $\mathcal{V}(\tau)$,

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

(a) The values of the language.

$$\begin{array}{lllr} \text{Types} & \tau & ::= & \alpha \qquad \text{Type variable} \\ & & \mid & \mathbf{1} \qquad \text{Unit} \\ & & \mid & \tau_1 \times \tau_2 \qquad \text{Product} \\ & & \mid & \tau_1 + \tau_2 \qquad \text{Sum} \\ & & \mid & \langle \tau_1 \rightarrow \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle \qquad \text{Arrow} \\ & & \mid & \mu\alpha.\, \tau \qquad \text{Isorecursive} \end{array}$$

(b) The types of the language.

$$\frac{}{\mathsf{tt} \in \mathcal{V}(\mathbf{1})} \qquad \frac{v_1 \in \mathcal{V}(\tau_1) \qquad v_2 \in \mathcal{V}(\tau_2)}{\mathsf{pair}(v_1;\, v_2) \in \mathcal{V}(\tau_1 \times \tau_2)} \qquad \frac{v_1 \in \mathcal{V}(\tau_1)}{\mathsf{inl}\, v_1 \in \mathcal{V}(\tau_1 \times \tau_2)} \qquad \frac{v_2 \in \mathcal{V}(\tau_2)}{\mathsf{inl}\, v_2 \in \mathcal{V}(\tau_1 \times \tau_2)}$$

$$\frac{}{\mathsf{fun}(f, x.\, e) \in \mathcal{V}(\langle \tau_1 \rightarrow \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle)} \qquad \frac{v \in \mathcal{V}([\mu\alpha.\, \tau/\alpha]\tau)}{\mathsf{fold}\, v \in \mathcal{V}(\mu\alpha.\, \tau)}$$

(c) The *syntactic* typing of values, $\mathcal{V}(\tau)$, defined as the least fixpoint satisfying these inference rules.

Fig. 1. Types and values

with the addition of an "end" index for recursive types.[3] One possible intuition for an index is to view it as a pattern in a pattern match that specifies specify a shape that values are compared against. However, whether or not a pattern matches 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(τ) that match against any list value exactly once; Cons(tt, Nil) matches against any list$(\mathbf{1})$ value exactly as many times as the length of the list; and Node(tt, Nil) matches against any tree$(\mathbf{1})$ value exactly as many times as there are nodes in the tree. These "counts" are formalized in §3.1.

Constant index set. A function is given in Figure 2b that defines a set of indices $C(\tau)$ for any type τ such that the sum of their counts on any value of type τ is exactly 1. The definition proceeds easily from the definition of index evaluation, which will be given shortly (but unfortunately depends on this definition, posing a chicken-and-egg problem for presentation). (We also include a definition of C for $\top_\mu$, a pseudotype representing any recursive type, that occurs purely during evaluation of $\mathcal{M}$.)

Following our running examples, we have $C(\mathsf{bool}) = \{\mathsf{True}, \mathsf{False}\}$, which indeed encompasses all possible values of type bool, and $C(\mathsf{list}(\tau)) = C(\mathsf{tree}(\tau)) = \{\mathsf{end}\}$, 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 2c, where τ is a type that is open in 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. Since this is at the core of our approach, we explain the cases in detail:

[3] Several parts of the type system rely on being able to describe constant potential for a value; we thus add end because this would otherwise not hold for some recursive types.

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

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

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

(a) The indices $\mathcal{I}(\tau)$ that name base polynomials for a given type τ. $\mathcal{I}(\tau)$ is defined as the least fixpoint closed under these inference rules.

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

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

(b) The constant set of indices $C(\tau)$, which sum to 1 when evaluated on any value of type τ.

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

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

(c) The set of indices $\mathcal{M}\{\alpha.\tau\}(i)$ which correspond to placing the index i at each occurrence of α in τ.

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

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

(d) Evaluation function $\phi_i(v : \tau)$ on value v of type τ for base polynomial of index i.

Fig. 2. Fundamental base polynomial index constructions.

α. We have found an occurence of α, so i goes here.

$\top_{\mu}$. This represents some recursive type *other* than α; while there may be more values of type α within a value here, they will be covered by some other instance of α.

$\mathbf{1}$. 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 there could be α values 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.

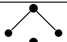
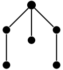
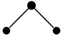
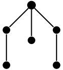
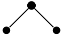
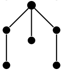
$\tau_1 \longrightarrow \tau_2$. We treat functions as opaque, so there are no α values in here.

$\mu\beta.\,\tau$. Here is the case critical for handling nested recursive types, but it turns out to be elegantly simple. As observed in §2.3.3, 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 either True or False 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 Figure 2d, 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.

We go into more detail for our running examples this time, in a tabular format:

Type τ	Index $i \in \mathcal{I}(\tau)$	Value $v \in \mathcal{V}(\tau)$	Result $\phi_i(v : \tau)$	Explanation
bool	False	True	1	These examples demonstrate the simple "pattern match"-like behavior when no fold indices are present.
		False	0	
	True	False	0	
		True	1	
list(**1**)	[]	[tt; tt]	1	Just as in previous work, evaluating an index of length k on a unit list of length n results in $\binom{n}{k}$.
		[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(**1**)	end		1	Index evaluation on rose trees works exactly as we wanted in §2.3.3, allowing for both linear and higher-degree polynomial potential.
			1	
	Tree(tt, [])		3	
			6	
	Tree(tt, [Tree(tt, [])])		2	
			7	

With index evaluation now defined, we may characterize the motivating property of the constant index set:

Lemma 3.1 (Constant indices sum). *For all types τ and values $v \in \mathcal{V}(\tau)$, $\sum_{i \in C(\tau)} \phi_i(v : \tau) = 1$.*

Proof. By induction on τ. □

Note that distinct indices may sometimes refer to the same base polynomial. For example, the two indices end and Nil for the type list(τ) both represent the constant function. Though this feels unsatisfying, it causes no practical issues in our development.

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 coefficients.

3.2 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 3.2 (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 such 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

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

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:

- Set coercion to annotation for a (finite) set of indices $B \subseteq \mathcal{I}(\tau)$, where $b_i = 1$ if $i \in B$ and 0 otherwise.
- Preorder $P \leq Q$, defined by $p_i \leq q_i$ for all i. Note that $P \leq Q$ is equivalent to whether there exists an R such that $P = Q + R$, and thus $\Phi(v : \langle \tau; \cdot \rangle)$ respects the order.
- Also note that because $\mathcal{I}(\tau_1 \times \tau_2) = \mathcal{I}(\tau_1) \times \mathcal{I}(\tau_2)$, 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.
- In the same vein, 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.

