Leveraging Linearity to Improve Automatic Amortized Resource Analysis

David M. Kahn

CMU-CS-24-133
July 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by: the National Science Foundation under award numbers CCF2311983, CCF1845514, CNS1801369, CCF1812876, and CCF2007784; the Algorand Foundation under award number 1031489; and HRL Laboratories, LLC under award number 17090181689USPOLINE13.

Keywords: automatic, resource, cost, linear, affine, efficient, remainder, exponential, multivariate, quantum, physicist, potential, amortized, type, functional

For my parents

Abstract

After correctness, the most important properties of programs concern their resource requirements, like how much time they take to run or how much memory they need. It is therefore desirable to automate the derivation of a program's resource requirements. One successful approach to such automatic derivation is the type system known as Automatic Amortized Resource Analysis (AARA). AARA finds polynomial bounds on resource usage by using its types to apply the physicist's method of amortized cost analysis. Type inference in AARA can be reduced to linear programming, thereby automating resource analysis. This balance of expressive bounds and efficient analysis has brought AARA success in analyzing various programs of interest.

Unfortunately, deriving a program's resource usage (i.e., costs) can be difficult—in fact it is generally not computable. Thus, despite AARA's success, it is not surprising that there are many natural program patterns that it cannot analyze well. Sometimes AARA finds loose cost bounds, other times it finds bounds slowly, and sometimes it cannot find any bounds at all.

This thesis addresses such shortcomings by developing a variety of upgrades to the AARA type system that allow the efficient derivation of tight cost bounds for more programs. The key theme underlying these upgrades is the leveraging of *linear* reasoning principles. These ideas integrate well with AARA because AARA exists in the intersection of various forms of linearity: the linear flavor of its type system, the linear relations of its cost bound templates, and the linear physicality behind the physicist's method of amortized cost analysis.

This work first upgrades the type system's infrastructure with *remainder contexts* to better reason about reusable resources like memory. Then the class of AARA's bounding functions is enlarged to include, e.g., *exponential* bounds, which can provide resource bounds for programs with multiple recursive calls. This class of functions is further enlarged to be *multivariate*, allowing dependence on products of data structure sizes, which is critical for analyzing functions with accumulators. Next, this work provides a more efficient, matrix-based approach to inferring the *cost-free* AARA types needed for, e.g., non-tail recursion. Finally the physicist's method of amortized cost analysis is refined into the *quantum physicist's method*, which provides an automatable framework for reasoning about resource reallocation, while also allowing resource bounds to depend on the height of data structures.

Each of these upgrades is proven sound with respect to an operational cost semantics, and various implementations are made to empirically evaluate their efficacy.

Acknowledgments

I would like to acknowledge my advisor for giving me the exact kind of guidance I needed and always supporting the directions I wanted to go.

I would like to acknowledge my committee for giving me invaluable insight and advice, as well as taking the time to read the things I write.

I would like to acknowledge my other mentors for helping me to decide my path.

I would like to acknowledge my family for sending me on this journey of curiosity.

I would like to acknowledge my friends for coming on this journey with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is about improving the capabilities of the automatic cost analysis tool known as Automatic Amortized Resource Analysis (AARA) by leveraging a variety of the tool's intrinsic features. What follows is an exploration of the interplay between cost reasoning, logic, and automation, all under the banner of "linearity". In this chapter, I set up the thesis statement and give the structure of the rest of the thesis.

1.1 Setup

After writing some piece of code, one of the first questions someone might ask is "How fast does it run?" Such questions about the time[1] usage of programs are central to the study of computer science—and for good reason: Programs execute in the real world, and that execution accrues real costs. In today's world, where more programs are being run than ever before, it is especially important that one can reason about a program's resource usage.

This thesis focuses on *automatic* resource analysis. Such automation is both intellectually satisfying as a computer scientist, and also perhaps practically necessary to deal with the sheer amount of code today. Unfortunately,[2] general cost analysis is the hardest kind of problem that there is: *uncomputable*[135]. Thus, every approach to cost analysis must make some tradeoff between its level of automation and its level of completeness. Approaches to cost analysis exist everywhere across this spectrum of tradeoffs. Toward the complete extreme, one finds mechanized proof frameworks requiring heavy user guidance such as *calf*[114] or implementations of separation logic with time credits[66]. Toward the automatic extreme, one finds approaches such as Metric[140], COSTA[3, 4], SPEED[69], LOOPUS[130], KoAT[21], or CHORA[20], that attempt to reduce the problem of cost analysis to some tractible domain. The work in this thesis focuses on a tool positioned toward the automatic extreme called Automatic Amortized Resource Analysis (AARA).

AARA is a type system originating from Hofmann and Jost's work on automatic derivation of heap requirements for functional progams [85]. The approach bookkeeps resource usage using the physicist's method of amortized cost analysis [133]. The physicist's method presumes that

[1]or memory, or space, or energy, etc.
[2]or perhaps fortunately for the job security of future computer scientists

data structures carry some amount of potential energy, and that energy can be freed to pay for costs. The types of AARA provide templates for how that potential energy is carried, e.g., the type $L^5(\mathbb{Z})$ indicates an integer list that holds 5 units of energy per element. Functions typed with potential energy then provide the cost information of interest: the potential energy of the function's input bounds the function's peak execution cost, and the difference in energy between the input and output bounds the net execution cost. Crucially, linear programming can be used to efficiently infer AARA types, which automates the cost analysis.

These features place AARA at an interesting intersection between many forms of *linearity*, which this thesis aims to exploit. AARA uses *linear*[3] type features because energy cannot be freely duplicated. AARA uses *linear* programming for automation. Even the physical reasoning principles introduced by the physicist's method come with linearity in the form of, e.g., quantum superposition. Perhaps AARA has found the success it has because all its different forms of linearity mesh well with one another— certainly, this thought underpins the work of this thesis.

Of course, despite AARA's success, the approach is not immune to the obstacles of uncomputability. Therefore AARA's analysis must make some tradeoffs. By favoring automation, AARA sacrifices the ability to analyze some programs well. For example, prior to the work of this thesis, AARA struggled to reuse resources like memory and could be easily counfounded by common program patterns like tree traversals.

This thesis will address these problems and others while maintaing AARA's high level of automation. In particular, this work shows how to improve AARA on multiple axes: better support for reusable resources, more expressive cost bounds, and more efficient analysis performance. And this work does so by leveraging AARA's aforementioned linearity. In a nutshell, the work can be summarized with the following thesis statement:

Thesis Statement:

1. **AARA's state-of-the-art automatic capabilities can be improved to derive tighter cost bounds more efficiently for more kinds of programs and more kinds of resource costs.**
2. **Such improvements can be made by leveraging key features of *linearity* intrinsic to the AARA type system.**

1.2 Thesis Structure

The remainder of this thesis is organized into three parts:

Part 1 lays out the formal groundwork of AARA for the rest of the thesis to build upon. This part is made up of the following chapters:

- Chapter 2 lays out the formal language and operational semantics for the programs analyzed in this thesis. This chapter is where the notion of cost is defined.
- Chapter 3 describes the state-of-the-art in AARA prior to the work of this thesis and lays out the base type system that this thesis uses. This base type system is essentially the

[3]More specifically, the type features are usually affine. However, this thesis explores extensions which bring out more fully linear features as well.

polynomial cost bounds system of [77].

- Chapter 4 goes over related work in the domain of cost analysis.

Part 2 contains the key contributions of this thesis. Each of Part 2's chapters builds upon the AARA system of the previous chapters, is structured in a problem-solution format, and includes a formal proof of the new AARA system's soundness.

