Exponential Automatic Amortized Resource Analysis[⋆]

David M. Kahn [✉] and Jan Hoffmann

Carnegie Mellon University, Pittsburgh PA, USA
`davidkah@cs.cmu.edu` `cs.cmu.edu/~davidkah`
`jhoffmann@cmu.edu` `cs.cmu.edu/~janh`

Abstract. Automatic amortized resource analysis (AARA) is a type-based technique for inferring concrete (non-asymptotic) bounds on a program's resource usage. Existing work on AARA has focused on bounds that are polynomial in the sizes of the inputs. This paper presents and extension of AARA to exponential bounds that preserves the benefits of the technique, such as compositionality and efficient type inference based on linear constraint solving. A key idea is the use of the Stirling numbers of the second kind as the basis of potential functions, which play the same role as the binomial coefficients in polynomial AARA. To formalize the similarities with the existing analyses, the paper presents a general methodology for AARA that is instantiated to the polynomial version, the exponential version, and a combined system with potential functions that are formed by products of Stirling numbers and binomial coefficients. The soundness of exponential AARA is proved with respect to an operational cost semantics and the analysis of representative example programs demonstrates the effectiveness of the new analysis.

Keywords: Functional programming · Resource consumption · Quantitative analysis · Amortized analysis · Stirling numbers · Exponential

1 Introduction

"Time is money" is a phrase that also applies to executing software, most directly in domains such as on-demand cloud computing and smart contracts where execution comes with a explicit price tag. In such domains, there is an increasing interest in formally analyzing and certifying the precise resource usage of programs. However, the cost of formally verifying properties by hand is an obstacle to even getting projects off the ground. For this reason, it would be desirable if such resource analyses could be performed mostly automatically, with reduced burden on the programmer.

Techniques and tools for automatic and semi-automatic resource analysis have been extensively studied. The applied methods range from deriving and analyzing recurrence relations [55,1,16,2,12,36,10,37], to abstract interpretation and static analysis [18,7,49,39], to type systems [11,56,53], to proof assistants and program logics [4,9,8,48,19,45,42], to term rewriting [6,5,47]. Many techniques focus on upper bounds on the worst-case bounds, but average-case bounds [15,35,43,54] and lower-bounds have also been studied [3,17,44].

In this paper, we extend automatic amortized resource analysis (AARA) to cover *exponential* worst-case bounds. AARA is an effective type-based technique for deriving concrete (non-asymptotic) worst-case bounds, in particular for functional languages. It has been introduced by Hofmann and Jost [31] to derive *linear bounds* on the heap-space usage of strict first-order functional programs with lists. Subsequently, AARA has been extended to programs with recursive types and general resource metrics [34], higher-order functions [33], lazy evaluation [52], parallel evaluation [29], univariate polynomial bounds [27], multivariate polynomial bounds [23,25], session-typed concurrency [13], and side effects [38,46]. However, none of the aforementioned works explores exponential bounds.

The idea of AARA is to enrich types with numeric annotations that represent coefficients in a potential function in the sense of amortized analysis [51]. Bound inference is reduced to Hindley-Milner type inference extended with linear constraints for the numeric annotations. Advantages of the technique include compositionality, efficient bound inference via off-the-shelf LP solving, and the ability to derive bounds on the high-water mark for non-monotone resources like memory. A powerful innovation leveraged in polynomial AARA is the representation of potential functions as non-negative linear combinations of binomial coefficients. Their combinatorial identities yield simple and local typing rules and support a natural semantic understanding of types and bounds. Moreover, these potential functions are more expressive than non-negative linear-combinations of the standard polynomial basis.

However, polynomial potential is not always enough. Functional languages make it particularly easy to use exponentially many resources just by having two or more recursive calls. The following function $\texttt{subsetSum} : \texttt{int list} \rightarrow \texttt{int} \rightarrow \texttt{bool}$ exemplifies this by naively solving the well-known NP-complete problem subset sum. In the worst case, it performs $3 * 2^{|nums|} - 2$ Boolean and arithmetic operations (where $|x|$ gives the length of the list x).

```
let subsetSum nums target =
    match nums with
    | [] → target = 0
    | hd::tl → subsetSum tl (target-hd) || subsetSum tl target
```

Such a function could appear in a program with polynomial resource usage if applied to arguments of logarithmic size. In this case, polynomial AARA would not be able to derive a bound. Section 6 contains a relevant example.

To handle such functions, we introduce an extension to AARA that allows working with potential functions of the form $f(n) = b^n$. This extension ex-

ploits the combinatorial properties of *Stirling numbers of the second kind* [50] in much the same way that AARA currently exploits those of binomial coefficients. Moreover, we allow both multiplicative and additive mixtures of exponential and polynomial potential functions. The techniques used in this process could easily be applied to other potential functions in the future.

The paper first details a generalized AARA type system fit for reuse between polynomial, exponential, and other potential functions. We then instantiate this system with Stirling numbers of the second kind, yielding the first AARA that can infer exponential resource bounds. Finally, we pick out the characteristics that allow for mixing different families of potential functions and maximizing the space they express, and we instantiate the general system with products of exponential and polynomial potential functions. To focus on the main contribution, we develop the system for a simple first-order language with lists in which resource usage is defined with explicit *tick* expressions. However, we are confident that the results smoothly generalize to more general resource metrics, recursive types, and higher-order functions. As in previous work, we prove the soundness of the analysis with respect to a big-step cost semantics that models the high-water mark of the resource usage.

2   Language and Cost Semantics

Abstract Syntax To begin, we define an abstract binding tree (ABT, see [20]) underlying a simple strict first-order functional language. Expressions are in let-normal form to simplify the AARA typing rules. For code examples, however, we overlay the ABT with corresponding ML-based syntax. For example, $1 :: [\,]$, $[1]$, and $cons(1, nil)$ all represent the same list.

A program *prog* is a collection of functions as defined in the following grammar. The symbols *lit*, *binop*, and *unop* refer to standard literal values, binary operations, and unary operations respectively, of *basic* types (*int*, *bool*, etc.). The symbols f, x, and r refer to function identifiers, variables, and rational numbers, respectively.

$$
\begin{aligned}
prog ::=\ & func\{f\}(x.e)\ prog \mid \epsilon \\
e ::=\ & lit \mid x \mid binop(x_1; x_2) \mid unop(x) \mid app\{f\}(x) \mid let(e_1; x.e_2) \\
& \mid share(x_1; x_2, x_3.e) \mid tick\{r\} \mid pair(x_1; x_2) \mid nil \mid cons(x_1; x_2) \\
& \mid cond(x; e_1; e_2) \mid pairMatch(x_1; x_2, x_3.e) \mid listMatch(x_1; e_1; x_2, x_3.e_2)
\end{aligned}
$$

Expressions include function applications, conditionals, and the usual introduction and elimination forms for pairs and lists. They also include two special expressions: $tick\{r\}$ and *share*. The former, $tick\{r\}$, is used to specify constant resource cost r. We allow r to be negative in the case of resources becoming available instead of being consumed. The latter, $share(x_1; x_2, x_3.e)$, provides two copies of its argument x_1 for use in e. This is useful because the affine features of the AARA type system do not allow naive variable reuse. In practice, *share* can be left implicit by automatically preceding every variable usage by *share*.

To focus on the technical novelties, we keep function identifiers and variables disjoint, that is, the types of variables do not contain arrow types and functions

are first-order. Higher-order functions can be handled as in previous AARA literature [25]. As a further simplification, we only let functions take one argument; multiple arguments can be simulated with nested pairs. Finally, the language here only supports the inductive types of lists; future work could extend this to more general types as in other AARA literature [38,25,30,28].

Operational Cost Semantics To define resource usage, AARA literature uses the operational big-step judgment $V \vdash e \Downarrow v \mid (q, q')$ (see e.g. [22]) defined in Figure 1. This judgment means that, under the environment V, the expression e evaluates to the value v under some resource constraints given by the pair q, q'. The environment V maps variables to values. The resource constraints are that q is the high-water mark of resource usage, and $q - q'$ is the net amount of resources consumed during evaluation. In other words, if one started with exactly as many resources needed to evaluate e, that amount would be q, and the amount of leftover resources after evaluation would be q'. It is essential to track both of these values to model resources that might be returned after use, like space. Space usage usually has a positive high-water mark but no net resource consumption, as space could be reused.

The above big-step judgment only formalizes terminating evaluations. To deal with divergence, the additional judgment $V \vdash e \Downarrow \circ \mid q$ has been introduced [26]. This merely drops the parts of the previous judgment relevant to post-termination, focusing on partial evaluation. It means that some partial evaluation of e uses a high-water mark of q resources. Should it exist, the largest q such that $V \vdash e \Downarrow \circ \mid q$ holds would be the high-water mark of resource usage across any partial evaluation of e. The formal definition can be found in Figure 2.