Then, for example, if we want n^2 potential for a unit list of length n, we can use the annotation $P = 1 \cdot [\mathsf{tt}] + 2 \cdot [\mathsf{tt}; \mathsf{tt}]$, so that $\Phi([v_1; \ldots; v_n] : \langle \mathsf{list}(\mathbf{1}); P \rangle) = 1 \cdot \phi_{[\mathsf{tt}]}([v_1; \ldots; v_n] : \mathsf{list}(\mathbf{1})) + 2 \cdot \phi_{[\mathsf{tt};\mathsf{tt}]}([v_1; \ldots; v_n] : \mathsf{list}(\mathbf{1})) = n + 2\binom{n}{2} = n^2$.

Shifting. A key requirement for our resource polynomials is the ability to fold and unfold recursive values while maintaining equal potential. Maintaining this ability was a key design constraint while constructing our system. We can accomplish this with the *additive shift* operator.

Definition 3.3 (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

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

We argue this definition is highly intuitive: end is the constant index, and 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). This intuition is codified in the key property we desired for this operator:

Theorem 3.4 (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 it 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 between them 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. The full definition is long and technical, but we consider it a key contribution of the paper, so we highlight the definition for the critical case, sharing two fold indices:

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

(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.

In any case, the full definition is efficiently computable and satisfies the key property stated below:

Theorem 3.5 (Share preserves potential). *For any $P, Q \in \mathcal{A}(\tau)$ and $v \in \mathcal{V}(\tau)$,*

$$\Phi(v : \langle\tau; P\rangle) \cdot \Phi(v : \langle\tau; Q\rangle) = \Phi(v : \langle\tau; P \curlyvee Q\rangle).$$

3.3 Comparison to Hoffmann et al. [2011]

From this description of potential functions, it is not entirely clear whether FancierAARA is a *generalization* of previous multivariate AARA potentials as in Hoffmann et al. [2011], or whether it is instead simply *different*. We show it is the former with the following theorem:

Theorem 3.6. *All resource polynomials representable in Hoffmann et al. [2011] are also representable in FancierAARA.*

Proof sketch. Because resource polynomials are linear combinations of base polynomials, it suffices to show that their base polynomials are representable as our annotations. We show this by induction over types; the only nontrivial case is for binary trees. After a further induction over the length of the list of indices that serves as an 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. □

The proof is extensible to the finite arity trees considered in Hoffmann et al. [2017].

4 LANGUAGE & TYPE SYSTEM

Our language is essentially FPC [Fiore and Plotkin 1994; Harper 2016] with a tick$\{q\}$ expression to express cost and an explicit let construct. The syntax is shown in Figure 3. The language is pure, except for the cost effect. 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.

4.1 Semantics

The cost semantics of the language is a standard small-step operational semantics, as presented in Figure 4. 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 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

$$\begin{array}{rccll}
\text{Variables} & f, x & & & \\
\text{Rationals} & q & & & \\
\text{Expressions} & e & ::= & x & \text{Variable} \\
 & & | & \mathsf{tt} & \text{Unit expression} \\
 & & | & \mathsf{fun}(f, x.\, e) & \text{Recursive function} \\
 & & | & \mathsf{app}(e_1;\, e_2) & \text{Function application} \\
 & & | & \mathsf{tick}\{q\} & \text{Cost collection} \\
 & & | & \mathsf{let}(e_1;\, x.\, e_2) & \text{Let expression} \\
 & & | & \mathsf{pair}(e_1;\, e_2) & \text{Product construction} \\
 & & | & \mathsf{casep}(e;\, x_1, x_2.\, e') & \text{Product destruction} \\
 & & | & \mathsf{fold}\, e & \text{Recursive folding} \\
 & & | & \mathsf{unfold}\, e & \text{Recursive unfolding} \\
 & & | & \mathsf{inl}\, e & \text{Sum construction} \\
 & & | & \mathsf{inr}\, e & \text{Sum construction} \\
 & & | & \mathsf{cases}(e;\, x_1.\, e_1;\, x_2.\, e_2) & \text{Sum destruction}
\end{array}$$

Fig. 3. Syntax of the expression language

Evaluation contexts $\boxed{\mathcal{K}}$

$$\begin{array}{rclclcl}
\mathcal{K} & ::= & \mathsf{app}(\square;\, e) & | & \mathsf{app}(v;\, \square) & | & \mathsf{let}(\square;\, x.\, e) \\
 & | & \mathsf{pair}(\square;\, e) & | & \mathsf{pair}(v;\, \square) & | & \mathsf{casep}(\square;\, x_1, x_2.\, e) \\
 & | & \mathsf{fold}\, \square & | & \mathsf{unfold}\, \square & & \\
 & | & \mathsf{inl}\, \square & | & \mathsf{inr}\, \square & | & \mathsf{cases}(\square;\, x_1.\, e_1;\, x_2.\, e_2)
\end{array}$$

Single step $\boxed{(e, q) \mapsto (e', q')}$

$$\frac{(e, q) \mapsto (e', q')}{(\mathcal{K}[e], q) \mapsto (\mathcal{K}[e'], q')} \qquad \frac{}{(\mathsf{app}(\mathsf{fun}(f, x.\, e);\, v), q) \mapsto ([v/x, \mathsf{fun}(f, x.\, e)/f]e, q)}$$

$$\frac{q \geq q_0}{(\mathsf{tick}\{q_0\}, q) \mapsto (\mathsf{tt}, q - q_0)} \qquad \frac{}{(\mathsf{let}(v_1;\, x.\, e_2), q) \mapsto ([v_1/x]e_2, q)}$$

$$\frac{}{(\mathsf{casep}(\mathsf{pair}(v_1;\, v_2);\, x_1, x_2.\, e'), q) \mapsto ([v_1/x_1, v_2/x_2]e', q)} \qquad \frac{}{(\mathsf{unfold}\,(\mathsf{fold}\, v), q) \mapsto (v, q)}$$

$$\frac{}{(\mathsf{cases}(\mathsf{inl}\, v;\, x_1.\, e_1;\, x_2.\, e_2), q) \mapsto ([v/x_1]e_1, q)} \qquad \frac{}{(\mathsf{cases}(\mathsf{inr}\, v;\, x_1.\, e_1;\, x_2.\, e_2), q) \mapsto ([v/x_2]e_2, q)}$$

Fig. 4. The call-by-value small-step cost semantics of our language.

also define a "pure" semantics $e \longrightarrow e' \triangleq \exists q, q'.\, (e, q) \mapsto (e', q')$ and denote the transitive reflexive closure of that as $e \longrightarrow^* e'$.