- Chapter 5 presents a formal account of *remainder contexts*, which improve reasoning about reusable resources like memory in AARA. In the process, remainder contexts simplify the AARA type system and introduce some new symmetries into the typing rules. These remainder contexts are related to I/O contexts[28, 75] from linear logic proof search and uncomputation[17] from reversible computing.
- Chapter 6 explains how to generalize the kinds of cost bounds inferrable in AARA by assigning potential energy according to linear recurrences. An optimal set of such potential assignments is characterized, and a specialized system based on Stirling numbers of the second kind is set up for exponential bounds specifically. Special rules are provided to mix this system with the preexisting polynomial bound system.
- Chapter 7 delves into *multivariate*[4] exponential bounds, extending the key contribution of the Chapter 6. Such bounds are not only more expressive, but also should help to put exponentials on equal footing with polynomials. The most developed existing implementation of AARA, Resource Aware ML (RaML), uses multivariate polynomials, and the results of this chapter alongside those of Chapter 6 should allow RaML to be extended in the future to cover exponential bounds.
- Chapter 8 presents an alternative, significantly more efficient way to infer *cost-free* types in AARA by deducing linear functional transformations from matrix inequalities. This approach also enables cost-free type inference for non-polynomial bounds, particularly exponential bounds like those of Chapter 6. Cost-free types are special AARA types used to describe how excess energy is reallocated across different data structures, and they are critical for cost bound composition and non-tail recursion, among other applications.
- Chapter 9 refines the physicist's method of amortized cost analysis into the *quantum physicist's method*, which leverages similar linear structures as are present in quantum physics to reason about the physcist's method's "potential energy". A novel bookkeeping principle called *resource tunneling* is presented to reallocate resources around potential barriers, similarly to quantum tunneling. Using remainder contexts, the result is incorporated into AARA to obtain tighter cost bounds based on resource reallocation. This system is then further adapted to reason about the depth of data structures, which enables reasoning about, e.g., tree traversals.

Finally, part 3 concludes with Chapter 10.

[4]*Multivariate* in this context refers to bounds with terms that are *products* of functions of the sizes of input data structures. For example, $x \cdot y$ is multivariate, but $x + y$ is merely *univariate*. These definitions are at odds with more typical usage of the words ”univariate” and ”multivariate”, but these definitions have already been established in work prior to this thesis [80].

Chapter 2

Language

Before one can derive a program's execution costs, one must actually specify what is meant by "program", "execution", and "cost"; i.e., one must formalize the language. This chapter does so by formalizing the expressions, values, operational semantics, and cost constructs that will be used throughout this thesis. No chapter deviates from this language. I also set up a new, simplified approach for reasoning about the costs of nonterminating programs in Section 2.4.

2.1 Expressions

For the purposes of this thesis, the expressions of AARA's language can be given by the grammar of Figure 2.1, where r is a rational number and the symbols f, x are used for variables. These expressions are mostly comprised of the basic features of a functional language with algebraic types and pattern matching (casing), and the syntax used is relatively normal. However, there are still a few details of note, which I go over in the following paragraphs.

Firstly, the language contains no expressions for data like Booleans or integers, e.g., no arithmetic operations or if-then-else statements. While an implementation of AARA should probably include such expressions, they are omitted here because they do not meaningfully interact with the type system, and because they can be simulated using the provided algebraic constructors (or

$$
\begin{array}{rll}
e ::= & x \mid \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 & \text{variables} \\
 & \mid \texttt{fun}\ f\ x\ =\ e \mid f\ x & \text{functions} \\
 & \mid \langle x_1,\ x_2\rangle \mid \texttt{case}\ x_p\ \texttt{of}\ \langle x_1,\ x_2\rangle \to e & \text{products} \\
 & \mid \texttt{l}(x) \mid \texttt{r}(x) \mid (\texttt{case}\ x_s\ \texttt{of}\ \texttt{l}(x_1) \to e_1 \mid \texttt{r}(x_2) \to e_2) & \text{sums} \\
 & \mid [\,] \mid x_1 :: x_2 \mid (\texttt{case}\ x_\ell\ \texttt{of}\ [\,] \to e_1 \mid x_1 :: x_2 \to e_2) & \text{lists} \\
 & \mid \texttt{leaf} \mid \texttt{node}(x_1,\ x_2,\ x_3) \mid (\texttt{case}\ x_t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x_1,\ x_2,\ x_3) \to e_2) & \text{trees} \\
 & \mid \texttt{tick}\{r\} & \text{cost}
\end{array}
$$

Figure 2.1: Let-normal form expression grammar

$$v ::= \texttt{C}(V;\ f,\ x.\,e) \mid \langle v_1,\ v_2\rangle \mid \texttt{l}(v) \mid \texttt{r}(v) \mid [\,] \mid v_1 :: v_2 \mid \texttt{leaf} \mid \texttt{node}(v_1,\ v_2,\ v_3) \mid \langle\rangle$$

Figure 2.2: Value grammar

even simulated just as the unit value). I assume such encodings for Booleans, integers, etc. and their corresponding operations throughout this thesis.

Secondly, this language fixes lists and binary trees as the only inductive data types. This restriction makes it easier to present the technical development of this thesis. However, this restriction is not essential to AARA, as other work has shown how to extend AARA with various user-defined inductive data types [65, 82].

Next, the given expression grammar in Figure 2.1 is for a language in *let-normal form*. This normal form requires that every subexpression is a variable as much as possible. Let-normal form is very similar to A-normal form[57] except that A-normal form allows subexpressions to be both variables *and constants*; let-normal expressions do not allow such constants.

Finally, there is a special "tick" expression $\texttt{tick}\{r\}$. This expression is used to define costs. Executing this expression consumes r resources and returns the unit value $\langle\rangle$. If r is negative, the expression corresponds to r resources being created instead. Such ticks can either be inserted manually for fine-grained control of the cost model, or inserted automatically through some predefined syntactic transformation. For example, to measure the cost in terms of recursive calls for `id` in Section 3.1, one can insert "`let y = tick{1} in`" before line 5 of Figure 3.2 where the recursive call is made. Alternatively, to measure cost in terms of call stack frames[1], one adds the same expression before line 5 and also the expression "`let z = tick{-1} in`" after line 5 to respectively allocate and free the stack frame. Costs in terms of the sizes of data structures can be encoded by adding more complex auxiliary code that recurses over those data structures while ticking.

2.2 Values

The values of this language are given by the grammar of Figure 2.2. These values include pairs, sums, lists, binary trees, the unit value, and closures. The only interesting value is that for function closures, $\texttt{C}(V;\ f,\ x.\,e)$, which represents a recursive function f that has formal parameter x, has the body expression e, and has captured the value environment V.

2.3 Operational Cost Semantics

Both execution and cost are formalized via a big-step operational semantics equipped with resource counters. These big-step evaluation rules are given in Figures 2.3 and 2.4 and make use of the following evaluation judgment:

$$V \vdash e \Downarrow v \mid (p, q)$$

[1]without tail-recursion optimization

E-VAR

$$\frac{}{V, x \mapsto v \vdash x \Downarrow v \mid (0,0)}$$

E-LET

$$\frac{V \vdash e_1 \Downarrow v' \mid (p,q) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r,s)}{V \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v \mid (p + \max(0, r-q), s + \max(0, q-r))}$$

E-FUN

$$\frac{}{V \vdash \texttt{fun}\ f\ x\ =\ e \Downarrow \texttt{C}(V;\ f,\ x.\, e) \mid (0,0)}$$

E-APP

$$\frac{V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (p,q)}{V, x \mapsto v', f \mapsto \texttt{C}(V';\ g,\ y.\, e) \vdash f\ x \Downarrow v \mid (p,q)}$$

E-PAIR

$$\frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash \langle x,\ y\rangle \Downarrow \langle v_1,\ v_2\rangle \mid (0,0)}$$

E-CASEP

$$\frac{V, x \mapsto \langle v_1,\ v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p,q)}{V, x \mapsto \langle v_1,\ v_2\rangle \vdash \texttt{case}\ x\ \texttt{of}\ \langle y,\ z\rangle \to e \Downarrow v \mid (p,q)}$$

E-SUML

$$\frac{}{V, x \mapsto v \vdash \texttt{l}(x) \Downarrow \texttt{l}(v) \mid (0,0)}$$

E-CASES-L

$$\frac{V, x \mapsto \texttt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p,q)}{V, x_s \mapsto \texttt{l}(v') \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 \Downarrow v \mid (p,q)}$$

E-SUMR

$$\frac{}{V, x \mapsto v \vdash \texttt{r}(x) \Downarrow \texttt{r}(v) \mid (0,0)}$$

E-CASES-R

$$\frac{V, x \mapsto \texttt{r}(v'), z \mapsto v' \vdash e_2 \Downarrow v \mid (p,q)}{V, x_s \mapsto \texttt{r}(v') \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 \Downarrow v \mid (p,q)}$$