3   Automatic Amortized Resource Analysis

Here we lay out a generalized version of the AARA system with the potential functions abstracted. Existing AARA literature is specialized to polynomial functions (see e.g. [27]). This existing polynomial system may be obtained as an instantiation, as may the exponential system that we introduce in Section 4.

AARA uses the *potential* (or physicist's) method to account for resource use, as is commonly used in amortized analyses. The potential method uses the physical analogy of converting between potential and actual energy that can be used to perform work. Whereas a physicist might find potential in the chemical bonds of a fuel, however, AARA places it in the constructors of lists.

To prime intuition with an example, consider paying a resource for each :: operation performed in the following code. It performs *snoc*, which is like *cons* but adds onto the back of the list rather than the front.

```
let snoc x xs =
    match xs with
    | [] → tick 1; x::[]                  (* pay 1 resource here *)
    | hd::tl → tick 1; hd::(snoc x tl)    (* pay 1 resource here *)
```

The resource consumption of *snoc x xs* as defined by the *tick* expressions is $1 + |xs|$. Using the potential method, we can justify this bound as follows.

Fig. 1. Terminating operational cost semantics rules.

$$\frac{q = max(r,0) \quad q' = max(-r,0)}{V \vdash tick\{r\} \Downarrow () \mid (q,q')} \; Tick \qquad \frac{binop(V(x_1),V(x_2)) \mapsto v}{V \vdash binop(x_1,x_2) \Downarrow v \mid (0,0)} \; Binop$$

$$\frac{}{V \vdash lit \Downarrow lit \mid (0,0)} \; Lit \qquad \frac{V(x) = v}{V \vdash x \Downarrow v \mid (0,0)} \; Var \qquad \frac{V(x_1) = v_1 \quad V(x_2) = v_2}{V \vdash pair(x_1,x_2) \Downarrow (v_1,v_2) \mid (0,0)} \; Pair$$

$$\frac{unop(V(x)) \mapsto v}{V \vdash unop(x) \Downarrow v \mid (0,0)} \; Unop \qquad \frac{V(x_p) = (v_1,v_2) \quad V[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash e \Downarrow v \mid (q,q')}{V \vdash pairMatch(x_p; x_1, x_2.e) \Downarrow v \mid (q,q')} \; PMat$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid (q,q') \quad V[x \mapsto v_1] \vdash e_2 \Downarrow v_2 \mid (p,p')}{V \vdash let(e_1; x.e_2) \Downarrow v_2 \mid (q + max(p - q', 0), p' + max(q' - p, 0))} \; Let$$

$$\frac{V(x_b) = true \quad V \vdash e_t \Downarrow v \mid (q,q')}{V \vdash cond(x_b; e_t; e_f) \Downarrow v \mid (q,q')} \; CondT \qquad \frac{V(x_b) = false \quad V \vdash e_f \Downarrow v \mid (q,q')}{V \vdash cond(x_b; e_t; e_f) \Downarrow v \mid (q,q')} \; CondF$$

$$\frac{func\{f\}(x'.e) \in prog \quad V(x) = v_x \quad V[x' \mapsto v_x] \vdash e \Downarrow v \mid (q,q')}{V \vdash app\{f\}(x) \Downarrow v \mid (q,q')} \; App$$

$$\frac{V(x) = nil \quad V \vdash e_1 \Downarrow v \mid (q,q')}{V \vdash listMatch(x; e_1; x_h, x_t.e_2) \Downarrow v \mid (q,q')} \; LMat0 \qquad \frac{V(x_h) = v_h \quad V(x_t) = v_t}{V \vdash cons(x_h; x_t) \Downarrow v_h :: v_t \mid (0,0)} \; Cons$$

$$\frac{V(x) = v_h :: v_t \quad V[x_h \mapsto v_h, x_t \mapsto v_t] \vdash e_2 \Downarrow v \mid (q,q')}{V \vdash listMatch(x; e_1; x_h, x_t.e_2) \Downarrow v \mid (q,q')} \; LMat1 \qquad \frac{}{V \vdash nil \Downarrow nil \mid (0,0)} \; Nil$$

$$\frac{V[x_2 \mapsto V(x_1), x_3 \mapsto V(x_1)] \vdash e \Downarrow v \mid (q,q')}{V \vdash share(x_1; x_2, x_3.e) \Downarrow v \mid (q,q')} \; Share$$

If 1 resource is initially available, then the base case of the empty list can be paid for. If there is 1 stored per element of the list then 1 resource is released in the cons case of the pattern match. This suffices to pay for the additional :: operation. The remaining potential on xs can be assigned to tl for the recursive call. One can sum these costs to infer that the initial potential $1 + |xs|$ covers the cost of all the :: operations. The AARA type system could describe this with the typing $L^1(\mathbb{Z})$ for xs (describing the linear potential in the superscript) and $\mathbb{Z} \times L^1(\mathbb{Z}) \xrightarrow{1/0} L^0(\mathbb{Z})$ for $snoc$ (describing the initial/remaining resources above the arrow). Another valid type is $\mathbb{Z} \times L^2(\mathbb{Z}) \xrightarrow{1/0} L^1(\mathbb{Z})$, which could be used in a context where the result of $snoc$ must be used to pay for additional cost.

Types The AARA system laid out here supports the types given below. The symbol F gives the types of functions, where q and q' are non-negative rationals. The symbol S gives the remaining non-function types, where *basic* stands for the basic types like *int* or *unit*, and the resource annotation P is an indexed family of rationals representing the coefficients in a linear combination of basic potential functions.

$$F ::= S \xrightarrow{q/q'} S \qquad\qquad S ::= basic \mid L^P(S) \mid S \times S$$

Fig. 2. Partial evaluation operational cost semantics rules.

$$\frac{}{V \vdash e \Downarrow \circ \mid 0} \; Partial \qquad \frac{V \vdash e \Downarrow v \mid (q, q')}{V \vdash e \Downarrow \circ \mid q} \; Termination \qquad \frac{V \vdash e_1 \Downarrow \circ \mid q}{V \vdash let(e_1; x.e_2) \Downarrow \circ \mid q} \; Let1$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid (q, q') \quad V[x \mapsto v_1] \vdash e_2 \Downarrow \circ \mid p}{V \vdash let(e_1; x.e_2) \Downarrow \circ \mid q + max(p - q', 0)} \; Let2$$

$$\frac{V(x) = (v_1, v_2) \quad V[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash e \Downarrow \circ \mid q}{V \vdash pairMatch(x; x_1, x_2.e) \Downarrow \circ \mid q} \; PMat$$

$$\frac{V(x) = true \quad V \vdash e_t \Downarrow \circ \mid q}{V \vdash cond(x; e_t; e_f) \Downarrow \circ \mid q} \; CondT \qquad \frac{V(x) = false \quad V \vdash e_f \Downarrow \circ \mid q}{V \vdash cond(x; e_t; e_f) \Downarrow \circ \mid q} \; CondF$$

$$\frac{func\{f\}(x'.e) \in prog \quad V(x) = v \quad V[x' \mapsto v] \vdash e \Downarrow \circ \mid q}{V \vdash app\{f\}(x) \Downarrow \circ \mid q} \; App$$

$$\frac{V(x) = nil \quad V \vdash e_l \Downarrow \circ \mid q}{V \vdash listMatch(x; e_1; x_h, x_t.e_2) \Downarrow \circ \mid q} \; LMat0$$

$$\frac{V(x) = v_h :: v_t \quad V[x_h \mapsto v_h, x_t \mapsto v_t] \vdash e_2 \Downarrow \circ \mid q}{V \vdash listMatch(x; e_1; x_h, x_t.e_2) \Downarrow \circ \mid q} \; LMat1$$

$$\frac{}{V \vdash tick\{r\} \Downarrow \circ \mid max(r, 0)} \; Tick$$

The typing rules for these types are given in Figure 3 and explained in the following sections. The values of these types are the usual values.

Potential To understand typing rules, it is necessary to define potential. The following potential constructs are generalized from polynomial AARA work [27].

As mentioned, $P = (p_i)_{i \in I}$ is in $\mathbb{Q}^I$ as an indexed family of rationals. Each entry represents a coefficient in a linear combination of basic potential functions. This linearity makes it natural to overload the type of P as a vector or matrix of rationals, so it is treated as such whenever the context is appropriate. Finally, let those basic potential functions be fixed as some family $(f_i)_{i \in I}$, where $f_i(0) = 0$.