One property about the semantics that will come in handy is as follows:

Lemma 4.1 (Cost semantics offset). *If $(e, q) \mapsto (e', q')$, then for all $r \geq 0$, $(e, q + r) \mapsto (e', q' + r)$.*

Proof. By induction on the derivation. □

This property extends to $\mapsto^*$.

4.2 Type judgements

Type judgements in our system are as follows:

$$\Gamma;\ \overset{\text{input annotation on }\Gamma}{\boxed{P}} \vdash_{\underset{\text{cost model}}{\boxed{c}}} e : \tau;\ \overset{\text{remainder annotation on }\Gamma, \circ : \tau}{\boxed{Q}}$$

This can be informally read as "in the context Γ with P resources, under cost model c, e has type τ with Q resources left over." We now go into detail about each of these constructs that we have yet to explain: context annotations, remainder contexts, and cost models.

4.2.1 Context annotations. Define $\mathcal{I}(\Gamma) \triangleq \bigtimes_{x:\tau\in\Gamma} \mathcal{I}(\tau)$; an element of this is written $(x \mapsto i_x)_{x\in\Gamma}$. Define the projection operator $\pi_j(P) \triangleq Q$ where $q_i = p_{i\cup j}$ where $i \in \mathcal{I}(\Gamma)$. Define ${}^x\curlyvee^y_y(P) \triangleq Q$ where P annotates a context $\Gamma, x : \tau, y : \tau$ and Q annotates a context $\Gamma, y : \tau$ such that the potential of y in Q is shared across x and y in P. When $I \subseteq \mathcal{I}(\Gamma)$ is a set of indices for a context Γ, it induces a resource annotation such that $I(i)$ is 1 if $i \in I$ and 0 otherwise.

We often treat isomorphic annotation semimodules as equal, such as $\mathcal{A}(\Gamma_1, \Gamma_2) \cong \mathcal{A}(\Gamma_1) \otimes \mathcal{A}(\Gamma_2)$ and $\mathcal{A}(x : \tau) \cong \mathcal{A}(\tau)$; we use this only where it *improves* clarity. Various other annotation operations we employ (all of which are linear maps) include:

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

4.2.2 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 [Cervesato et al. 2000; Hodas and Miller 1994] but introduced in the programmatic AARA setting by Kahn and Hoffmann [2021]. 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, but not all, of the benefits of remainder contexts noted by Kahn and Hoffmann [2021] extend to our setting. Key conserved advantages include functions' ability to return potential back to their arguments after being called and the elimination of explicit sharing expressions or structural rule applications. However, consideration of sharing cannot be eliminated completely in our multivariate analysis because of mixed potential terms.

4.2.3 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).

4.3 Types

We give an overview of most of the types in our language in §3.1, but we 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 2.3 and 3, 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 any power there over inductive types; 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 $\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle$, where $\Theta, \Theta_{\mathrm{cf}} \subseteq \mathcal{A}(\tau_1) \times \mathcal{A}(\tau_1 \times \tau_2)$. 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 §4.2.2. Θ_{cf} serves a similar purpose for the cost-free model discussed in §4.2.3–$(P, Q) \in \Theta_{\mathrm{cf}}$ 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 any resources gained or consumed during execution.

4.4 Typing rules

The typing rules for our language are available in Figures 5 and 6. 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 a representative subset of the *interesting* typing rules given in Figure 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 [Hoffmann et al. 2011]. 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. Several rules involve tensor constructions that look complex but are really just unfortunate victims of cumbersome bookkeeping. This rule is one of them, which we hope to use as an illustrative example to untangle the notations. The condition for T:Pair may be graphically expressed as the following string diagram:

Resource typing

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

$$\textsc{T:TickPos}\quad \frac{q \geq 0}{\Gamma; q \cdot C(\Gamma) \vdash_{\mathrm{cp}} \mathsf{tick}\{q\} : \mathbf{1}; 0} \qquad \textsc{T:TickNeg}\quad \frac{q \geq 0}{\Gamma; 0 \vdash_{\mathrm{cp}} \mathsf{tick}\{-q\} : \mathbf{1}; q \cdot C(\Gamma, \circ : \mathbf{1})} \qquad \textsc{T:TickFree}\quad \frac{}{\Gamma; 0 \vdash_{\mathrm{cf}} \mathsf{tick}\{q\} : \mathbf{1}; 0}$$

$$\textsc{T:Var}\quad \frac{{}^{\circ}\curlyvee^{x}_{x}(Q) = P}{\Gamma, x : \tau; P \vdash_c x : \tau; Q} \qquad \textsc{T:Unit}\quad \frac{}{\Gamma; 0 \vdash_c \mathsf{tt} : \mathbf{1}; 0} \qquad \textsc{T:Let}\quad \frac{\begin{array}{c}\Gamma; P \vdash_c e_1 : \tau'; R \\ (\mathcal{A}(\Gamma) \otimes \mathcal{A}(\tau'))(R) = (\mathcal{A}(\Gamma) \otimes \mathcal{A}(\tau'))(S) \\ \Gamma, x : \tau'; S \vdash_c e_2 : \tau; Q \otimes C(x : \tau')\end{array}}{\Gamma; P \vdash_c \mathsf{let}(e_1;\, x.\, e_2) : \tau; Q}$$

$$\textsc{T:InL}\quad \frac{P = (\mathcal{A}(\Gamma) \otimes (\curlyvee \circ (\mathcal{A}(\tau_l) \otimes \mathsf{inl}^{-1})))(Q)}{\Gamma, x : \tau_l; P \vdash_c \mathsf{inl}\, x : \tau_l + \tau_r; Q} \qquad \textsc{T:InR}\quad \frac{P = (\mathcal{A}(\Gamma) \otimes (\curlyvee \circ (\mathcal{A}(\tau_r) \otimes \mathsf{inr}^{-1})))(Q)}{\Gamma, x : \tau_r; P \vdash_c \mathsf{inr}\, x : \tau_l + \tau_r; Q}$$

$$\textsc{T:CaseSum}\quad \frac{\begin{array}{c} x' \text{ fresh} \qquad {}^{x'}\curlyvee^{x}_{x}(P') = P \qquad {}^{x'}\curlyvee^{x}_{x}(Q') = Q \\ \forall n \in \{\mathsf{l}, \mathsf{r}\}.\, \Big(\qquad (\mathcal{A}(\Gamma) \otimes \mathsf{in}_n^{-1} \otimes \mathsf{in}_n^{-1})(P') = (\mathcal{A}(\Gamma) \otimes \mathsf{in}_n^{-1} \otimes \mathcal{A}(\tau_n))(R_n) \\ \Gamma, x : \tau_1 + \tau_2, x_n : \tau_n; R_n \vdash_c e_n : \tau; S_n \\ (\mathcal{A}(\Gamma) \otimes \mathsf{in}_n^{-1} \otimes \mathcal{A}(\tau_n) \otimes \mathcal{A}(\tau))(S_n) = (\mathcal{A}(\Gamma) \otimes \mathsf{in}_n^{-1} \otimes \mathsf{in}_n^{-1} \otimes \mathcal{A}(\tau))(R_n)\Big) \end{array}}{\Gamma, x : \tau_l + \tau_r; P \vdash_c \mathsf{cases}(x;\, x_l.\, e_l;\, x_r.\, e_r) : \tau; Q}$$