Figure 2.3: Big-step cost evaluation rules 1

E-NIL
$$\frac{}{V \vdash [\,] \Downarrow [\,] \mid (0,0)}$$

E-CONS
$$\frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

E-CASEL-NIL
$$\frac{V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)}{V, x \mapsto [\,] \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

E-CASEL-CONS
$$\frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p,q)}{V, x \mapsto v_1 :: v_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

E-LEAF
$$\frac{}{V \vdash \texttt{leaf} \Downarrow \texttt{leaf} \mid (0,0)}$$

E-NODE
$$\frac{}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\ y,\ z) \Downarrow \texttt{node}(v_1,\ v_2,\ v_3) \mid (0,0)}$$

E-CASET-LEAF
$$\frac{V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p,q)}{V, t \mapsto \texttt{leaf} \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p,q)}$$

E-CASET-NODE
$$\frac{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p,q)}{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3) \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p,q)}$$

E-TICK
$$\frac{}{V \vdash \texttt{tick}\{r\} \Downarrow \langle\rangle \mid (\max(0,r), \max(0,-r))}$$

E-SHARE
$$\frac{V, x_1 \mapsto v', x_2 \mapsto v', x_3 \mapsto v' \vdash e \Downarrow v \mid (p,q)}{V, x_1 \mapsto v' \vdash \texttt{share}\ x_1\ \texttt{as}\ x_2,\ x_3\ \texttt{in}\ e \Downarrow v \mid (p,q)}$$

Figure 2.4: Big-step cost evaluation rules 2

This judgment means that, given the value environment V:

- the expression e evaluates to v
- the evaluation of e requires $p \geq 0$ resources to run
- the evaluation of e leaves $q \geq 0$ resources leftover

I call the pair (p, q) the *cost behaviour* for simplicity. The latter two bullet points in the evaluation judgment description explain how cost behaviours capture the notion of execution costs: The value p is the peak cost,[2] and the value of $p - q$ is the net cost. These are the costs that AARA aims to bound using its potential method bookkeeping.

The actual evaluation dynamics expressed by the rules of Figures 2.3 and 2.4 are relatively standard for an eager functional language. Most rules also do not meaningfully interact with resources, with the exceptions of the rules *E-Tick* and *E-Let*. I explain the operation of each of these in the subsequent paragraphs.

The rule *E-Tick* is the only rule that directly alters the amount of resources. This rule uses $\max$ to encode the cost behaviour described in Section 2.1, where positive values of r encode consumption of resources, and negative values of r encode the release of resources. When $r \geq 0$, the cost behaviour reduces to to $(r, 0)$, indicating that r resources are consumed to evaluate $\texttt{tick}\{r\}$. Otherwise the cost behaviour reduces to $(0, |r|)$, indicating that the evaluation of $\texttt{tick}\{r\}$ requires no resource payment and in fact leaves $|r|$ additional resources available. Such giving of additional resources would occur for a resource like money where change is given back or a resource like memory which can be freed and reused.

The rule *E-Let* is the only rule that otherwise interacts with resources. All this rule does cost-wise is use $\max$ to compose the cost behaviours of two expressions e_1 and e_2. Intuitively, this composition reuses the q leftover resources from evaluating e_1 for the evaluation of e_2. There are two cases to consider depending on whether those q leftover resources cover the peak cost r of evaluating e_2. When $q \geq r$ so that the peak cost is fully covered, the cost behaviour reduces to $(p, q - (r - s))$, which corresponds to paying e_2's net cost $r - s$ out of the q leftover resources from evaluating e_1. For example, a cost behaviour of $(3, 10)$ followed by $(5, 1)$ is represented by the cost behaviour of $(3, 6)$, where the net cost of $5 - 1 = 4$ is taken out of the leftover 10. Otherwise, when $q < r$, the cost behaviour reduces to $(p + (r - q), s)$, which corresponds to evaluating e_1 with $r - q$ extra resources available so that afterward exactly r resources are available to cover the peak cost of evaluating e_2. For example, a cost behavour of $(5, 1)$ followed by $(3, 10)$ is represented by the cost behaviour of $(7, 10)$, where $3 - 1 = 2$ additional resources are added to the starting amount of 5 to leave enough for the second cost behaviour.

2.4 Nontermination

So far, using the rules of Figures 2.3 and 2.4, one can only define the evaluation and cost of a terminating program execution. However, *non*terminating programs can still have interesting cost bounds. While net costs are only sensible for terminating computations, one can still meaningfully consider the *peak* cost of a nonterminating computation. For example, it could be useful to ensure that some indefinitely running computational service will only ever require finite memory.

[2] or "high-water mark" of cost, as described by other sources

$$\frac{}{V \vdash e \Downarrow \bullet \mid (0, \infty)}\;\text{E-NONT}$$

Figure 2.5: Nontermination evaluation rule

The only way that previous work on AARA has handled nontermination is by defining a set of *partial* big-step evaluation rules to augment the (total) evaluation rules [78].[3] This setup gives every expression in the language multiple ways to evaluate: either do so totally as in Figures 2.3 and 2.4, or abort evaluation partway through and record the peak cost. As a result, every finite prefix of a computation can be given an evaluation judgment assigning a peak cost. A uniform bound on the peak cost of every finite prefix of a computation then yields a bound on the peak cost of the entire computation, even if that computation is nonterminating.

While partial evaluation rules do indeed allow for reasoning about the costs of nonterminating executions, the existing literature does so in a rather verbose way: providing a new partial evaluation rule (sometimes more) for most expressions of the language [78]. As a result, proving statements about the total *and* partial evaluations of a program requires roughly double the proof cases of proving statements *only* about the total evaluations. I would like to avoid such long proofs.

To avoid ballooning proof obligations, I introduce the following simplification for the work of this thesis: Instead of adding a new nontermination rule for each expression, I introduce the *single* new rule given in Figure 2.5 alongside a special new dummy value $\bullet$. The rule *E-Nont* takes the form a *total* evaluation rule and simulates aborting evaluation by returning the value $\bullet$ and leaving infinitely many resources leftover. This infinity absorbs future costs, thus rendering the cost behaviour dependent only on computation up to the abort.

One way to see why this infinity is the right choice is to treat the leftover count like a ghost variable in a Hoare logic postcondition [74]. The leftover count is usually supposed to be measured *after the total evaluation of* e, but aborting evaluation means e does not totally evaluate, rendering the postcondition false. Therefore, the condition in which one considers leftover resources is *false*, and falsum implies anything. Having infinitely many resources leftover is an equivalently strong condition.

Because my introduction of dummy values allow the rule *E-Nont* to take the form of a total evaluation rule, it seamlessly integrates with the existing total evaluation rules of Figures 2.3 and 2.4. In particular, no other nontermination rules are needed— the rule *E-Nont* naturally induces the correct behaviour through the other terminating rules. One simply interprets a cost behaviour of the form (p, ∞) as coming from an evaluation that achieves a peak cost of p before later aborting. I formalize this notion in Lemma 2.4.2, but for now consider the example of how the cost behaviour of $(0, \infty)$ interacts in the evaluation rule *E-Let*. When evaluating the

[3] In principle, small-step semantics would also suffice for reasoning about nontermination. However, such an approach had not been used with AARA by the start of my thesis work. It is not completely trivial to compare AARA against small-step semantics as such comparison raises questions concerning the potential energy of unevaluated expressions.

expression `let` $x = e_1$ `in` e_2, if evaluation immediately aborts while evaluating e_1, before e_2 should be evaluated, then the cost behaviour of the whole expression should be $(0, \infty)$. The rule *E-Let* gives exactly this result no matter the cost behaviour of e_2, successfully ignoring computation after aborting.[4] Alternatively, if e_1 evaluates with cost behaviour (p, q), and then evaluation is aborted at the start of e_2, then the cost behaviour of the whole expression should be (p, ∞). The rule *E-Let* gives exactly this result, and thus the peak cost behaviour of e_1 is successfully recorded.