We define the potential represented with P using the function ϕ where

$$\phi(n, P) = \sum_i p_i \cdot f_i(n) \; .$$

The function ϕ yields the total potential on a list (excluding the potential of its elements) as a function of the list's size n and its potential annotation P.

We can then relate resource potential between different sizes of list with the shift operator $\lhd : \mathbb{Q}^I \to \mathbb{Q}^I$ and constant difference operator $\delta : \mathbb{Q}^I \to \mathbb{Q}$. These functions need only satisfy the following property equation.

$$\phi(n + 1, P) = \delta(P) + \phi(n, \lhd P) \tag{1}$$

Though we leave open the explicit definition of these functions for generality,

we only later work with instances of them that are linear operators, such that Equation 1 denotes a linear recurrence. Such a refinement leaves $\lhd P$ and $\delta(P)$ linear functions of P.

These functions come in handy for understanding the stored potential in a value of a given type, defined by the potential function Φ as follows.

$$\Phi(v : basic) = 0$$
$$\Phi((v_1, v_2) : A_1 \times A_2) = \Phi(v_1 : A_1) + \Phi(v_2 : A_2)$$
$$\Phi([] : L^P(A)) = 0$$
$$\Phi(h :: t : L^P(A)) = \delta(P) + \Phi(h : A) + \Phi(t : L^{\lhd P}(A))$$

We often need to measure the potential across an entire evaluation context of typed values $V : \Gamma$ given by a typing context Γ and variable bindings V. We do so by extending the definition of potential Φ as follows.

$$\Phi(\emptyset) = 0 \qquad \Phi(V : (\Gamma, v : A)) = \Phi(V : \Gamma) + \Phi(v : A)$$

Finally, we can use these definitions to obtain a closed-form expression for the potential over an entire list (including its elements) with the following:

Lemma 1. *Let* $l = [a_n, ..., a_1]$ *be a list of n values. Then* $\Phi(l : L^P(A)) = \phi(n, P) + \sum_{i=1}^n \Phi(a_i : A)$

Proof. We induct over the structure of the list l.

For the empty list of length 0:

$$\Phi([] : L^P(A)) = 0 = \sum_i p_i \cdot f_i(0) = \phi(0, P) + \sum_{i=1}^0 \Phi(a_i : A)$$

For $l = h :: t$ of size $n + 1$:

$$\Phi(a_{n+1} :: b : L^P(A)) = \delta(P) + \Phi(a_{n+1} : A) + \Phi(l' : L^{\lhd P}(A))$$
$$= \delta(P) + \Phi(a_{n+1} : A) + \phi(n, \lhd P) + \sum_{i=1}^n \Phi(a_i : A)$$
$$= \phi(n + 1, P) + \sum_{i=1}^{n+1} \Phi(a_i : A)$$

We can apply Lemma 1 to the previously defined function *snoc* to see the change in potential between input and output. This difference in potential should bound the resources consumed. For this case, the basic potential functions (f_i) only need contain $\lambda n.n$, and we can let $\lhd(p) = p = \delta((p))$. Letting y be the result of *snoc x xs*, the type $\mathbb{Z} \times L^1(\mathbb{Z}) \xrightarrow{1/0} L^0(\mathbb{Z})$ indicates the following bound

$$\Phi(x : \mathbb{Z}, xs : L^1(\mathbb{Z})) + 1 - \Phi(y : L^0(\mathbb{Z})) = \phi(|xs|, 1) + 1 - \phi(|y|, 0) = |xs| + 1$$

This is exactly the amount of resources consumed, so the bound is tight.

In this work we only consider so-called *univariate* potential, wherein every term in the potential sum is dependent on the length of only one input list. However, different univariate potential summands may depend on different inputs, and thus univariate potential may still be multivariate. The term *multivariate potential* refers to using more general multivariate functions for potential. There is existent work on multivariate potential using polynomial functions [24]. We expect that the work here extends to multivariate potential similarly.

Fig. 3. AARA typing rules.

Basic rules:

$$\frac{}{\Sigma;\emptyset \vdash^{0}_{0} lit : basic}\ Lit \qquad \frac{\Sigma;\Gamma_1 \vdash^{q}_{p} e_1 : A \quad \Sigma;\Gamma_2, x : A \vdash^{p}_{q'} e_2 : B}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{q}_{q'} let(e_1;x.e_2) : B}\ Let$$

$$\frac{}{\Sigma; x : basic \vdash^{0}_{0} unop(x) : basic'}\ Unop \qquad \frac{}{\Sigma; x_i : basic \vdash^{0}_{0} binop(x_1, x_2) : basic'}\ Binop$$

$$\frac{}{\Sigma; x : A \vdash^{0}_{0} x : A}\ Var \qquad \frac{}{\Sigma; x_1 : A_1, x_2 : A_2 \vdash^{0}_{0} pair(x_1, x_2) : A_1 \times A_2}\ Pair$$

$$\frac{\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e : B}{\Sigma; \Gamma, x : A_1 \times A_2 \vdash^{q}_{q'} pairMatch(x; x_1, x_2.e) : B}\ PMat$$

$$\frac{\Sigma; \Gamma, x : bool \vdash^{q}_{q'} e_1 : A \quad \Sigma; \Gamma, x : bool \vdash^{q}_{q'} e_2 : A}{\Sigma; \Gamma, x : bool \vdash^{q}_{q'} cond(x; e_1; e_2) : A}\ Cond$$

Function rules:

$$\frac{A \overset{q/q'}{\to} B \in \Sigma(f)}{\Sigma; x : A \vdash^{q}_{q'} app\{f\}(x) : B}\ App \qquad \frac{func\{f\}(x.e) \in prog \quad \Sigma; x : A \vdash^{q}_{q'} e : B}{A \overset{q/q'}{\to} B \in \Sigma(f)}\ Fun$$

Potential-focused rules:

$$\frac{}{\Sigma; \Gamma \vdash^{max(r,0)}_{max(-r,0)} tick\{r\} : unit}\ Tick \qquad \frac{\Sigma; \Gamma \vdash^{p}_{p'} e : A \quad q \geq p \quad q - p \geq q' - p'}{\Sigma; \Gamma \vdash^{q}_{q'} e : A}\ Relax$$

$$\frac{\Sigma; \Gamma, x : A \vdash^{q}_{q'} e : B \quad A' <: A}{\Sigma; \Gamma, x : A' \vdash^{q}_{q'} e : B}\ SubWeakL \qquad \frac{\Sigma; \Gamma \vdash^{q}_{q'} e : A' \quad A' <: A}{\Sigma; \Gamma \vdash^{q}_{q'} e : A}\ SubWeakR$$

$$\frac{\Sigma; \Gamma, x_2 : A_2, x_3 : A_3 \vdash^{q}_{q'} e : B \quad A_1 \curlyvee (A_2, A_3)}{\Sigma; \Gamma, x_1 : A_1 \vdash^{q}_{q'} share(x_1; x_2, x_3.e) : B}\ Sharing$$

List rules:

$$\frac{}{\Sigma; \emptyset \vdash^{0}_{0} nil : L^P(A)}\ Nil \qquad \frac{}{\Sigma; x_h : A, x_t : L^{\lhd P}(A) \vdash^{\delta(P)}_{0} cons(x_h; x_t) : L^P(A)}\ Cons$$

$$\frac{\Sigma; \Gamma \vdash^{q}_{q'} e_1 : B \quad \Sigma; \Gamma, x_h : A, x_t : L^{\lhd P}(A) \vdash^{q + \delta(P)}_{q'} e_2 : B}{\Sigma; \Gamma, x : L^P(A) \vdash^{q}_{q'} listMatch(x; e_1; x_h, x_t.e_2) : B}\ ListMatch$$

Fig. 4. AARA subtyping and sharing judgments.

$$\frac{\forall i. p_i \geq q_i}{L^P(A) <: L^Q(A)} \; Subtype \qquad\qquad \frac{}{basic \curlyvee (basic, basic)} \; ShareBasic$$

$$\frac{A_1 \curlyvee (A_2, A_3) \quad B_1 \curlyvee (B_2, B_3)}{A_1 \times B_1 \curlyvee (A_2 \times B_2, A_3 \times B_3)} \; SharePair \qquad \frac{A_1 \curlyvee (A_2, A_3) \quad P = Q + R}{L^P(A_1) \curlyvee (L^Q(A_2), L^R(A_3))} \; ShareList$$

Typing Rules The typing rules in Figure 3 use the judgment $\Sigma; \Gamma \vdash^q_{q'} e : A$. In this typing judgment, Γ maps variables to types, while Σ maps function labels to sets of types. This judgment holds when, in the typing environment given by Σ and Γ, the expression e is of type A, subject to the constraints that q and q' are the amount of available resources before and after some evaluation of e. Unlike the judgment $V \vdash e \Downarrow v \mid (q, q')$, these values need not be tight.