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

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

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

$$\textsc{T:App}\quad \frac{\forall i \in C(\Gamma).\, (\pi_{\Gamma \mapsto i}(P), \pi_{\Gamma \mapsto i}(Q)) \in \Theta \qquad \forall i \notin C(\Gamma).\, (\pi_{\Gamma \mapsto i}(P), \pi_{\Gamma \mapsto i}(Q)) \in \Theta_{\mathrm{cf}}}{\Gamma, f : \langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle, x : \tau_1; P \vdash_c \mathsf{app}(f;\, x) : \tau_2; Q}$$

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

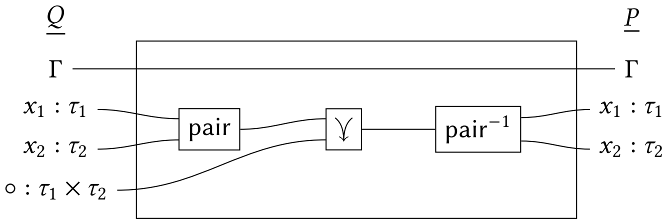
Fig. 5. Resource typing inference rules. See §4.4 for explanations of select rules.

$$\textsc{T:Relax}\quad \frac{\Gamma'; R \vdash_c e : \tau'; S \qquad R \leq P \qquad Q \leq S}{\Gamma; P \vdash_c e : \tau; Q}$$

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

$$\textsc{T:Augment}\quad \frac{\Gamma; R \vdash_c e : \tau; S \qquad P = R + T \qquad Q = S + T \otimes C(\tau)}{\Gamma; P \vdash_c e : \tau; Q}$$

Fig. 6. Resource typing inference rules, continued.



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, but they are written so simply! Of course, the hard work is done in the definition of the additive shift operator in §3.2.

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.

5 SOUNDNESS

The soundness result we desire, as originally proposed by Milner [1978] and more recently advocated for by Ahmed et al. [2010] and Dreyer [2018], is *semantic type soundness*, which in Dreyer's formulation informally consists of defining a semantic typing relation $\Gamma \vDash e : \tau$, then showing

- *adequacy*: if $\cdot \vDash e : \tau$, then e does not go wrong and (if terminating) results in a value corresponding to τ, and
- *compatibility*: all uses of "$\vdash$" in typing rules can be replaced with "$\vDash$".

We proceed exactly corresponding to this plan. One unusual complication we choose for simplicity's sake is to work in an ambient logic with a "later" modality $\triangleright$ to avoid explicit step-indexing. Our adequacy result thus depends also on said logic's adequacy result, but these concerns are addressed by our Coq mechanization, which we discuss below.

5.1 Semantic typing relation

To state our notion of semantic well-typedness, in Figure 7 we first define a weakest precondition unary relation $\mathcal{WP}$ on expressions and a type-indexed unary logical relation $\mathcal{V}$ on values. Both are indexed by a cost model c, as explained in §4.2.3.

Our weakest precondition relation $\mathcal{WP}$ is parameterized by a postcondition Ψ and takes as arguments an expression e and initial resources q. Intuitively, $\mathcal{WP}_c[\Psi](e, q)$ holds if e "can't

Weakest Precondition Relation $\boxed{\mathcal{WP}_c[\Psi](e,q)}$

$$\begin{aligned}\mathcal{WP}_{\mathrm{cp}}[\Psi] \triangleq \mu \mathcal{W}.\ \lambda(e,q).\ &(e \text{ val} \wedge \Psi(e,q)) \vee \\ &((\exists e',q'.\ (e,q) \mapsto (e',q')) \wedge \forall e',q'.\ (e,q) \mapsto (e',q') \Rightarrow \triangleright\mathcal{W}(e',q'))\end{aligned}$$

$$\begin{aligned}\mathcal{WP}_{\mathrm{cf}}[\Psi] \triangleq \mu \mathcal{W}.\ \lambda(e,q).\ &(e \text{ val} \wedge \Psi(e,q)) \vee \\ &((\exists e'.\ e \longrightarrow e') \wedge \forall e'.\ e \longrightarrow e' \Rightarrow \triangleright\mathcal{W}(e',q))\end{aligned}$$