If the evaluation of e_1 is aborted during the evaluation of the expression `let` $x = e_1$ `in` e_2, one might be concerned about the presence of the dummy value bound to x when attempting to derive an evaluation judgment for e_2. For instance, no evaluation rules cover pattern matching a dummy value $\bullet$. However, there always exists at least one cost behaviour assignable to e_2 via the rule *E-Nont*, and it would not matter if any other cost behaviour was assignable to e_2 because it would be ignored as described previously. Without loss of generality, one could therefore assume that e_2 is always evaluated using *E-Nont* whenever e_1 is. Thus, the dummy value causes no problems for the evaluation rules.

One might also be concerned about arithmetic using ∞. While there are no problems with statements like $\infty \geq q$ and $\infty + q = \infty + \infty = \infty$ (where $q \in \mathbb{Q}$), indeterminate expressions like $\infty - \infty$ could potentially be problematic. It turns out, however, that such problems can be avoided, so the evaluation judgment is well-defined. One proves this property by simultaneously proving a fact about the restricted form of cost behaviours: in cost behaviours (p, q), the value of p is not ∞. The well-definedness of the evaluation judgement is formalized in Lemma 2.4.1.

Lemma 2.4.1 (well-definedness and form of evaluation cost)**.** *The evaluation judgment given by $V \vdash e \Downarrow v \mid (p, q)$ is well-defined. That is, even though the rule E-Nont introduces ∞, the evaluation rules never produce indeterminate arithmetic expressions like $\infty - \infty$. Moreover, in such judgments, $p \neq \infty$.*

Proof. The well-definedness and inequality statements are proved simultaneously by induction over the derivation of the judgment $V \vdash e \Downarrow v \mid (p, q)$. Every case of the induction is trivial except for the rule *E-Let*, where the inductive hypothesis that $p \neq \infty$ ensures that the rule's subtraction is well-defined. □

Now that the arithmetic has been confirmed to be well-defined, one can formally capture the intuition that cost behaviours of the form (p, ∞) correspond precisely to aborting computation.

Lemma 2.4.2 (cost behaviour of abort)**.** *Let $\mathcal{D}$ be a derivation of $V \vdash e \Downarrow v \mid (p, q)$. Then $\mathcal{D}$ uses E-Nont iff $q = \infty$.*

Proof. The backward implication is trivial because only the rule *E-Nont* introduces ∞.

[4]Put another way, each cost behaviour (p, ∞) is a left annihilator in the "resource monoid" sometimes used for defining AARA's operational semantics [76]. This monoid's notion of multiplication is the *E-Let* rule's cost behaviour composition, and $(p, \infty) \cdot (q, r) = (p, \infty)$.

The forward implication holds via induction. Every case is trivial except for *E-Let*:

$$\textsc{E-Let}\quad \frac{V \vdash e_1 \Downarrow v' \mid (p,q) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r,s)}{V \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v \mid (p + \max(0, r-q), s + \max(0, q-r))}$$

For the derivation $\mathcal{D}$ to contain the rule *E-Nont*, one of the derivations for its premisses must. Then one of s or q is ∞ by the inductive hypothesis. Because Lemma 2.4.1 ensures $r \neq 0$, it follows that $s + \max(0, q - r) = \infty$, completing the case. □

I complete the formal treatment of the new dummy value $\bullet$ in Section 3.6 so that AARA can be proved sound with respect to both termination and nontermination in a succinct, uniform manner throughout this thesis.

Chapter 3

Automatic Amortized Resource Analysis

This chapter describes the type system of Automatic Amortized Resource Analysis (AARA). While many extensions of AARA exist in the literature, I fix one version here to provide this thesis with a clear base to build upon. This base version is essentially the polynomial cost bound system of Hoffmann and Hofmann[77] but with uniform notation that will be used throughout this thesis. To keep matters simple, I do not set up this system to use polymorphic types. I also include a few extra components to simplify reasoning about the costs of nonterminating computations, as set up in Chapter 2 —this is the only part of this section that is at all novel. Section 3.8 concludes by explicitly highlighting some of the ways that linearity plays a role in AARA, which will help prime the reader for future chapters.

3.1 Overview

This section gives an overview of the Automatic Amortized Resource Analysis type system and its workings. Following sections formalize the ideas of this section and lead up to a formal statement of AARA's soundness.

Introducing AARA

AARA is a type system for the automatic, static derivation of cost bounds. These cost bounds are expressed as concrete (non-asymptotic) functions of the sizes of a program's data structures, and they can be in terms of various user-defined costs models, including time, space, energy, and money. No matter the chosen cost model, a soundness proof guarantees that AARA's cost bounds are accurate, so that type derivations are cost bound certificates. AARA type inference is reducible to linear programming, which allows such cost bounds certificates to be found efficiently. And through the composition of function types, these bounds can be composed, allowing for a modular cost analysis.

Hofmann and Jost originally developed AARA for the automatic derivation of worst-case, linear heap space requirements for first-order functional programs [85]. Later work (including some of the work of this thesis) has extended that original work in a variety of ways. Some work has extended AARA to reason about other resources[93] and non-linear cost bounds such as polynomials[77, 80], exponentials[95], and logarithms[89]. Other work has derived lower cost bounds[51] and cost bounds based on various features of data types [26, 65, 82, 96]. Still other work has adapted the approach to other programming domains, including higher-order polymorphic programs[94], probabilistic programs[112, 138], parallel programs[79], imperative programs[27], object-oriented programs[86], smart contracts[45, 46, 47], and CUDA kernels[109].

Physicist's Method

AARA performs its cost analysis by relying on the physicist's method of amortized analysis [133]. This method posits that data structures hold some amount of potential energy. When a data structure is destructed, this energy is freed and can be used to pay for costs. Similarly, when a data structure is constructed, it can store more energy away for later. AARAs types dictate just how much energy is stored on each data structure as a function of that data structure's size. For example, the type $L^5(\mathbb{Z})$ is the type of an integer list holding 5 units of potential energy per element. The key to the physicist's method is that the amount of energy initially present is an upper bound on the *peak* execution cost, and the difference in energy between the initial and final program state is an upper bound on the *net* execution cost. These bounds are really just reformulations of the law of conservation of energy from the first law of thermodynamics,[1] hence the name "phycisist's method".

One can see the power of the physicist's method by examinining AARA's function types. By interpreting function types using the physicist's method, one can obtain bounds on both the peak and net cost of running that function. The potential energy of the function argument gives the peak cost bound and the difference between the energy of the argument and return gives the net cost bound. Thus, a function type like $L^5(\mathbb{Z}) \to \mathbb{Z}$ represents a linear net execution cost bound—specifically a bound of $5n$ where n is the length of the input list. Crucially, functions can be given different types to specialize to the amount of energy initially present, which is key to AARA's compositionality.

Because AARA uses the physicist's method for reasoning about cost, the cost bounds it provides are naturally amortized. For example, consider the 2-stack queue, which is a classic target of amortized analysis that implements a queue using two stacks (lists). Code implementing the interface to such a 2-stack queue can be found in Figure 3.1. If the function `reverse` spends 1 unit of time per element of the input list, then a classic amortized analysis result is that `dequeue` runs in *amortized* constant time. This result is proven by storing 1 time prepayment with every `enqueue` operation. Because there is 1 prepayment stored per element of `in_stack`, the prepayment covers the eventual costly call to `reverse`, leaving only a constant marginal time cost. When analyzed with AARA, the type assigned to `in_stack` is indeed of the form $L^1(\tau)$, which matches the 1-per-element prepayment.

[1]To spell out this reformulation in more detail: treat the potential energy of a program state be the energy of a physical system, and treat the program cost as some amount of useful work to accomplish using the system. The first law of thermodynamics states that the change in the system's energy is equal to the useful work done plus some amount of energy that is lost due to heat. Thus the system's change in energy is in general an upper bound on the work done. In the programmatic setting, the inevitable heat loss seems to correspond to the inevitable imprecision arising from the uncomputability of general cost analysis.

```
fun enqueue (q, x) = case q of
  | (in_stack, out_stack) -> (x::in_stack, out_stack)

fun dequeue q = case q of
  | ([], []) -> None
  | (in_stack, x::out_stack) -> Some (x, (in_stack, out_stack))
  | (in_stack, []) -> dequeue ([], reverse in_stack)
```

Figure 3.1: Code for the 2-stack queue

Polynomial Bounds

To represent more interesting cost bounds than the linear bounds used thus far, AARA just needs a way to express potential energy as more interesting functions of the size of its data structures. That is, AARA must represent a more interesting set of *resource functions*.