By expressing available resources on the turnstile, and potential resources in the types given by Σ, Γ, and A, the type system is set up to formalize the reasoning of the potential method. Theorem 1 shows that it is sound with respect to the operational semantics of Section 2.

Many typing rules preserve the total resource potential they are given, consuming none of it themselves. They therefore usually either have no explicit interaction with potential (e.g. *Lit*) or pass around exactly what they are given (e.g. *Let*). All basic rules in the first block of Figure 3 fit this characterization.

The typing rules concerning functions in second block of Figure 3 are the only to make use of Σ. For each function f defined in *prog* via $func\{f\}(x.e)$, $\Sigma(f)$ refers to the set of types that its body e could be given. That we allow for sets of types is important because recursive calls to a function may not always make use of a type with the same resource annotations; this is called *resource-polymorphic recursion*. Despite these rules capturing the intuition behind typing resource-polymorphic recursion, they are not used in existing implementation, as they lead to infinite type derivations. Nonetheless there exists an effective way to type resource-polymorphic recursion with a finite derivation; see [26]. In the examples provided in this article, it usually suffices to consider only *resource-monomorphic recursion*, wherein inner and outer calls use the same annotation.

All of the rules discussed so far are simply those of existing AARA literature with their parameter for operation cost set to 0 (see e.g. [27]). This does not change their generality, as such constant cost can (and could already in prior work) be simulated using *tick*. Similarly, non-constant costs could be simulated by running helper functions using *tick* the appropriate number of times.

The remaining rules cover sharing, subtype-weakening, and the rules concerning lists. Weakening, though not listed, is also allowed.

Sharing is a form of contraction. By sharing, the rest of the typing rules can become affine, allowing only single usages of a given variable. Intuitively, sharing is meant to prevent duplicating potential across multiple usages of a variable, and instead split the potential across them. The rules for the sharing judgment, indicating how to split potential, can be found in Figure 4. Note that

the rule *ShareList* adds indexed collections of rationals; this should be interpreted pointwise, as if the addends were vectors or matrices.

Subtype-weakening is a form of subtyping based on potential. It discards potential on a list, weakening the upper bound on resources it represents. This rule follows all usual subtyping rules, as well as *Subtype* from Figure 4. Relaxing behaves similarly, but loosens the bounds on the available resources instead.

The intuition for the rules concerning lists in the last block of Figure 3 is that total resources should be conserved between constructions and destructions. Because $\delta(P)$ expresses the difference in potential, it is exactly how many resource units are released after a pattern match on a list of type $L^P(A)$. For the same reason, it is also how many need to be stored when reversing the process and putting an element on a list of type $L^{\lhd P}(A)$. Finally, when a list is empty, it has no room to store potential. Every potential function f_i maps 0 to 0, so an empty list can safely be assigned any scalar of zero potential.

Soundness The soundness of the type system is expressed with the following theorem. It states that the evaluation of an expression e does not require more resources than initially present, and (should evaluation terminate) it leaves at least as many resource as dictated. The proof is a straightforward generalization of the version from [27], but we nonetheless reproduce the proof below.

Theorem 1. *Let* $\Sigma; \Gamma \vdash^{q}_{q'} e : B$ *and* V *provide the variable bindings for* Γ

1. *If* $V \vdash e \Downarrow v \mid (p, p')$ *then* $p \leq \Phi(V : \Gamma) + q$ *and* $p - p' \leq \Phi(V : \Gamma) + q - \Phi(v : B) - q'$
2. *If* $V \vdash e \Downarrow \circ \mid p$ *then* $p \leq \Phi(V : \Gamma) + q$

Proof. Assume V binds Γ's variables and perform nested induction on the type derivation and operational judgment for an expression in let-normal form. We show the induction below only for the terminating operational judgment cases, but the partial-evaluation cases are nearly identical.

(**Base Non-Cons**) Suppose the last rule applied in the typing derivation is any non-*Cons* base case, i.e., *Lit*, *Var*, *Unop*, *Binop*, *Pair*, *Nil*, or *Tick*. Then assume the appropriate terminating operational judgment rule applies. In such a case, one finds $p \leq q$, $p' \geq q'$, and $\Phi(v : B) = \Phi(V : \Gamma)$. This and the non-negativity of potential are sufficient to satisfy the desired inequalities.

(**Base Cons**) Suppose the last rule is *Cons*, so $q = \delta(P)$ and $q' = 0$. Assume the *Cons* operational judgment applies, so that $p = p' = 0$. Note $\Phi(v_h :: v_t : L^P)$ is equal to $\delta(P) + \Phi(v_h : A) + \Phi(v_t : L^P(A))$ by definition. This identity and the non-negativity of potential satisfy the desired inequalities.

(**Step Implicit Inequalities**) Suppose the last rule is one of *SubWeakL*, *SubWeakR*, *Relax*, or substructural weakening, and assume some operational judgment applies. Each typing requires a similar typing judgment as a premiss. Further, none changes any values, so the same operational judgment still applies. Thus, the inductive hypothesis applies, and gives almost the inequalities we need. Each case provides the inequalities needed to finish. For subtype-weakening, it is sufficient note that $C <: D$ entails $\Phi(v : C) \geq \Phi(v : D)$, since C is pointwise greater-then-or-equal to D. For *relax*, the premisses of the *relax* rule directly

include the inequalities needed to complete the case. And we can complete the substructural weakening case by noting that the non-negativity of potential entails $\Phi(V : \Gamma, v : A) \geq \Phi(V : \Gamma)$.

(**Step Let**) Suppose the last rule is *Let*, and suppose its operational judgment applies. The premisses of the typing rule require that $\Sigma; \Gamma_1 \vdash^{q}_{r} e_1 : A$ and $\Sigma; \Gamma_2, x : A \vdash^{r}_{q'} e_2 : B$. The premisses of the operational judgment require that $V \vdash e_1 \Downarrow v_1 \mid (s, s')$ and $V[x \mapsto v_1] \vdash e_2 \Downarrow v_2 \mid (t, t')$, where $p = s + max(t - s', 0)$ and $p' = t' + max(s' - t, 0)$. Applying the inductive hypothesis to these premiss pairs and adding the resulting inequalities cancels terms to complete the case.

(**Step Sharing**) Suppose the last is *Sharing*, so that $\Gamma = \Gamma', x_1 : A_1$. It requires as a premiss that $\Sigma; \Gamma', x_2 : A_2, x_3 : A_3 \vdash^{q}_{q'} e : B$, where $A_1 \curlyvee (A_2, A_3)$. Assuming the operational judgment *Share* applies, $V[x_2 \mapsto V(x_1), x_3 \mapsto V(x_1)] \vdash e \Downarrow v \mid (p, p')$ also holds. The inductive hypothesis applies, yielding the needed inequalities, but for x_2, x_3 instead of x_1. However, the sharing relation ensures that $\Phi(v_1 : A_1) = \Phi(v_2 : A_2, v_3 : A_3)$, and this identity finishes the case.

(**Step ListMatch**) Suppose the last is *ListMatch*, so $\Gamma = \Gamma', x : L^P(A)$. There are two operational judgments which could apply: *LMat0* and *LMat1*.

Suppose the former judgment applies. It requires that $V \vdash e_1 \Downarrow v \mid (p, p')$. At the same time, the *ListMatch* rule requires as a premiss that $\Sigma; \Gamma' \vdash^{q}_{q'} e_1 : B$. The inductive hypothesis applies, yielding the needed inequalities, but for Γ' instead of Γ. However, because $\Phi(nil : L^P(A)) = 0$, we see $\Phi(V : \Gamma') = \Phi(V : \Gamma)$, and the desired inequalities result.

Suppose instead the latter judgment applies. This judgment requires as a premiss that $V[x_h \mapsto v_h, x_t \mapsto v_t] \vdash e_2 \Downarrow v \mid (p, p')$. At the same time, the *ListMatch* rule requires that $\Sigma; \Gamma', x_h : A, x_t : L^{\triangleleft P}(A) \vdash^{q + \delta(P)}_{q'} e_2 : B$. The inductive hypothesis applies, telling us that $p - p' \leq \Phi(V : \Gamma', v_h : A, v_t : L^{\triangleleft P}(A)) + q + \delta(P) - \Phi(v : B) - q'$ and $p \leq \Phi(V : \Gamma', v_h : A, v_t : L^{\triangleleft P}(A)) + q + \delta(P)$. By definition, $\Phi(v_h :: v_t : L^P) = \delta(P) + \Phi(v_h : A) + \Phi(v_t : L^P(A))$, and applying this identity to the inequalities yields the inequalities needed.