Value Logical Relation $\boxed{\mathcal{V}_c[\![\tau]\!](v)}$

$$\begin{aligned}
\mathcal{V}'^R_c[\![\mathbf{1}]\!](v) &\triangleq v = \mathsf{tt} \\
\mathcal{V}'^R_c[\![\tau_1 \times \tau_2]\!](v) &\triangleq \exists v_1, v_2.\ v = \mathsf{pair}(v_1; v_2) \wedge \mathcal{V}'^R_c[\![\tau_1]\!](v_1) \wedge \mathcal{V}'^R_c[\![\tau_2]\!](v_2) \\
\mathcal{V}'^R_c[\![\tau_1 + \tau_2]\!](v) &\triangleq (\exists v_1.\ v = \mathsf{inl}\, v_1 \wedge \mathcal{V}'^R_c[\![\tau_1]\!](v_1)) \vee (\exists v_2.\ v = \mathsf{inr}\, v_2 \wedge \mathcal{V}'^R_c[\![\tau_2]\!](v_2)) \\
\mathcal{V}'^R_c[\![\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle]\!](v) &\triangleq \exists f, x, e.\ v = \mathsf{fun}(f, x.\, e) \wedge \\
&\qquad \triangleright (\forall (P,Q) \in \Theta.\ \forall q \geq 0, v'.\ v' \in \mathcal{V}(\tau_1) \wedge \mathcal{V}'^R_c[\![\tau_1]\!](v') \Rightarrow \\
&\qquad\quad \mathcal{WP}_c[\lambda(v'', q').\ v'' \in \mathcal{V}(\tau_2) \wedge \mathcal{V}'^R_c[\![\tau_2]\!](v'') \wedge \\
&\qquad\qquad q' \geq q + \Phi(\mathsf{pair}(v'; v'') : \langle \tau_1 \times \tau_2; Q \rangle)] \\
&\qquad\quad ([v'/x, v/f]e, q + \Phi(v' : \langle \tau_1; P \rangle))) \wedge \\
&\qquad \triangleright (\forall (P,Q) \in \Theta_{\mathrm{cf}}.\ \forall q \geq 0, v'.\ v' \in \mathcal{V}(\tau_1) \wedge \mathcal{V}'^R_{\mathrm{cf}}[\![\tau_1]\!](v') \Rightarrow \\
&\qquad\quad \mathcal{WP}_{\mathrm{cf}}[\lambda(v'', q').\ v'' \in \mathcal{V}(\tau_2) \wedge \mathcal{V}'^R_{\mathrm{cf}}[\![\tau_2]\!](v'') \wedge \\
&\qquad\qquad q' \geq q + \Phi(\mathsf{pair}(v'; v'') : \langle \tau_1 \times \tau_2; Q \rangle)] \\
&\qquad\quad ([v'/x, v/f]e, q + \Phi(v' : \langle \tau_1; P \rangle))) \\
\mathcal{V}'^R_c[\![\mu\alpha.\,\tau]\!](v) &\triangleq \exists v'.\ v = \mathsf{fold}\, v' \wedge \triangleright R_c[\![[\mu\alpha.\,\tau/\alpha]\tau]\!](v')
\end{aligned}$$

$$\mathcal{V}_c[\![\tau]\!](v) \triangleq v \in \mathcal{V}(\tau) \wedge (\mu R. \mathcal{V}'^R)_c[\![\tau]\!](v)$$

Fig. 7. Weakest precondition relation and logical relation on values.

go wrong" starting with q resources and $\Psi(v,q')$ holds on any resulting value v and remaining resources q'. The definition uses the ambient logic's fixpoint operator to perform step-indexing, which as we will see is critical for the definition of $\mathcal{V}$ on recursive types.

The core theorem for relating cost-paid and cost-free weakest preconditions, as needed by the application rule, is that we can combine both into one cost-paid weakest precondition; formally:

Lemma 5.1 (Weakest precondition glomming). *If $\mathcal{WP}_{\mathrm{cp}}[\Psi_p](e,q_p)$ and $\mathcal{WP}_{\mathrm{cf}}[\Psi_f](e,q_f)$ both hold, then so does $\mathcal{WP}_{\mathrm{cp}}[\lambda(v,q').\ \exists q'_p, q'_f.\, q' = q'_p + q'_f \wedge \Psi_p(v,q'_p) \wedge \Psi_f(v,q'_f)](e, q_p + q_f)$.*

Proof. By induction over steps and use of the definition of $\longrightarrow$. □

Because our language has arbitrary recursive types, we cannot define $\mathcal{V}$ purely by structural induction on the type index, so we proceed by two layers of induction, as in Ahmed et al. [2009]. The definition of $\mathcal{V}$ first uses the logic's fixpoint construction μ to obtain the relation R, then passes it off to $\mathcal{V}'^R$ to induct over types, which leads to a well-defined result as long as $\mathcal{V}'^R$ is contractive in R (which it is). This construction is involved, but ultimately leads to the nice, intuitive laws being able to be proven afterward–e.g., we have $\mathcal{V}_c[\![\tau_1 \times \tau_2]\!](v)$ iff $\exists v_1, v_2.\ v = \mathsf{pair}(v_1; v_2) \wedge \mathcal{V}_c[\![\tau_1]\!](v) \wedge \mathcal{V}_c[\![\tau_2]\!](v)$.

The value relation is completely standard for all types but function types, so we will only detail arrows. As explained in §4.3, an arrow type has form $\langle \tau_1 \to \tau_2, \Theta, \Theta_{\mathrm{cf}} \rangle$ where $\Theta, \Theta_{\mathrm{cf}} \subseteq \mathcal{A}(\tau_1) \times \mathcal{A}(\tau_1 \times \tau_2)$, which are sets of specifications of the potential needed and returned by the function. Our interpretation thus intuitively states that for each annotation specification $(P, Q) \in \Theta$ (respectively Θ_{cf}), for all resource states q and arguments v', assuming that v' is semantically well typed at τ_1, running the function with q resources augmented with the added potential that P mandates of v, the result (if terminating) is semantically well typed at τ_2 and the remaining resources are at least q augmented with that which Q guarantees. Here the quantification over all resources q acts as a Kripke-style quantification over future states. Strictly speaking, it is not needed because of Lemma 4.1, but it highlights the Kripke-like requirement.

With those relations under our belt, we define the semantic resource typing judgement:

Definition 5.2 (Semantic resource typing).

$$\begin{aligned}\Gamma; P \vDash_c e : \tau; Q \triangleq \forall \gamma, q \geq 0.\ & \mathcal{V}_c[\![\Gamma]\!](\gamma) \Rightarrow \\ & \mathcal{WP}_c[\lambda(v, q').\ \mathcal{V}_c[\![\tau]\!](v) \wedge q' \geq q + \Phi(\gamma[\circ \mapsto v] : \langle \Gamma, \circ : \tau; Q \rangle)] \\ & (e, q + \Phi(\gamma : \langle \Gamma; P \rangle))\end{aligned}$$

That is, under cost model c and in annotated context $\langle \Gamma; P \rangle$, expression e is semantically well-typed at τ with remainder annotation Q iff for all semantically well-typed closing substitution contexts γ, e never crashes and results in a semantically well-typed remainder value context with potential at least as much as according to Q if it terminates, assuming the run is started with at least as many resources as P says γ should.

Now we can state and prove our adequacy result:

Theorem 5.3 (Adequacy). *Assume $\cdot; p \vDash_{\mathrm{cp}} e : \tau; Q$. 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. By the soundness result for the ambient logic we have implicitly been developing in, we know that if $\triangleright^n P$ is derivable (where P is a logical statement in the outer logic), then P holds. We can instantiate this where n is the number of steps taken in the execution, at which point we can use the definition of semantic resource typing and weakest precondition to show the desired result. □

5.2 Compatibility

At this point, we wish to prove compatibility of our syntactic typing rules with this semantic notion, that is:

Theorem 5.4 (Compatibility). *If all instances of $\vdash$ in Figure 5 are replaced with $\vDash$, the inference rules are logical entailments.*

Proof sketch. Given the way we've set things up, this is actually quite trivial to prove for almost all rules, as the statement boils down to equality of potential evaluations that are easily discharged with theorems from §3 and §4.2.1. For those rules where this is not true, we give sketches of proofs:

- T:Let. Inducts over the execution derivation for e_1 and uses the evaluation context stepping rule for let.