The base system used in this thesis represents polynomial resource functions in the following way: First, AARA adds additional annotations to the types of its data structures. These additional annotations can be grouped as an annotation *vector*, like $\vec{a}$ in the type $L^{\vec{a}}(\mathbb{Z})$. Then, AARA interprets the vector to give the coefficients of a polynomial, i.e., to give a linear combination of basis polynomials. However, these basis polynomials are *not* the standard linear basis for polynomial functions $\lambda n.\, n^k$— It turns out to be more convenient to use binomial coefficients $\lambda n.\, \binom{n}{k}$ for reasons discussed in the next paragraph. Thus the energy on a list typed $L^{6,3}(\mathbb{Z})$ is $6 \cdot \binom{n}{2} + 3 \cdot \binom{n}{1} = 3 \cdot n^2$, where n is the length of the list.[2] Because $\lambda n.\, \binom{n}{k} \in \Theta(\lambda n.\, n^k)$, I refer to $\binom{n}{k}$ potential energy as *degree-k* potential energy, e.g., $L^{6,3}(\mathbb{Z})$ indicates 6 units of quadratic (degree-2) energy and 3 units of linear (degree-1) energy.

Binomial coefficients are useful for representing polynomial resource functions for a variety of reasons:

- The binomial coefficent $\binom{n}{k}$ counts the number of ways to pick k objects out of n without replacement. This combinatorial interpretation allows one to, e.g., treat the base-k energy of a list of length n as counting the number of ordered k-tuples of list elements. Thus, the potential energy can be given a natural semantic meaning.
- The value of $\binom{0}{k}$ is 0 for $k \geq 1$, so no potential energy assigned to empty data structures.
- The function $\lambda n.\, \binom{n}{k}$ is *nonnegative* for $n \geq 0$, so nonnegative scalars of this function yield nonnegative amounts of energy. This circumstance makes it simple to ensure energy is nonnegative without needing to concretely know the size of the data structure n. Negative amounts of energy are to be avoided because they ruin physicist's method reasoning about peak costs—more energy than the initial energy can be spent if negative energy is allowed.
- While both $\lambda n.\, n^k$ and $\lambda n.\, \binom{n}{k}$ can generate all polynomials with their linear combinations, they have distinct *conical* combinations (i.e., linear combinations over nonnegative coefficients). In particular, the conical combinations of $\lambda n.\, n^k$ are strictly included within the conical combinations of $\lambda n.\, \binom{n}{k}$. As a result, $\lambda n.\, \binom{n}{k}$ is a more expressive function basis.

[2]In some other AARA literature, the order of annotations is reversed.

```
fun id lst =                (* lst : 1 per element,     0 free *)
  case lst of
  | [] -> []                (* return : 0 per element, 0 free *)
  | x::xs ->                (* xs : 1 per element,      1 free *)
    let tmp = id xs in      (* tmp : 0 per element,     0 free *)
    x::tmp                  (* return : 0 per element, 0 free *)
```

Figure 3.2: Code for `id` with concrete energy comments

- Using binomial coefficients, there is an easy way to redistribute energy from a list of size $n+1$ to a list of size n and vice versa. This circumstance is because binomial coefficients follow the linear recurrence of Pascal's identity: $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$. It is useful to have such a way to redistribute energy as data structures of varying sizes are constructed and destructed. For example, Pascal's identity tells us that $6\cdot\binom{n+1}{2}+3\cdot\binom{n+1}{1} = 6\cdot\binom{n}{2}+9\cdot\binom{n}{1}+3$, so if the list `x::xs`: $L^{6,3}(\mathbb{Z})$ is destructed, then AARA can conserve energy by leaving 3 units of energy free and typing `xs` as $L^{6,9}(\mathbb{Z})$.

More generally, one of the key benefits of representing cost bounds with resource functions is that AARA does not need to reason deeply about the sizes of data structures. In contrast, many other approaches to cost analysis, like solving recurrence relations, depend on size analyses. In AARA, each resource function acts as an energy density function by describing a distribution of resources over a data structure. That distribution is constant for linear resource functions, linearly increasing for quadratic resource functions, etc. As long as AARA can reason appropriately in terms of these densities, a direct size analysis is never required. Intuitively, such reasoning works as follows: energy density is total energy over size, so doubling size while total energy remains unchanged results in halving energy density. Such reasoning works no matter the size involved. For the polynomial case given here, the correct density reasoning is given by Pascal's identity, which holds for all n.

Using the setup described here, AARA can find so-called *univariate* polynomial cost bounds. A univariate polynomial in this setting is a polynomial where each term is a function of at most one unknown. Thus, despite having two variables, $x+y$ is univariate. Univariate bounds like $x+y$ can arise because each inductive data structure contributes potential energy based only on its own size, but multiple such data structures can exist in a piece of code. Most of this thesis will only consider univariate bounds, but I do consider *multivariate* bounds in Chapter 7. In multivariate expressions, terms can be functions of the sizes of multiple data structures, like $x \cdot y$.

The Feel of AARA

To get a more concrete feel of how AARA's energy bookkeeping works, consider the code for `id` in Figure 3.2. This code implements the identity function on lists by fully destructing and then reconstructing the input list. Because the code recurses over the entire input list, one should expect it to make 1 recursive call per element of the input list. By assigning 1 unit of cost to every recursive call, AARA can represent a bound on the number of recursive calls using a type like $L^1(\mathbb{Z}) \to L^0(\mathbb{Z})$, which perfectly matches the desired cost of 1 recursive call per element

```
1  fun id lst =            (* lst : a per element,     0 free *)
2    case lst of
3    | [] -> []            (* return : b per element, 0 free *)
4    | x::xs ->            (* xs : a per element,      a free *)
5      let tmp = id xs in  (* tmp : b per element,     a-1 free *)
6      x::tmp              (* return : b per element, a-b-1 free *)
```

Figure 3.3: Code for `id` with symbolic energy comments

$$\tau, \sigma ::= \mathbb{1} \mid \tau \otimes \sigma \mid \tau \oplus \sigma \mid L(\tau) \mid T(\tau) \mid \tau \overset{\vec{a}|\vec{b}}{\to} \sigma$$

Figure 3.4: AARA unannotated type grammar

of the input list. Figure 3.2 includes comments showing the bookkeeping that justifies this type. Note how in line 4, the input list `lst` is divided into a head element `x` and tail list `xs`, which frees the unit of energy stored with the head. That free energy then pays for the recursive call in line 5.

Of course $L^1(\mathbb{Z}) \to L^0(\mathbb{Z})$ is not the only type one could have assigned to `id`. If one instead were to consider the *symbolic* function type template $L^a(\mathbb{Z}) \to L^b(\mathbb{Z})$, one should expect that any choices of a and b would work if $a - 1 \geq b \geq 0$. One can symbolically justify such a type as indicated by the comments of Figure 3.3, where the condition $a - 1 \geq b \geq 0$ simply ensures that all amounts of energy are nonnegative.

Having access to multiple function types allows AARA to assign types to compositions like `id` $\circ$ `id` by assigning the left instance of `id` the type $L^1(\mathbb{Z}) \to L^0(\mathbb{Z})$ and the right instance the type $L^2(\mathbb{Z}) \to L^1(\mathbb{Z})$. Then function type composition directly yields the type $L^2(\mathbb{Z}) \to L^0(\mathbb{Z})$, which represents the bound of twice the length of the input list. This bound is exactly the cost one should expect for applying the function `id` twice.

The symbolic energy bookkeeping of Figure 3.3 also suggests how AARA type inference is automated. The only symbolic expressions are linear combinations of a, b, and 1; and the condition $a - 1 \geq b \geq 0$ can be recovered by ensuring that every amount of symbolic energy is nonnegative. Because only linear expressions and inequalities arise, a linear program can efficiently find the best choices of a and b. Because Pascal's identity is a (2-dimensional) linear recurrence, this same approach also works with polynomial bounds.