(**Step Cond**) Suppose the last rule is *Cond*, and that either of the *CondT* or *CondF* operational judgments apply. In either case, applying the inductive hypothesis to its premiss and the premiss of *Cond* gives the needed inequalities.

(**Step PMat**) Suppose that the last rule applied is *PMat*, so that $\Gamma = \Gamma', x : A_1 \times A_2$. This rule would require as a premiss that $\Sigma; \Gamma', x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e' : B$, for e' the body of the match statement e. Suppose the *PMat* operational judgment applies. This judgment requires as a premiss that $V[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash e' \Downarrow v \mid (p, p')$, where the value of x is (v_1, v_2). Applying the inductive hypothesis to these premisses followed by the definitional identity $\Phi((v_1, v_2) : A_1 \times A_2) = \Phi(v_1 : A_1) + \Phi(v_2 : A_2)$ completes the case.

(**Step App**) Suppose the last rule is *App*. Note that this rule requires *Fun* as a premiss, which in turn requires $\Sigma; x : A \vdash^{q}_{q'} e' : B$ where e' is the body of the function being applied. If the *App* operational judgment applies, its premiss would require $V[x' \mapsto V(x)] \vdash e \Downarrow v \mid (p, p')$. Although e' might not be a smaller

expression than e, the operational judgment derivation still shrinks. This means the inductive hypothesis applies, and it gives the exact inequalities needed.

Type Inference Type inference for the Hindley-Milner part of the type system is decidable [21,41]. The only new barrier for automating inference in AARA is obtaining witnesses for all the coefficients in each annotation P in a derivation.

Each typing rule naturally gives a set of linear constraints on the entries of P. If the relation given by $\lhd$ and δ can likewise be expressed with linear constraints, then all such constraints are linear. So long as $|P|$ is finite, this forms a linear program. A linear program solver can then find minimal witnesses efficiently.

Existing AARA literature (see e.g. [27]), however, uses binomial coefficients as the basis functions for P, of which there are infinitely many. This nonetheless works because only a particular finite prefix of their set, $\binom{-}{1}, \ldots, \binom{-}{k}$, are used as a basis in a given analysis. Each such prefix basis also yields the same locally-definable shift operation: the linear equality $\lhd p_i = p_i + p_{i+1}$, where p_k is the coefficient of $\binom{-}{k}$ and is 0 if the function is outside the prefix. As this is a linear relation, and each prefix is finite, inference can be performed via linear program. The prefix bases of binomial coefficients thereby form an infinite family of finite bases, each of which allows automated inference of resource polynomials up to a fixed degree in the AARA system.

As a caveat, not all programs use resources in a manner compatible with the AARA system. Indeed, it is undecidable whether or not a program uses e.g. polynomial amounts of resources, as this could solve the halting problem.

4   Exponential Potential

Stirling numbers of the second kind $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$ count the number of ways to form a k-partition of a set of n elements. These can be used to express exponential potential functions similarly to how binomial coefficients can express polynomial ones. In particular, we make use of Stirling numbers with arguments n, k offset by 1, $\left\{ \begin{smallmatrix} n+1 \\ k+1 \end{smallmatrix} \right\}$, so that $\phi(n, P) = \sum_i p_i \cdot \left\{ \begin{smallmatrix} n+1 \\ i+1 \end{smallmatrix} \right\}$. While other bases could also express exponential potential, these offset Stirling numbers have a few particularly desirable properties, which are described in this section.

Simple Shift Operation Like binomial coefficients, the prefixes of the basis of the offset Stirling numbers of the second kind form an infinite family of finite bases, each of which allows automated inference in the AARA system. However, these potential functions are exponential rather than polynomial.

Stirling numbers of the second kind satisfy the recurrence $\left\{ \begin{smallmatrix} n+1 \\ k+1 \end{smallmatrix} \right\} = (k + 1)\left\{ \begin{smallmatrix} n \\ k+1 \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$. This recurrence allows the $\lhd$ operation to have the same local definition for every annotation entry in every prefix basis: $\lhd p_i = (i+1)p_i + p_{i+1}$, where p_k is the coefficient of $\left\{ \begin{smallmatrix} n+1 \\ k+1 \end{smallmatrix} \right\}$, and is 0 if the function index is outside the chosen prefix. Given this definition for $\lhd$ and letting $\delta(P) = p_0$, we find $p_0 + \sum_i \lhd p_i \left\{ \begin{smallmatrix} n+1 \\ i+1 \end{smallmatrix} \right\} = \sum_i p_i \left\{ \begin{smallmatrix} n+2 \\ i+1 \end{smallmatrix} \right\}$, satisfying Equation 1.

This shift operation yields a linear relation, as the coefficient of a given p_i is a constant scalar. Thus, exactly like when using binomial coefficients, inference is automatable via linear programming. Certain other exponential bases, like Gaussian binomial coefficients, could be similarly automated.

Expressivity Because $\left\{{n+1 \atop k+1}\right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i}(i+1)^n \in \Theta((k+1)^n)$, the offset Stirling numbers of the second kind can form a linear basis for the space of sums of exponential functions. Each function $\lambda n.b^n$ with $b \geq 1$ can be expressed as a linear combination of the functions $\lambda n.\left\{{n+1 \atop k+1}\right\}$.

The function $\lambda n.\left\{{n+1 \atop k+1}\right\}$ is also non-negative for natural n, and non-decreasing with respect to n. These are two natural properties to require of basic potential functions, since amortized analysis requires non-negative resources, and larger inputs should not usually become cheaper to process. Further, the properties are preserved by non-negative linear (i.e. conical) combination, and by $\lhd$ when defined with a non-negative linear recurrence - the combinations given by P and $\lhd P$ always satisfy the two potential function properties.

Ensuring these properties for more general potential functions requires determining if such a function on a natural domain is always non-negative. This is non-trivial. In the existing literature on multivariate polynomials, we find this is *undecidable* in the worst case [40]. However, restricting to non-negative linear (that is, *conical*) combinations of non-negative, non-decreasing functions - as we have done here - gives simple linear constraints that ensure both desired properties. For finite bases, this is easily handled via linear programming.

When considering expressivity in this conical combination model of potential functions, one finds some otherwise-valid potential functions are not be expressible in the conical space given by the offset Stirling number functions. Nonetheless, Stirling number functions are a *maximally expressive* basis; it is not possible to express additional potential functions using a different basis without losing expressibility elsewhere. Notably, the standard exponential basis is *not* maximal in this sense. The formal statement of such maximal expressivity is generalized in the theorem below. Any finite, sequential subset of the offset Stirling number functions satisfy the prerequisites of this theorem, as do the binomial coefficient functions and other well-known functions like the Gaussian polynomials.

Theorem 2. *Let $\{f_i\}$ be a finite set of linearly independent functions on the naturals that are non-negative and non-decreasing. Let $f_i(n)$ be 0 until $n \geq i$, and let $i \leq j$ imply that $O(f_i) \subseteq O(f_j)$, with asymptotic equality only when $i = j$. Let L be the linear span (collection of linear combinations) of $\{f_i\}$, and let C be its conical span (collection of conical combinations).*

There does not exist another linearly independent basis $\{g_i\}$ with linear span L and conical span $D \supsetneq C$ such that each function in $\{g_i\}$ is non-negative and non-decreasing. That is, $\{f_i\}$ has a maximally expressive conical span.

Proof. Suppose there is such a basis $\{g_i\}$. We express each basis $\{f_i\}$ and $\{g_i\}$ with linear combinations of the other, and derive a contradiction.

If there is any function in the conical span D of $\{g_i\}$ that is not in C, then this is the case for some basis function g_k. Because $g_k \in L$, it can be written as a linear combination of $\{f_i\}$; let $\sum_i \alpha_i f_i = g_k$. Because $g_k \notin C$, there is at least one coefficient $\alpha_i < 0$; let it be α_m. In case there are multiple candidate elements g_k, pick g_k to be the basis function such that this index m is minimized.

We then see that $g_k(m) = \sum_i \alpha_i f_i(m) = (\sum_{i<m} \alpha_i f_i(m)) + \alpha_m f_m(m)$ because $f_i(m)$ for $i > m$ is 0. This yields two observations: First, $m < k$, as

otherwise the fastest-growing term of g_k would be negative, but g_k is never negative. Second, the term $\alpha_m f_m(m)$ is negative, yet $g_k \geq 0$, so it must be that $\sum_{i<m} \alpha_i f_i(m) > 0$. Thus there exists a coefficient $\alpha_p > 0$ where $p < m$.