- T:Unfold. The equality of potentials follows immediately; the only typing wrinkle comes from the "later" in the μ type, but this is eliminated with the unfold (fold v) to v step.
- T:App. First picks the one cost-paid annotation in Θ to be used, then inducts over the finite set of nonconstant indices in P with nonzero coefficients to create one big weakest precondition for the function application, using Lemma 5.1.
- T:Relax. Follows from annotation evaluation respecting the ordering on annotations.

□

As a corollary, we also get the traditional fundamental theorem of logical relations.

Corollary 5.5 (Fundamental theorem). *If* $\Gamma; P \vdash_c e : \tau; Q$, *then* $\Gamma; P \vDash_c e : \tau; Q$.

5.3 Example of manual proof of well-typedness

As an example of how the semantic typing relation can be applied beyond our proof of soundness, using the same metric of cons cell creations, we offer the type of a bubblesort function that states it creates at most $2n(n+1)$ cons cells and a high-level proof that the function satisfies that type semantically. (In practice, to really guarantee this, we would mechanically prove it, but here we present a paper proof to communicate the idea.) Recall again our earlier definition of bubblesort from §2.4.

AARA can easily infer that bubble has (as one) type $\langle \mathsf{list(int)} \to \mathsf{list(int)}, \{(2 \cdot [\star], 0)\}, \emptyset\rangle$. By using the FTLR, we know it has this type semantically, too, which in turn means that for all $q \geq 0$ and lists of ints l, the result of running $[l/\mathsf{l}]$bubble starting with $q + 2|l|$ resources will also be a list of ints, with at least q resources remaining.

To prove a resource bound on bubblesort, knowing the resource behavior of bubble is not enough, which is why AARA cannot derive a bound for bubblesort; we must also prove a specification of bubble. Define $I(l_1, l_2)$ to hold iff l_1 is sorted and all elements of l_2 are no smaller than any element of l_1. We must then prove this statement: for any lists l_1 and l_2 such that $I(l_1, l_2)$ holds, bubble $(l_1 +\!\!+\, l_2)$ results in a list $l_1' +\!\!+\, l_2'$, where $|l_1'| = |l_1| + \min(1, |l_2|)$, $|l_2'| = |l_2| - \min(1, |l_2|)$, and $I(l_1', l_2')$ holds; additionally, if $l_2 = []$, then the list returned is just $l_1 +\!\!+\, l_2$. This can be accomplished by a variety of verification methods orthogonal to our point here, but note we can completely ignore resource usage while proving this!

Now we want to show that bubblesort semantically has the type $\langle \mathsf{list(int)} \to \mathsf{list(int)}, (2 \cdot [\star;\star] + 4 \cdot [\star], 0), \emptyset\rangle$. This boils down to showing that for all $q \geq 0$ and lists of ints l, the result of running $[l/\mathsf{l}]$bubblesort starting with $q + 2n(n+1)$ resources will also be a list of ints, with at least q resources remaining. For this, it suffices to show that for any lists l_1 and l_2 such that $I(l_1, l_2)$ holds, running $[(l_1 +\!\!+\, l_2)/\mathsf{l}]$bubblesort with $q + 2(|l_1| + |l_2|)(|l_2| + 1)$ resources results in a list of ints and at least q resources remaining. This can be shown by induction on l_2: by the resource bound on bubble, we know that after running it we have $q + 2|l_2|(|l_1| + |l_2|)$ resources left; by our specification of bubble, if $l_2 = []$, we are done, but otherwise we can use our induction hypothesis.

At this point, we can now use this semantic type for bubblesort together with our typing rules; for example, we could apply it to an argument and use T:App to derive a type judgement.

5.4 Mechanization

We have mechanized this development in the Coq proof assistant using the Iris logic for its handling of step-indexing [Jung et al. 2018; Team 2022].[4] To the best of the authors' knowledge, this is the first

[4] Not everything is quite proven yet, but, for one, we believe the core, sensitive parts are, and for another, we plan to have proven everything by the rebuttal period.

mechanization of soundness for a multivariate AARA type system, a significant contribution in and of itself. The bulk of the proof work was not for the type system per se, but in justifying the theorems underlying the annotation manipulations. On paper, many of these are look straightforward, but make use of myriad implicit isomorphisms and assumptions of finite support, which makes their mechanized proofs quite tedious. For example, manipulating infinite sums of annotation evaluations for the application rule comes for free on paper, but in Coq requires proving that the summand functions have finite support, which makes equational reasoning tricky. As a rough estimate of complexity, the development involves 11k lines of Coq code and took about five person-months, when an attempt is made to disentangle "formal proving time" from "research time."

6 RELATED WORK

Our work builds upon the literature of the Automatic Amortized Resource Analysis (AARA) type system. AARA was first introduced in [Hofmann and Jost 2003], using potential method[Tarjan 1985] 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 [Hoffmann and Hofmann 2010], exponentials [Kahn and Hoffmann 2020], logarithms [Hofmann et al. 2021], and maxima [Campbell 2009; Kahn and Hoffmann 2021]. However, each of these works bakes their size parameters and resource functions into the inductive data types of trees and lists (or labeled trees [Hoffmann et al. 2017]). The one exception is the Schopenhauer language introduced by Jost et al. [2010], which included support for deriving bounds on programs using nested recursive types, but only in the very restricted 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 [Hoffmann et al. 2011, 2017], 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 [Carbonneaux et al. 2015], object-oriented [Hofmann and Jost 2006; Hofmann and Rodriguez 2013], probabilistic [Ngo et al. 2018; Wang et al. 2020], and parallel computation [Hoffmann and Shao 2015], digital contract protocols [Das et al. 2021], and lower bound costs [Dehesa-Azuara et al. 2017].

Aside from AARA, other type-based systems have also been used to analyze the resource usage of programs. These include linear dependent types [Dal Lago and Gaboardi 2011; Dal Lago and Petit 2013], refinement types [Radiček et al. 2017; Wang et al. 2017], modal types [Rajani et al. 2021], size types [Serrano et al. 2014; Vasconcelos 2008], annotation-based systems [Crary and Weirich 2000; Danielsson 2008], 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 like in [Niu et al. 2022], which require significant user work to prove cost bounds. Rajani et al. [2021] actually provide a semantic soundness proof of linear AARA with lists via their language, but their technique would not scale to polynomial or multivariate annotations.