3.2 Type System

This subsection presents the formal polynomial AARA type system. This system is composed of the AARA types and typing rules, as well as an index system for the annotations.

$$Ind(\mathbb{1}) = \emptyset \qquad Ind(\tau \xrightarrow{\vec{a}|\vec{b}} \sigma) = \emptyset \qquad Ind(\tau \otimes \sigma) = \mathtt{1^{st}}.Ind(\tau) \cup \mathtt{2^{nd}}.Ind(\sigma)$$

$$Ind(\tau \oplus \sigma) = \mathtt{l}.Ind(\tau) \cup \mathtt{r}.Ind(\sigma) \qquad Ind(L(\tau)) = \{\mathtt{d}_k \mid 1 \leq k \leq D_{max}\} \cup \mathtt{e}.Ind(\tau)$$

$$Ind(T(\tau)) = \{\mathtt{d}'_k \mid 1 \leq k \leq D_{max}\} \cup \mathtt{e}'.Ind(\tau) \qquad Ind(\Gamma) = \{\mathtt{c}\} \cup \bigcup_{x \in \mathtt{dom}(\Gamma)} x.Ind(\Gamma(x))$$

Figure 3.5: AARA annotation indices

3.2.1 Types

Thus far, this work has referred to AARA types like $L^5(\mathbb{Z})$ with the annotation 5 representing a list carrying 5 units of potential per element. While this notation will continue to be used in less formal prose, it is more convenient in the type system formalization to separate types from their annotations, at least for non-function types. In particular, this separation will make later chapters of this thesis much easier to implement. For this reason, I give *unannotated* types of AARA by the grammar of Figure 3.4, where $L(-)$ and $T(-)$ are the list and tree type constructors, respectively. Note that unannotated function types *do* still carry annotations, represented by the vectors $\vec{a}$ and $\vec{b}$. I describe how this annotation system works in the following subsection.

Additionally, to keep matters simple, this work does not use polymorphic types. AARA has no problem handling polymorphic types[94], but they play no role in the contributions of this thesis and complicate some technical details in type inference.

3.2.2 Annotation Indices

Annotations are associated to types via a special index system. An index is a string of symbols concatenated by the symbol ".", and this string is essentially a pathname for a particular location that an annotation could appear on a type.[3] To annotate a type, one simply provides a map from the type's indices to the rationals that should annotate the type at those indices. Throughout this thesis, such maps are usually represented by vectors of rationals with the appropriate indices, where subscripting is application.

The indices of a type τ are given by $Ind(\tau)$ as defined in Figure 3.5, which also extends the notation over type contexts Γ. To simplify notation, Figure 3.5 uses the convention that operations like concatenation distribute pointwise over sets. The components of these indices pick out the following locations for annotations:

- $\mathtt{1^{st}}, \mathtt{2^{nd}}$ — locations on the first or second type in a product, respectively
- $\mathtt{l}, \mathtt{r}$ — locations on the left or right member of a sum, respectively
- $\mathtt{d}_k, \mathtt{d}'_k$ — locations of the degree-k potential annotations of lists and trees, respectively, up to some pre-determined maximum degree D_{max}.

[3]This pathname addressing is comparable to that of SNAX [52]. Such SNAX addresses indicate the data layout of type contexts in derivations for a similar sort of let-normal form as AARA.

- $\mathtt{e}, \mathtt{e}'$ — locations on the elements contained in lists and trees, respectively
- x — locations on the type associated to the variable x
- $\mathtt{c}$ — the location of the free energy for a type context that is used to pay for costs

Using this index system, a variable x of annotated type $L^5(\mathbb{1})$ can be equivalently typed using the unannotated type $L(\mathbb{1})$ and annotation mapping $x.\mathtt{d}_{\mathit{1}} \mapsto 5$.

I treat the annotation vectors $\vec{a}, \vec{b}$ in the type $\tau \overset{\vec{a}|\vec{b}}{\to} \sigma$ as if they were maps annotating type contexts for the function's argument and return, respectively. To this end, I introduce two special variable names: `arg`, which represents the otherwise-anonymous function argument, and `ret`, which represents the otherwise-anonymous function return. Thus, $\vec{a}$ is a vector of rationals indexed by $\{\mathtt{c}\} \cup \mathtt{arg}.\mathit{Ind}(\tau)$, and $\vec{b}$ is a vector of rationals indexed by $\{\mathtt{c}\} \cup \mathtt{ret}.\mathit{Ind}(\sigma)$.

Because these full function types include free energy annotations, additional notation is required to be able to conveniently write out full function types in prose. For this purpose, an annotation map with nonzero free energy like $\mathtt{c} \mapsto 2, \mathtt{arg} \cdot \mathtt{d}_{\mathit{1}} \mapsto 7$ is associated to a unit list $L(\mathbb{1})$ using the notation $\langle L^7(\mathbb{1});\, 2\rangle$. An identity function type might then be represented by $\langle L^7(\mathbb{1});\, 2\rangle \to \langle L^7(\mathbb{1});\, 2\rangle$. This function type takes a list carrying 7 units of energy per element and 2 units of free energy as input and then returns the same as output.

3.2.3 Typing Rules

The type system is given by its structural rules[4] in Figure 3.6 and its remaining rules in Figure 3.7. These rules use the following typing judgment:

$$\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$$

This judgment means that, in type context Γ annotated by $\vec{a} \geq 0$, the expression e is typed τ, where both τ and the leftover energy are annotated by $\vec{b} \geq 0$. To be explicit, $\vec{a}$ is indexed by $\mathit{Ind}(\Gamma)$ and $\vec{b}$ is indexed by $\{\mathtt{c}\} \cup \mathtt{ret}.\mathit{Ind}(\tau)$.

The typing rules use a number of notational conventions for annotation maps to keep the rules succinct: Comparisons between annotation maps are all pointwise. Annotation maps (with disjoint indices) are combined using a comma. The domains of annotation maps are kept implicit, though they must comport with the domains specified by the typing judgment. Finally, a special substitution over indices may be used on annotation maps, as if manipulating the keys in a map represented by key-value pairs.

The structural typing rules of the AARA type system include weakening (*T-Weak*) and contraction (*T-Contract*). The presence of such rules would appear to make the type system expressly *non*linear, but the contraction rule is somewhat special. While this rule does essentially duplicate the variable x as x_1 and x_2, it does so via providing new variable bindings x_1, x_2. As a result, no variable is used more than once, so the type system is affine, despite allowing contraction in spirit. At the same time, the contraction rule does *not* duplicate the potential energy held by x. Instead, AARA uses a form of contraction called "sharing", which splits the energy of x between

[4]To keep matters simple, I exclude some more complicated structural rules that are present in other AARA literature. These rules include subtyping for functions and a "relax" rule that allows additional free energy to be passed through turnstiles. However, such rules could be adapted here without issue.

$$\textsc{T-Weak}\quad \frac{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}{\Gamma, x : \sigma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

$$\textsc{T-Contract}\quad \frac{\Gamma, x_1 : \sigma, x_2 : \sigma \mid \vec{a} \vdash e : \tau \mid \vec{b}}{\Gamma, x : \sigma \mid \curlyvee_{x}^{x_1,x_2}(\vec{a}) \vdash [x/x_1,\, x/x_2]e : \tau \mid \vec{b}}$$

$$\textsc{T-Sub}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a} \geq \vec{a'} \qquad \vec{b} \leq \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

Figure 3.6: AARA structural typing rules

x_1 and x_2. This is formalized by the sharing operator $\curlyvee_{-}^{-,-}$ defined in Definition 3.2.1. Because this operator splits energy instead of duplicating it, potential energy is treated affinely. Some other AARA literature introduces a special sharing expression through a syntactic transformation to make contraction syntax-directed, but this rule accomplishes the same purpose just as well.

Definition 3.2.1 (sharing). *The sharing operator $\curlyvee_{z}^{x,y}$ combines the annotations of indices prefixed by x and y pointwise into new annotation bindings for the indices prefixed by z (which requires that the indices prefixed x, y, and z match).*[5] *This destroys the original bindings for x and y. Formally, given a map a from indices to rationals, sharing is defined by:*

$$\curlyvee_{z}^{x,y}(a) = \lambda i.\ \begin{cases} a(x.j) + a(y.j) & i = z.j \\ a(i) & \textit{otherwise} \end{cases}$$

The remaining structural typing rule, *T-Sub* concerns subtyping. The intuition behind subtyping is that a type like $L^5(\tau)$ is a subtype of $L^3(\tau)$ because anywhere that 3 potential energy per element is enough to cover costs, 5 potential energy will also be enough. In effect, this leaves subtyping as a form of weakening for potential energy. One could include additional subtyping rules for functions, but such rules are not so important and complicate matters beyond what are necessary for this thesis.