Now we look at representing $\{f_i\}$ with $\{g_i\}$. Because the conical span D contains C, it can represent each f_i as a conical combination. Notably, a given f_i cannot be represented only with functions outside of $\Omega(f_i)$, nor any function outside of $O(f_i)$, due to growth rates. There is therefore at least one function in $\{g_i\}$ that is $\Theta(f_i)$, for each i. Since the linear span of these corresponding g_i already has the same (finite) dimension as L, any additional functions would not be linearly independent. Due to this, we can say $g_i \in \Theta(f_i)$ uniquely for each i.

Take f_k in particular as a conical combination of $\{g_i\}$. We now consider replacing each element of $\{g_i\}$ in that conical combination with its equivalent linear combination of elements of $\{f_i\}$. Because of the above correspondence of growth rates, there must be a positive coefficient for g_k. Because g_k has positive weight α_p on f_p where $p < m < k$, another basis function g_i in the conical combination must have negative weight on f_p to cancel it out in their linear combination. However, g_k was picked such that it had the lowest index m with negative weight across all $\{g_i\}$; it is contradictory for there to be such a $p < m$.

Natural Semantics The values of $\left\{{n+1 \atop k+1}\right\}$ count the number of ways to pick k non-empty disjoint subsets of n elements. Many programs with exponential resource use iterate over collections of subsets, so these numbers naturally arise.

Recall the naive solution to subset sum from the introduction. The algorithm iterates through all the subsets of numbers in the input list. When considering Fagin's descriptive complexity result that NP problems are precisely those expressible in existential second order logic [14], it becomes clear that naive solutions to any NP-complete problem fit this characterization: naively brute-forcing through second order terms to find an existential witness is just iterating through tuples of subsets.

Example Consider the naive solution to subset sum from the introduction. One can verify that the number of Boolean and arithmetic operations used on an input of size n is $3 * 2^n - 2$ by induction. We find the same bound here by preceding each such operation with an explicit $tick\{1\}$ operation. Thee AARA type system then verifies that the type of *subsetSum* is $L^3(\mathbb{Z}) \times \mathbb{Z} \xrightarrow{1/0} bool$.

Here is the code again, with type annotations on each line tracking the amount of $\left\{{n+1 \atop 2}\right\}$ potential on lists, and comments tracking available constant potential. For clarity, the code is re-written in a let-normal form, and sharing locations are marked.

```
let subsetSum nums:L³(ℤ) target =                    (* 1 *)
    match nums:L³(ℤ) with
    | [] →                                           (* 1 *)
        tick 1; target = 0                           (* 0 *)
    | hd::(tl:L⁶(ℤ)) →                               (* 4 *)
        tick 1; let newTarget = target - hd in       (* 3 *)
        (* share tl:L⁶(ℤ) as L³(ℤ), L³(ℤ) *)
        let withNum = subsetSum tl:L³(ℤ) newTarget in (* 2 *)
```

```
          let without = subsetSum tl:L³(ℤ) target in    (* 1 *)
          tick 1; withNum || without                    (* 0 *)
```
The indicated values yield witnesses for the AARA typing rules, so we know via soundness that the difference between initial and ending potential gives an upper bound on how many operations were used. That difference is $1+3*\left\{{n+1 \atop 2}\right\} = 3*2^n - 2$, where n is the size of *nums*, exactly the amount used.

Exponential terms with higher bases than 2 can come into play with more recursive calls, like in the code below enumerating the 3^n ways to put n labelled balls into 3 labelled bins.

```
let helper xs:L^{2,2}(ℤ) a b c =                              (* 1 *)
      match xs with
      | [] →                                                 (* 1 *)
            tick 1; [(a,b,c)]                                 (* 0 *)
      | hd::(tl:L^{6,6}(ℤ)) →                                 (* 3 *)
            (* share tl:L^{6,6}(ℤ) as L^{2,2}(ℤ), L^{2,2}(ℤ), L^{2,2}(ℤ) *)
            let newA = hd::a in                              (* 3 *)
            let tmp1 = helper tl:L^{2,2}(ℤ) newA b c in   (* 2 *)
            let newB = hd::b in                              (* 2 *)
            let tmp2 = helper tl:L^{2,2}(ℤ) a newB c in   (* 1 *)
            let newC = hd::c in                              (* 1 *)
            let tmp3 = helper tl:L^{2,2}(ℤ) a b newC in   (* 0 *)
            tmp1 @ tmp2 @ tmp3                               (* 0 *)

let ballBins3 xs:L^{2,2}(ℤ) =                                 (* 1 *)
      helper xs:L^{2,2}(ℤ) [] [] []                           (* 0 *)
```
By paying a unit of resource for each such way using *tick*, we can use AARA to bound the count. It assigns a type of $L^{2,2}(\mathbb{Z}) \xrightarrow{1/0} L^{0,0}(L^{0,0}(\mathbb{Z}) \times L^{0,0}(\mathbb{Z}) \times L^{0,0}(\mathbb{Z}))$ to *ballBins*3, where the superscript tracks $\left\{{n+1 \atop 2}\right\}$ and $\left\{{n+1 \atop 3}\right\}$ potential, respectively. Since $2\left\{{n+1 \atop 3}\right\} + 2\left\{{n+1 \atop 2}\right\} + 1 = 3^n$, this bound is exact.

5 Mixed Potential

It is possible to combine the existing polynomial potential functions with these new exponential potential functions to not only conservatively extend both, but further represent potentials functions with their products. This space represents functions in $\Theta(n^k(b+1)^n)$ for naturals k, b, and does so with terms of the form $\binom{n}{k}\left\{{n+1 \atop b+1}\right\}$ so that $\phi(n, P) = \sum_{b,k} p_{b,k} \cdot \binom{n}{k}\left\{{n+1 \atop b+1}\right\}$. Note that for k or b equal to 0, the potential functions here reduce to the offset Stirling numbers or binomial coefficients, respectively.

The methods used to combine these potential functions here can easily be generalized to combine any two suitable sets.

Simple Shift Operation It is straightforward to find a linear recurrence for these products by distributing over their linear recurrences.

$$\binom{n+1}{k+1}\left\{{n+2 \atop b+2}\right\} = \left(\binom{n}{k+1}+\binom{n}{k}\right)\left((b+2)\left\{{n+1 \atop b+2}\right\}+\left\{{n+1 \atop b+1}\right\}\right)$$
$$= (b+2)\binom{n}{k+1}\left\{{n+1 \atop b+2}\right\}+(b+2)\binom{n}{k}\left\{{n+1 \atop b+2}\right\}+\binom{n}{k+1}\left\{{n+1 \atop b+1}\right\}+\binom{n}{k}\left\{{n+1 \atop b+1}\right\}$$

As before, this yields a definition for δ and $\lhd$ with Equation 1. Letting P now be indexed by pairs b, k: $\lhd p_{b,k} = (b+1)p_{b,k} + (b+1)p_{b,k+1} + p_{b+1,k} + p_{b+1,k+1}$,

and $\delta(P) = p_{0,1} + p_{1,0} + p_{1,1}$. Noting that these definitions are linear again yields automatability for finite (2-dimensional) prefixes of the basis.

Expressivity The product of non-negative, non-decreasing functions is still non-negative and non-decreasing, so products of valid potential functions are still valid. Soundness is preserved by letting p_0 be shorthand for the new constant function coefficient $p_{0,0}$ wherever it is used in Theorem 1. Moreover, maximality of expressivity is preserved, simply by giving index pairs the ordering relation $(i_1, i_2) \le (j_1, j_2) \iff i_1 \le j_1 \land i_2 \le j_2$ and applying Theorem 2.

Example Consider bounding the number of Boolean and arithmetic operations in a variation of subset sum: *single-use* subset sum. Here the input may contain duplicate numbers that should be ignored, so as to treat the input as a true set. This is a trivial change to the mathematical problem, but one that real code might have to deal with, depending on the implementation of sets.

The code can be changed to handle this by removing all later duplicates of each number it reaches, so that later recursive calls will never see the number again. It is easy to create a function *remove* of type $\mathbb{Z} \times L^{a+1,b,c}(\mathbb{Z}) \xrightarrow{d/d} L^{a,b,c}(\mathbb{Z})$ to do this for any a, b, c, d, where the superscript values represent linear, $\left\{ {n+1 \atop 2} \right\}$, and $n\left\{ {n+1 \atop 2} \right\}$ potential, respectively.