The term-rewriting space provides some work that is comparable with ours. Specifically, the system from [Hofmann and Moser 2015] generalizes multivariate potential functions over aribtrary types using tree automata. We speculate that this system is general enough to contain the resource functions generated by our approach. However, their work leaves the open questions of how to pick the appropriate automata, and how to solve the constraints they induce. There has been much work [Bauer 2019] to even solve simpler cases than the multivariate case. Our work could potential be seen as a step toward this automation. Other resource analysis work using term-rewriting include [Avanzini et al. 2015; Avanzini and Moser 2016; Hirokawa and Moser 2014; Naaf et al. 2017].

Recurrence relations are another common approach to resource analysis, especially in a functional setting [Danner et al. 2015; Kavvos et al. 2019; Kincaid et al. 2017a]. Some recent work uses potential-based reasoning for amortization [Cutler et al. 2020]. 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 [Gulwani 2009; Gulwani et al. 2009] and logic [Navas et al. 2007] program analyses also can reason about general notions and functions 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 a common tool for resource analysis, it does not focus on the sorts of intrinsic feature of data structures that our work does, making it harder to compare.

Other approaches to resource analysis include abstract interpretation [Albert et al. 2007, 2015; Lopez-Garcia et al. 2018], loop analyses [Blanc et al. 2010; Kincaid et al. 2017b], relational cost analysis [Çiçek et al. 2017; Qu et al. 2021], ranking functions [Chatterjee et al. 2019], and program logics [Atkey 2010; Guéneau et al. 2018; Mével et al. 2019]. The field varies broadly and mixes and matches many approaches. In particular, the cited program logics have been combined with potential-based reasoning like that used in AARA.

7 CONCLUSION AND FUTURE WORK

This work's contributions include the extension of multivariate resource polynomials through nested data types, a semantic characterization of the soundness of AARA, a Coq formalization of the core components of the type system, 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. Semantic soundness enables the integration of non-structurally-derived resource bounds, like for bubblesort. The Coq formalization ensures that these results are trustworthy.

The extended resource polynomials of this work comprise a major step forward in automatable resource analysis through the structural combinatorics of (possibly-nested) data types. However, this is not the last step to be made. Future work will include the implementation of these resource functions in a fully automated resource analysis typechecker, like RaML [Hoffmann et al. 2017]. We expect no difficulties with this because the constraints we generate lie completely within the space that previous AARA systems generated. Further, we would like to extend our methods for generating resouce polynomials to also cover resource exponentials [Kahn and Hoffmann 2020], which currently are not supported at the multivariate level at all.

The semantic characterization of soundness is also a new development in polynomial AARA. Previously, AARA's soundness proofs were characterized syntactically, which precludes sound integration of externally-known bounds that are not expressible through the type system. With the new semantic characterization of soundness, there is now the possibility of including such externally-known bounds. This would allow more sophisticated resource analysis techniques to be combined with AARA to deal with particularly complex programs. Future work can explore this direction, including implementing such integration and developing an AARA type system with purely semantically characterized types.

REFERENCES

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

A. Ahmed, D. Dreyer, and L. Birkedal. 2009. Logical step-indexed logical relations. In *Logic in computer science, symposium on* (Los Alamitos, CA, USA, 2009-08). IEEE Computer Society, 71–80. https://doi.org/10.1109/LICS.2009.34 ISSN: 1043-6871.

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2007. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In *International Symposium on Formal Methods for Components and Objects*. Springer, 113–132.

Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative resource analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 85–100.

Elvira Albert, Samir Genaim, and Abu Naser Masud. 2013. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Trans. Comput. Log.* 14, 3 (2013), 22:1–22:35. https://doi.org/10.1145/2499937.2499943

Robert Atkey. 2010. Amortised resource analysis with separation logic. In *European Symposium on Programming*. Springer, 85–103.

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the complexity of functional programs: higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 152–164.

Martin Avanzini and Georg Moser. 2016. A combination framework for complexity. *Information and Computation* 248 (2016), 22–55.

Sabine Bauer. 2019. *Decidability of linear tree constraints for resource analysis of object-oriented programs.* Ph. D. Dissertation. lmu.

Régis Blanc, Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: algebraic bound computation for loops. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 103–118.

Brian Campbell. 2009. Amortised memory analysis using the depth of data structures. In *European Symposium on Programming*. Springer, 190–204.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 467–478.

Iliano Cervesato, Joshua S Hodas, and Frank Pfenning. 2000. Efficient resource management for linear logic proof search. *Theoretical Computer Science* 232, 1-2 (2000), 133–163.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 4 (2019), 1–52.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. *ACM SIGPLAN Notices* 52, 1 (2017), 316–329.

Karl Crary and Stephnie Weirich. 2000. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 184–198.

Joseph W Cutler, Daniel R Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 133–142.

Ugo Dal Lago and Barbara Petit. 2013. The geometry of types. *ACM SIGPLAN Notices* 48, 1 (2013), 167–178.

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. *ACM SIGPLAN Notices* 43, 1 (2008), 133–144.

Norman Danner, Daniel R Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 140–151.

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-aware session types for digital contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 1–16.

Mario Dehesa-Azuara, Matthew Fredrikson, Jan Hoffmann, et al. 2017. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 710–728.

Derek Dreyer. 2018. *Milner Award Lecture: The Type Soundness Theorem That You Really Want to Prove (and Now You Can) (POPL 2018 - Research Papers) - POPL 2018*. https://popl18.sigplan.org/details/POPL-2018-papers/14/Milner-Award-Lecture-The-Type-Soundness-Theorem-That-You-Really-Want-to-Prove-and-N

Marcelo P. Fiore and Gordon D. Plotkin. 1994. 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, 92–102. https://doi.org/10.1109/LICS.1994.316083

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming*. Springer, 533–560.

Sumit Gulwani. 2009. Speed: Symbolic complexity bound analysis. In *International Conference on Computer Aided Verification*. Springer, 51–62.

Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. 2009. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices* 44, 1 (2009), 127–139.

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

Nao Hirokawa and Georg Moser. 2014. Automated complexity analysis based on context-sensitive rewriting. In *Rewriting and Typed Lambda Calculi*. Springer, 257–271.

Joshua S Hodas and Dale Miller. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and computation* 110, 2 (1994), 327–365.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 359–373.

Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*. Springer, 287–306.

Jan Hoffmann and Zhong Shao. 2015. Automatic static cost analysis for parallel programs. In *European Symposium on Programming Languages and Systems*. Springer, 132–157.

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices* 38, 1 (2003), 185–197.