Finally it is time for the meat of the AARA type system: the typing rules of Figure 3.7. The majority of these rules simply relabel annotations in an unsurprising way. For instance, when a pair is typed by *T-Pair*, the indices of the pair elements are reassigned to the appropriate pair indices. However, a few rules have some interesting components that I discuss in the following paragraphs.

The rule *T-Tick* is the rule that actually accounts for paying costs. This rule takes r units of energy out of the pool of free energy annotated at index c.

The rules for lists and trees make use of a special "shifting" operator $\lhd$. This operator is defined in Definition 3.2.2. This shifting is the mechanism by which AARA reasons about polynomial costs. The pattern matching rules *T-CaseL, T-CaseT* each simulate Pascal's identity by adding the annotation degree-(k+1) potential energy to the existing annotation of the degree-k

T-Var

$$\frac{}{x : \tau \mid \vec{a} \vdash x : \tau \mid [\mathtt{ret}/x]\vec{a}}$$

T-Let

$$\frac{\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{d} \qquad \Delta, x : \sigma \mid \vec{b}, [x/\mathtt{ret}]\vec{d} \vdash e_2 : \tau \mid \vec{c}}{\Gamma, \Delta \mid \vec{a}, \vec{b} \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau \mid \vec{c}}$$

T-Fun

$$\frac{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid 0 \cdot \vec{a}, [x/\mathtt{arg}]\vec{c} \vdash e : \sigma \mid \vec{d}}{\Gamma \mid \vec{a}, \vec{b} \vdash \mathtt{fun}\ f\ x = e : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid \vec{a}, \vec{b}}$$

T-App

$$\frac{}{x : \tau, f : \tau \xrightarrow{\vec{a}|\vec{b}} \sigma \mid [x/\mathtt{arg}]\vec{a} \vdash f\ x : \sigma \mid \vec{b}}$$

T-Pair

$$\frac{}{x : \tau, y : \sigma \mid \vec{a} \vdash \langle x,\ y \rangle : \tau \otimes \sigma \mid [\mathtt{ret.1^{st}}/x,\ \mathtt{ret.2^{nd}}/y]\vec{a}}$$

T-CaseP

$$\frac{\Gamma, y : \sigma, z : \rho \mid [y/x.1^{\mathtt{st}},\ z/x.2^{\mathtt{nd}}]\vec{a} \vdash e : \tau \mid \vec{b}}{\Gamma, x : \sigma \otimes \rho \mid \vec{a} \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\ z \rangle \to e : \tau \mid \vec{b}}$$

T-Tick

$$\frac{\vec{a}_{\mathsf{c}} = \vec{b}_{\mathsf{c}} + r}{\cdot \mid \vec{a} \vdash \mathtt{tick}\{r\} : \mathbb{1} \mid \vec{b}}$$

T-SumL

$$\frac{}{x : \tau \mid \vec{a} \vdash \mathtt{l}(x) : \tau \oplus \sigma \mid [\mathtt{ret.l}/x]\vec{a}, \vec{b}}$$

T-SumR

$$\frac{}{x : \sigma \mid \vec{a} \vdash \mathtt{r}(x) : \tau \oplus \sigma \mid [\mathtt{ret.r}/x]\vec{a}, \vec{b}}$$

T-CaseS

$$\frac{\Gamma, y : \sigma \mid \vec{a}, [y/x.\mathtt{l}]\vec{b} \vdash e_1 : \tau \mid \vec{d} \qquad \Gamma, z : \rho \mid \vec{a}, [z/x.\mathtt{r}]\vec{c} \vdash e_2 : \tau \mid \vec{d}}{\Gamma, x : \sigma \oplus \rho \mid \vec{a}, \vec{b}, \vec{c} \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid \vec{d}}$$

T-Nil

$$\frac{}{\cdot \mid \vec{a} \vdash [\,] : L(\tau) \mid \vec{a}, \vec{b}}$$

T-Cons

$$\frac{}{x : \tau, y : L(\tau) \mid \lhd^{\mathtt{ret}}_{x,y}(\vec{a}) \vdash x :: y : L(\tau) \mid \vec{a}}$$

T-CaseL

$$\frac{\Gamma \mid \vec{a} \vdash e_1 : \tau \mid \vec{c} \qquad \Gamma, y : \sigma, z : L(\sigma) \mid \lhd^{x}_{y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \vec{c}}{\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b} \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}}$$

T-Leaf

$$\frac{}{\cdot \mid \vec{a} \vdash \mathtt{leaf} : T(\tau) \mid \vec{a}, \vec{b}}$$

T-Node

$$\frac{}{x : T(\tau), y : \tau, z : T(\tau) \mid \lhd^{\mathtt{ret}}_{x,y,z}(\vec{a}) \vdash \mathtt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

T-CaseT

$$\frac{\Gamma \mid \vec{a} \vdash e_1 : \tau \mid \vec{c} \qquad \Gamma, x : T(\sigma), y : \sigma, z : T(\sigma) \mid \lhd^{t}_{x,y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \vec{c}}{\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b} \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{leaf} \to e_1 \mid \mathtt{node}(x,\ y,\ z) \to e_2 : \tau \mid \vec{c}}$$

Figure 3.7: AARA non-structural typing rules

energy (or to that for $\mathtt{c}$ where k would be 0). On the flip side, the constructor rules *T-Cons, T-Node* each invert this process.

Definition 3.2.2 (shifting)**.** *The shifting operator $\lhd$ transforms an annotation map to redistribute the potential of a list or tree over its constitutent parts. Here, that means applying Pascal's identity to the degree-k potential annotations and copying the annotations for elements. This operator is overloaded across both lists and trees, but to disambiguate whenever necessary, the following explicit definitions are provided:*
$\lhd^{\ell}_{x,y}$ acts on the annotation map a for a list ℓ where $\ell = x :: y$. Formally:

$$\lhd^{\ell}_{x,y}(a) = \lambda i. \begin{cases} a(\ell.\mathtt{e}.j) & i = x.j \lor i = y.\mathtt{e}.j \\ a(\ell.\mathtt{d}_{1}) + a(\mathtt{c}) & i = \mathtt{c} \\ a(\ell.\mathtt{d}_{k+1}) + a(\ell.\mathtt{d}_k) & i = y.\mathtt{d}_k \land k < D_{max} \\ a(\ell.\mathtt{d}_{D_{max}}) & i = y.\mathtt{d}_{D_{max}} \\ a(i) & otherwise \end{cases}$$

$\lhd^{t}_{x,y,z}$ acts on the annotation map a for a tree t where $t = \mathtt{node}(x,\ y,\ z)$. Formally:

$$\lhd^{t}_{x,y,z}(a) = \lambda i. \begin{cases} a(t.\mathtt{e}'.j) & i = x.\mathtt{e}'.j \lor i = y.j \lor i = z.\mathtt{e}'j \\ a(t.\mathtt{d}'_{1}) + a(\mathtt{c}) & i = \mathtt{c} \\ a(t.\mathtt{d}'_{k+1}) + a(t.\mathtt{d}'_k) & (i = x.\mathtt{d}'_k \lor i = z.\mathtt{d}'_k) \land k < D_{max} \\ a(t.\mathtt{d}'_{D_{max}}) & i = x.\mathtt{d}'_{D_{max}} \lor i = z.\mathtt{d}'_{D_{max}} \\ a(i) & otherwise \end{cases}$$

When unambiguous, one may simply write $\lhd$.

The remaining rules for lists and trees, *T-Nil* and *T-Leaf*, do not use shifting, but are still interesting. Empty lists and trees may be annotated with any annotation vector. This is allowed because the resource functions AARA uses are all 0 given an input of size 0, and scaling 0 by any annotation is still 0. Thus, any annotation is as good as any other for these empty data structures.

Finally, the rule *T-Fun* is the only rule that scales an annotation vector. Specifically, it scales the annotation vector for the closure's type context by 0 so that the function closure cannot capture any potential energy from its environment.[6] This zeroing makes it feasible to use functions an arbitrary number of times, as is necessary to support recursive functions. Without such zeroing, closures would hold energy that function calls could spend, and such spending could only be done so many times until the closure would have no energy left to give. Thus, these hypothetical functions might have limited numbers of uses, preventing arbitrary depth recursion. To avoid this issue, closures carry no energy.