One can prove by induction that at most $4 * 2^n - n - 3$ Boolean or arithmetic operations are required. Although this can be bounded with only exponential functions, the purely exponential potential system cannot reason about the exact (linear) cost associated with *remove*, and overestimates the bound to be in $\theta(3^n)$. This mixed system can provide a better (though still loose) bound of $n2^n + 2 * 2^n - n - 1$, giving a type of $L^{0,2,1}(\mathbb{Z}) \times \mathbb{Z} \xrightarrow{1/0} bool$ to *subSum*1. After showing this derivation, we will show how to find the exact bound with AARA.

The following is the single-use subset sum code, with comments on each line tracking the amount of available resources on each line. For clarity, we indicate sharing and subtype-weakening locations.

```
let subSum1 nums:L^{0,2,1}(ℤ) target =                    (* 1 *)
   match nums with
   | [] →                                                 (* 1 *)
      tick 1; target = 0                                  (* 0 *)
   | hd::(tl:L^{1,6,2}(ℤ)) →                               (* 4 *)
      let otherNums:L^{0,6,2}(ℤ) = remove hd tl:L^{1,6,2}(ℤ) in   (* 4 *)
      tick 1; let newTarg = target - hd in               (* 3 *)
      (* weaken otherNums:L^{0,6,2}(ℤ) to L^{0,4,2}(ℤ) *)
      (* share otherNums:L^{0,4,2}(ℤ) as L^{0,2,1}(ℤ), L^{0,2,1}(ℤ) *)
      let withNum = subSum1 otherNums:L^{0,2,1}(ℤ) newTarg in (* 2 *)
      let without = subSum1 otherNums:L^{0,2,1}(ℤ) target in  (* 1 *)
      tick 1; withNum || without                         (* 0 *)
```

The difference between initial and ending potential gives the upper bound of $1 + 2\left\{ {n+1 \atop 2} \right\} + n * \left\{ {n+1 \atop 2} \right\} = n2^n + 2 * 2^n - n - 1$ Boolean or arithmetic operations.

Note that we use the subtype-weakening rule, throwing away 2 units of $\left\{ {n+1 \atop 2} \right\}$ potential. This indicates why the bound is not tight. Next we show how to improve this bound using potential demotion.

Demotion There is one special exception to the non-negativity of potential annotations that may be added due to the particular nature of the relation between binomial coefficients and Stirling numbers. It represents the concept of *demoting* exponential potential into polynomial potential.

The relevant relation is $\left\{{n+1 \atop 2}\right\} = 2^n - 1 = \sum_{i=1}^{\infty} \binom{n}{i} \geq \sum_{i=1}^{k} \binom{n}{i}$. This allows a unit of $\left\{{n+1 \atop 2}\right\}$ potential to account for one unit *each* of all non-constant binomial coefficient potentials. We can express this with the following additional subtyping rule. In this rule we interpret the 2-dimensional indexing of the potential annotation as a matrix, and we let $\overrightarrow{p}$ refer to the vector of potential entries at index coordinates $0, i$ for $i \geq 1$.

$$P = R + \begin{bmatrix} 0 & \overrightarrow{p} \\ r & 0 \end{bmatrix} \quad Q = R + \begin{bmatrix} 0 & \overrightarrow{p} + s * \overrightarrow{1} \\ r - s & 0 \end{bmatrix}$$
$$\frac{}{L^P(A) <: L^Q(A)} \; Demote$$

Theorem 3. *The demotion rule is sound.*

Proof. We need only show that $C <: D$ implies $\Phi(v : D) \leq \Phi(v : C)$ for unchanged values v. The rest of soundness then follows as in Theorem 1. To do so, it is sufficient to show for $l = [a_1, \ldots, a_n]$ we have $\Phi(a : L^Q(A)) \leq \Phi(a : L^P(A))$.

Without loss of generality, we need only consider where $R = 0$.

$$\Phi(l : L^Q(A)) = \phi(n, Q) + \sum_{i=1}^{n} \Phi(a_i : A)$$
$$= (r - s)\left\{{n+1 \atop 2}\right\} + \sum_{i=1}^{k}(\overrightarrow{p}_{i-1} + s)\binom{n}{i} + \sum_{i=1}^{n} \Phi(a_i : A)$$

$$= \sum_{i=1}^{\infty}(r - s)\binom{n}{i} + \sum_{i=1}^{k}(\overrightarrow{p}_{i-1} + s)\binom{n}{i} + \sum_{i=1}^{n} \Phi(a_i : A)$$
$$\leq \sum_{i=1}^{\infty} r\binom{n}{i} + \sum_{i=1}^{k} \overrightarrow{p}_{i-1}\binom{n}{i} + \sum_{i=1}^{n} \Phi(a_i : A)$$
$$= r\left\{{n+1 \atop 2}\right\} + \sum_{i=1}^{k} \overrightarrow{p}_{i-1}\binom{n}{i} + \sum_{i=1}^{n} \Phi(a_i : A)$$
$$= \phi(n, P) + \sum_{i=1}^{n} \Phi(a_i : A) = \Phi(l : L^P(A))$$

As a corollary, this allows us to loosen the constraint that every annotation P contains only non-negative rationals. In particular, it is no longer required that $\forall i. p_{0,i} \geq 0$. Instead, we require that $\forall i. p_{0,i} + p_{1,0} \geq 0$. Each unit of $\left\{{n+1 \atop 2}\right\}$ potential may "pay" for one unit of deficit from each polynomial potential function. Because this is still a linear constraint, type inference remains automatable.

Using *Demote*, tighter bounds can be obtained. Consider the single-use subset sum solution from the previous section. Here it is again below, but this time allowing the linear potential to be paid for by $\left\{{n+1 \atop 2}\right\}$ potential. AARA can now provide a type of $L^{-1,4,0}(\mathbb{Z}) \times \mathbb{Z} \xrightarrow{1/0} bool$ for $subSum1$, corresponding to the exact upper bound of $4 * 2^n - n - 3$ operations. This time $n * \left\{{n+1 \atop 2}\right\}$ is elided in the annotated potentials, as it is not needed.

```
let subSum1 nums:L^{-1,4}(Z) target =                          (* 1 *)
    match nums with
    | [] →                                                    (* 1 *)
        tick 1; target = 0                                    (* 0 *)
    | hd::(tl:L^{-1,8}(Z)) →                                  (* 4 *)
        let otherNums:L^{-2,8}(Z) = remove hd tl:L^{-1,8}(Z) in   (* 4 *)
        tick 1; let newTarg = target - hd in                  (* 3 *)
        (* share otherNums:L^{-2,8}(Z) as L^{-1,4}(Z), L^{-1,4}(Z) *)
        let withNum = subSum1 otherNums:L^{-1,4}(Z) newTarg in (* 2 *)
        let without = subSum1 otherNums:L^{-1,4}(Z) target in  (* 1 *)
        tick 1; withNum || without                            (* 0 *)
```

The difference between initial and ending potential gives the upper bound of $1 - n + 4\{{n+1 \atop 2}\} = 4 * 2^n - n - 3$, as desired.

6   Exponentials, Polynomials, and Logarithms

The addition of exponential potential also allows for the inference of previously nonderivable polynomial-resource types for certain programs. One such way this can happen is by compacting the potential of a list into a new list logarithmic in size to the first. Performing exponential-cost operations, such as *subsetSum*, on a list of logarithmic size only has linear cost in total.

In the code below, *log* takes a list x of length n and returns a list of length roughly $log_2(n)$. If x begins with one unit of linear potential, the type system assigns the output of *log* one unit of base-2 exponential ($2^n - 1$) potential. We show in the code below with types of the form $L^{a,b}$, where a is the linear potential, and b is the base-2 exponential potential. This lets us find that *half* can have type $L^{1,0}(\mathbb{Z}) \xrightarrow{0/0} L^{2,0}(\mathbb{Z})$ and *log* has type $L^{1,0}(\mathbb{Z}) \xrightarrow{0/0} L^{0,1}(\mathbb{Z})$. The typing of *log* shows the conversion from linear to exponential potential.

```
let half x: L^{1,0}(Z) =                            (* 0 *)
    match x with
    | [] →                                          (* 0 *)
        []: L^{2,0}(Z)                              (* 0 *)
    | hd::(tl: L^{1,0}(Z)) →                        (* 1 *)
        match tl with
        | [] →                                      (* 1 *)
            []: L^{2,0}(Z)                          (* 1 *)
        | hd2::(tl2: L^{1,0}(Z)) →                  (* 2 *)
            let halfTail: L^{2,0}(Z) = half tl2 in  (* 2 *)
            (hd::halfTail): L^{2,0}(Z)              (* 0 *)

let log x: L^{1,0}(Z) =                             (* 0 *)
    match x with
    | [] →                                          (* 0 *)
        []: L^{0,1}(Z)                              (* 0 *)
    | hd::(tl: L^{1,0}(Z)) →                        (* 1 *)
        let halfTail: L^{2,0}(Z) = half tl in       (* 1 *)
        let subSoln: L^{0,2}(Z) = log halfTail in   (* 1 *)
```

```
(hd::subSoln): L^{0,1}(ℤ)                    (* 0 *)
```

Typing *log* above requires resource-polymorphic recursion. However, this can be justified by noting that the above can be thought of to show *half* has type $L^{a,0}(\mathbb{Z}) \stackrel{0/0}{\to} L^{2a,0}(\mathbb{Z})$ and *log* has type $L^{a,0}(\mathbb{Z}) \stackrel{0/0}{\to} L^{0,a}(\mathbb{Z})$ for any $a \geq 0$.