Martin Hofmann and Steffen Jost. 2006. Type-based amortised heap-space analysis. In *European Symposium on Programming*. Springer, 22–37.

Martin Hofmann, Lorenz Leutgeb, David Obwaller, Georg Moser, and Florian Zuleger. 2021. Type-based analysis of logarithmic amortised complexity. *Mathematical Structures in Computer Science* (2021), 1–33.

Martin Hofmann and Georg Moser. 2015. 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.

Martin Hofmann and Dulma Rodriguez. 2013. Automatic type inference for amortised heap-space analysis. In *European Symposium on Programming*. Springer, 593–613.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. 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*. 223–236.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. 2 (2017), 66:1–66:34. Issue POPL. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. 28 (2018). https://doi.org/10.1017/S0956796818000151 Publisher: Cambridge University Press.

David M Kahn and Jan Hoffmann. 2020. Exponential automatic amortized resource analysis. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, Cham, 359–380.

David M Kahn and Jan Hoffmann. 2021. Automatic amortized resource analysis with the quantum physicist's method. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

GA Kavvos, Edward Morehouse, Daniel R Licata, and Norman Danner. 2019. Recurrence extraction for functional programs through call-by-push-value. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017a. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017b. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid resource types. 4 (2020), 106:1–106:29. Issue ICFP. https://doi.org/10.1145/3408988

Pedro Lopez-Garcia, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V Hermenegildo. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming* 18, 2 (2018), 167–223.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming*. Springer, 3–29.

Robin Milner. 1978. A theory of type polymorphism in programming. 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl. 2017. Complexity analysis for term rewriting by integer transition systems. In *International Symposium on Frontiers of Combining Systems*. Springer, 132–150.

Jorge Navas, Edison Mera, Pedro López-García, and Manuel V Hermenegildo. 2007. User-definable resource bounds analysis for logic programs. In *International Conference on Logic Programming*. Springer, 348–363.

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. *ACM SIGPLAN Notices* 53, 4 (2018), 496–512.

Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

Weihao Qu, Marco Gaboardi, and Deepak Garg. 2021. Relational cost analysis in a functional-imperative setting. *Journal of Functional Programming* 31 (2021).

Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2017. Monadic refinements for relational cost analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–32.

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

Alejandro Serrano, Pedro López-García, and Manuel V Hermenegildo. 2014. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming* 14, 4-5 (2014), 739–754.

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.5846982 Language: eng.

Pedro B Vasconcelos. 2008. *Space cost analysis using sized types.* Ph. D. Dissertation. University of St Andrews.

Di Wang, David M Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–31.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

A NOTATION GLOSSARY

As this paper introduces a lot of notation, we thought it might be useful to have a handy reference for a quick refresher for that outside a typical type system paper. The order roughly matches that in which they are introduced in the paper. Without further ado:

Notation	Loc.	Overview
$\mathcal{V}(\tau)$	§3.1	The set of values that syntactically correspond to type τ
$\mathcal{I}(\tau)$	§3.1	The indices for type τ, the set of names for the base polynomials of type τ
$C(\tau)$	§3.1	The constant index set, the set of indices that sum to 1 when evaluated on a value of type τ
$\mathcal{M}\{\alpha.\tau\}(i)$	§3.1	The set of indices that correspond to placing the index i at each occurrence of α in τ
$\phi_i(v : \tau)$	§3.1	The evaluation of the base polynomial corresponding to index i at the value v
$\mathcal{R}(\tau)$	§3.1	The set of resource polynomials for τ, formed by taking all linear combinations (over $\mathbb{Q}_{\geq 0}$ of base polynomials for τ. More abstractly, these are all the possible potential functions for τ within our system
$\mathcal{A}(\tau)$	§3.2	The set of annotations for τ, the free semimodule over $\mathbb{Q}_{\geq 0}$ generated by $\mathcal{I}(\tau)$. $\mathcal{R}(\tau)$ is the same as $\mathcal{A}(\tau)$ with the indices that induce equivalent base polynomials quotiented out.
$\Phi(v : \langle \tau; P \rangle)$	§3.2	The evaluation of the annotation P on the value v, $\Phi(v : \langle \tau; P \rangle) = \sum_{i \in \mathcal{I}(\tau)} p_i \cdot \phi_i(v : \tau)$.
$P \curlyvee Q$	§3.2	The sharing operator, which produces a single annotation that corresponds to the multiplication of two annotations, i.e., we have $\Phi(v : \langle \tau; P \curlyvee Q \rangle) = \Phi(v : \langle \tau; P \rangle) \cdot \Phi(v : \langle \tau; Q \rangle)$
$\triangleleft(P)$	§3.2	The shifting operator, which produces an annotation for an unfolded type with exactly the same potential as the folded type, i.e., we have $\Phi(\text{fold}\, v : \langle \mu\alpha.\, \tau; P \rangle) = \Phi(v : \langle [\mu\alpha.\, \tau/\alpha]\tau; \triangleleft(P) \rangle)$.
$^{x}\curlyvee^{y}_{z}(P)$	§4.2.1	Sharing extended to context annotations, $^{x}\curlyvee^{y}_{z} = (\mathcal{A}(\Gamma) \otimes \curlyvee) : \mathcal{A}(\Gamma, x : \tau, y : \tau) \to \mathcal{A}(\Gamma, z : \tau)$. The intuition is that the potential in x and y is being combined into just z
$\triangleleft_x(P)$	§4.2.1	Shifting extended to context annotations, $\triangleleft_x = (\mathcal{A}(\Gamma) \otimes \triangleleft) : \mathcal{A}(\Gamma, x : \mu\alpha.\, \tau) \to \mathcal{A}(\Gamma, x : [\mu\alpha.\, \tau/\alpha]\tau)$
$\circ$	§4.2.2	A pseudovariable representing the result in remainder contexts
$\Gamma; P \vdash_c e : \tau; Q$	§4.2	The type judgement for our language, where P is an annotation on the context Γ, c is the cost model, e is the expression being typed against τ, and Q is an annotation on the remainder context of $\Gamma, \circ : \tau$
$\mathcal{WP}_c[\Psi](e, q)$	§5.1	The weakest precondition relation, which says that under cost model c, running e with q resources doesn't get stuck, and if it terminates with value v and remaining resources q', $\Psi(v, q')$ holds
$\mathcal{V}_c[\![\tau]\!](v)$	§5.1	The logical relation on values, which says that under cost model c, v semantically has type τ
$\Gamma; P \vDash_c e : \tau; Q$	§5.1	The semantic equivalent of $\Gamma; P \vdash_c e : \tau; Q$