Coincidentally, *log* conversion of linear to exponential potential certifies that the output list's size can be bounded by a logarithm of the input's size. Nonetheless, logarithmic *potential* is not directly compatible with the approach this work takes. Sublinear functions have negative second derivatives, and this yields negative annotation entries under $\lhd$ applications. This may not be insurmountable, as the demotion rule showed here, but new ideas are needed overall. Logarithmic potential has been explored in [32], though the approach there departs from the automatable AARA framework of linear constraint solving.

7   Conclusion and Future Work

Using Stirling numbers of the second kind allows for the automated inference of exponential resource usages via Automatic Amortized Resource Analysis. This may be combined with the existing polynomial system, allowing mixtures of polynomial and exponential functions to be inferred. Under this system, more kinds of programs can now be automatically analyzed, in particular those making use of multiple recursive calls, or logarithmically-sized lists. Finally, the framework put in place to accomplish this separates the concerns of the type system and potential functions, paving the way to allow modular addition of different potential functions. Future work could extend the work here to cover additional language features supported in polynomial AARA literature, like trees [22].

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07) (2007)
2. Albert, E., Fernández, J.C., Román-Díez, G.: Non-cumulative Resource Analysis. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15) (2015)
3. Albert, E., Genaim, S., Masud, A.N.: On the Inference of Resource Usage Upper and Lower Bounds. ACM Transactions on Computational Logic **14**(3) (2013)
4. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: 19th Euro. Symp. on Prog. (ESOP'10) (2010)
5. Avanzini, M., Lago, U.D., Moser, G.: Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In: 29th Int. Conf. on Functional Programming (ICFP'15) (2012)
6. Avanzini, M., Moser, G.: A Combination Framework for Complexity. In: 24th International Conference on Rewriting Techniques and Applications (RTA'13) (2013)
7. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: Algebraic Bound Computation for Loops. In: Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10) (2010)
8. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated Resource Analysis with Coq Proof Objects. In: 29th International Conference on Computer-Aided Verification (CAV'17) (2017)

9. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional Certified Resource Bounds. In: 36th Conference on Programming Language Design and Implementation (PLDI'15) (2015), artifact submitted and approved

10. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. In: Computer Aided Verification - 29th International Conference (CAV '17). pp. 41–63 (2017)

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

12. Danner, N., Licata, D.R., Ramyaa, R.: Denotational Cost Semantics for Functional Languages with Inductive Types. In: 29th Int. Conf. on Functional Programming (ICFP'15) (2012)

13. Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session types. In: 33th ACM/IEEE Symposium on Logic in Computer Science (LICS'18) (2018)

14. Fagin, R.: Generalized First-Order Spectra, and Polynomial-Time Recognizable Sets. SIAM-AMS Proc. **7** (01 1974)

15. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic Average-case Analysis of Algorithms. Theoret. Comput. Sci. **79**(1), 37–109 (1991)

16. Flores-Montoya, A., Hähnle, R.: Resource Analysis of Complex Programs with Cost Equations. In: Programming Languages and Systems - 12th Asian Symposiu (APLAS'14) (2014)

17. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower Runtime Bounds for Integer Programs. In: Automated Reasoning - 8th International Joint Conference (IJCAR'16) (2016)

18. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09) (2009)

19. Guneau, A., Charguraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 10801, pp. 533–560. Springer (Apr 2018)

20. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press (2016)

21. Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society **146**, 29–60 (1969), http://www.jstor.org/stable/1995158

22. Hoffmann, J.: Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis. Ph.D. thesis, Ludwig-Maximilians-Universität München (2011)

23. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symposium on Principles of Programming Languages (POPL'11) (2011)

24. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. (2012)

25. Hoffmann, J., Das, A., Weng, S.C.: Towards Automatic Resource Bound Analysis for OCaml. In: 44th Symposium on Principles of Programming Languages (POPL'17) (2017)

26. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: 8th Asian Symposium on Programming Languages (APLAS'10) (2010)

27. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th European Symposium on Programming (ESOP'10) (2010)

28. Hoffmann, J., Shao, Z.: Type-Based Amortized Resource Analysis with Integers and Arrays. In: 12th International Symposium on Functional and Logic Programming (FLOPS'14) (2014)
29. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. In: 24th European Symposium on Programming (ESOP'15) (2015)
30. Hoffmann, J., Shao, Z.: Type-Based Amortized Resource Analysis with Integers and Arrays. J. Funct. Program. (2015)
31. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03) (2003)
32. Hofmann, M., Moser, G.: Analysis of logarithmic amortised complexity (2018)
33. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th ACM Symp. on Principles of Prog. Langs. (POPL'10) (2010)
34. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In: 16th Symp. on Form. Meth. (FM'09) (2009)
35. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest Precondition Reasoning for Expected RunTimes of Probabilistic Programs. In: Proceedings of the European Symposium on Programming Languages and Systems (ESOP'16). Springer (2016)
36. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.: Compositional recurrence analysis revisited. In: Conference on Programming Language Design and Implementation (PLDI'17) (2017)
37. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. Proc. ACM Program. Lang. **2**(POPL), 54:1–54:33 (Dec 2017)
38. Lichtman, B., Hoffmann, J.: Arrays and References in Resource Aware ML. In: 2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17) (2017)
39. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: Proceedings of the 44th Symposium on Principles of Programming Languages (POPL'17) (2017)
40. Matiyasevich, Y.V.: The Diophantineness of Enumerable Sets. In: Doklady Akademii Nauk. vol. 191, pp. 279–282. Russian Academy of Sciences (1970)
41. Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences **17**, 348–375 (1978)
42. Mvel, G., Jourdan, J.H., Pottier, F.: Time credits and time receipts in Iris. In: Caires, L. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 11423, pp. 1–27. Springer (Apr 2019)
43. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: Resource analysis for probabilistic programs. In: 39th Conference on Programming Language Design and Implementation (PLDI'18) (2018)
44. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Verifying and Synthesizing Constant-Resource Implementations with Types. In: 38th IEEE Symposium on Security and Privacy (S&P '17) (2017)
45. Nipkow, T., Brinkop, H.: Amortized complexity verified. J. Autom. Reasoning **62**(3), 367–391 (2019)
46. Niu, Y., Hoffmann, J.: Automatic space bound analysis for functional programs with garbage collection. In: 22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'18) (2018)

47. Noschinski, L., Emmes, F., Giesl, J.: Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. J. Autom. Reasoning **51**(1), 27–56 (2013)
48. Radiček, I., Barthe, G., Gaboardi, M., Garg, D., Zuleger, F.: Monadic Refinements for Relational Cost Analysis. Proc. ACM Program. Lang. **2**(POPL) (2017)
49. Sinn, M., Zuleger, F., Veith, H.: A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In: Computer Aided Verification - 26th Int. Conf. (CAV'14) (2014)
50. Stirling, J.: The Differential Method: Or, A Treatise Concerning Summation and Interpolation of Infinite Series. E. Cave (1749)
51. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Algebraic Discrete Methods **6**(2), 306–318 (1985)
52. Vasconcelos, P.B., Jost, S., Florido, M., Hammond, K.: Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In: 24th European Symposium on Programming (ESOP'15) (2015)
53. Wang, P., Wang, D., Chlipala, A.: TiML: A Functional Language for Practical Complexity Analysis with Invariants. In: Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17) (2017)
54. Wang, P., Fu, H., Goharshady, A.K., Chatterjee, K., Qin, X., Shi, W.: Cost analysis of nondeterministic probabilistic programs. In: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19). pp. 204–220 (2019)
55. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM **18**(9), 528–539 (1975)
56. iek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational Cost Analysis. In: 44th Symposium on Principles of Programming Languages (POPL'17) (2017)