[6]Conveniently, because the annotations of a function are stored on the function's type, the annotations of functions in closures avoid this zeroing. Thus functions can be captured in closures without clobbering their energy behaviour.

$$\textsc{V-Unit}\quad \frac{}{\langle\rangle : \mathbb{1}} \qquad \textsc{V-Fun}\quad \frac{V : \Gamma \qquad \Gamma \mid \vec{c} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{a}|\vec{b}} \sigma \mid \vec{d}}{\texttt{C}(V;\ f,\ x.\, e) : \tau \xrightarrow{\vec{a}|\vec{b}} \sigma} \qquad \textsc{V-Pair}\quad \frac{v_1 : \tau \qquad v_2 : \sigma}{\langle v_1,\ v_2\rangle : \tau \otimes \sigma}$$

$$\textsc{V-SumL}\quad \frac{v : \tau}{\texttt{l}(v) : \tau \oplus \sigma} \qquad \textsc{V-SumR}\quad \frac{v : \sigma}{\texttt{r}(v) : \tau \oplus \sigma} \qquad \textsc{V-Nil}\quad \frac{}{[\,] : L(\tau)} \qquad \textsc{V-Cons}\quad \frac{v_1 : \tau \qquad v_2 : L(\tau)}{v_1 :: v_2 : L(\tau)} \qquad \textsc{V-Leaf}\quad \frac{}{\texttt{leaf} : T(\tau)}$$

$$\textsc{V-Node}\quad \frac{v_1 : T(\tau) \qquad v_2 : \tau \qquad v_3 : T(\tau)}{\texttt{node}(v_1,\ v_2,\ v_3) : T(\tau)} \qquad \textsc{V-Context}\quad \frac{\forall x \in \texttt{dom}(\Gamma).\, V(x) : \Gamma(x)}{V : \Gamma}$$

Figure 3.8: AARA value well-formedness rules

3.3 Well-formed Values

The notion of the well-formedness of a value is formalized in Figure 3.8, and this notion is extended over contexts as well. These rules simply state that well-formed values are structurally consistent with their type, and most rules do not have any interaction with AARA's potential energy system. The one exception is that the well-formedness of a function closure at some type requires that the closure is actually typable as that type in the AARA type system.

While the well-formedness of values might be easily taken for granted, this property does play an important role in the typical formulation of AARA's soundness. One reason for this importance is simply to be able to ensure potential energy is well-defined (Section 3.4). But also, it is important to proving the soundness of AARA that well-formed function closures have well-typed function bodies. This importance is because a function's cost behaviour must be recovered when reasoning about the function's application. The function type alone does not carry the function's source code, but well-formedness assures us that such source code exists, and that the code comes with a type derivation.

3.4 Potential Energy

Now that the well-formedness of values has been defined, potential energy can be assigned to such well-formed values. To assign potential energy to some well-formed values $v : \tau$ according the annotation $\vec{a}$, AARA makes use of the potential function $\Phi(v : \tau \mid \vec{a})$ defined in Figure 3.9, which in turn makes use of a special shifting operator defined via Definition 3.4.1. The potential function only assigns nonzero potential energy to lists, trees, values built out of lists or trees, and the free energy of contexts. Note that the annotation $\vec{a}$ may annotate more than just the type τ in the potential assignment $\Phi(v : \tau \mid \vec{a})$. The potential function is also extended to act over well-formed contexts.

$$\Phi(\langle\rangle : \mathbb{1} \mid \vec{a}) = 0 \qquad \Phi(\mathtt{C}(V;\ f,\ x.\,e) : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \vec{a}) = 0$$

$$\Phi(\langle v_1,\ v_2\rangle : \tau \otimes \sigma \mid \vec{a}) = \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{1^{\mathtt{st}}.i}) + \Phi(v_2 : \sigma \mid \lambda i.\, \vec{a}_{2^{\mathtt{nd}}.i})$$

$$\Phi(\mathtt{l}(v) : \tau \oplus \sigma \mid \vec{a}) = \Phi(v : \tau \mid \lambda i.\, \vec{a}_{\mathtt{l}.i}) \qquad \Phi(\mathtt{r}(v) : \tau \oplus \sigma \mid \vec{a}) = \Phi(v : \sigma \mid \lambda i.\, \vec{a}_{\mathtt{r}.i})$$

$$\Phi(v_1 :: v_2 : L(\tau) \mid \vec{a}) = \vec{a}_{\mathtt{d}_1} + \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{i.\mathtt{e}}) + \Phi(v_2 : L(\tau) \mid \blacktriangleleft(\vec{a})) \qquad \Phi([\,] : L(\tau) \mid \vec{a}) = 0$$

$$\Phi(\mathtt{node}(v_1,\ v_2,\ v_3) : T(\tau) \mid \vec{a}) = \vec{a}_{\mathtt{d}'_1} + \Phi(v_1 : T(\tau) \mid \blacktriangleleft(\vec{a})) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{i.\mathtt{e}'}) + \Phi(v_3 : T(\tau) \mid \blacktriangleleft(\vec{a}))$$

$$\Phi(\mathtt{leaf} : T(\tau) \mid \vec{a}) = 0 \qquad \Phi(V : \Gamma \mid \vec{a}) = \vec{a}_{\mathtt{c}} + \sum_{x \in \mathtt{dom}(\Gamma)} \Phi(V(x) : \Gamma(x) \mid \lambda i.\, \vec{a}_{x.i})$$

Figure 3.9: Potential energy definition

Definition 3.4.1 (potential shifting)**.** *The potential shifting operator* $\blacktriangleleft$ *mirrors the critical action of the shifting operator* $\lhd$ *over indices like* $\mathtt{d}_i$, $\mathtt{d}'_i$, $\mathtt{c}$, $\mathtt{e}.i$, $\mathtt{e}'.i$, *but where no labels are present like* x *in* $x.\mathtt{d}_i$.
For lists annotated by a, *let* $b(\ell.i) = a(i)$. *Then formally:*

$$\blacktriangleleft(a) = \lambda i.\ \lhd^{\ell}_{x,y}(b)(y.i)$$

For trees annotated by a, *let* $b(t.i) = a(i)$. *Then formally:*

$$\blacktriangleleft(a) = \lambda i.\ \lhd^{t}_{x,y,z}(b)(x.i) = \lambda i.\ \lhd^{t}_{x,y,z}(b)(z.i)$$

By design, the amount of potential energy assigned by the potential function aligns well with the typing rules of Figure 3.4. To yield tight cost bounds, the physicist's method must conserve as much energy as possible. In AARA, most (but not all) rules exhibit perfect conservation of energy. This conservation is straightforward when indices are simply relabelled. Then for the more intersting manipulations of sharing and shifting, the following lemmas show that energy is conserved:

Lemma 3.4.1 (sharing conserves energy[76])**.**

$$\Phi((x \mapsto v, y \mapsto v) : (x : \tau, y : \tau) \mid \vec{a}) = \Phi((z \mapsto v) : (z : \tau) \mid \curlyvee^{x,y}_{z}(\vec{a}))$$

Lemma 3.4.2 (shifting conserves energy[76])**.**

$$\Phi((x \mapsto v_1 :: v_2) : (x : L(\tau)) \mid \vec{a}) = \Phi((y \mapsto v_1, z \mapsto v_2) : (y : \tau, z : L(\tau)) \mid \lhd^{x}_{y,z}(\vec{a}))$$

$$\Phi((t \mapsto \texttt{node}(v_1,\, v_2,\, v_3)) : (t : T(\tau)) \mid \vec{a})$$
$$= \Phi((x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (x : T(\tau), y : \tau, z : T(\tau)) \mid \lhd^{t}_{x,y,z}(\vec{a}))$$

Nonetheless, there are a few rules that do not conserve energy: in *T-Tick*, the energy differs by r across the turnstile; each of the forms of weakening (*T-Weak,T-Sub*) allow energy to be lost; and *T-App* can lose energy if the function typing does not preserve energy. Because *T-Tick* records the actual cost and *T-App* simply reifies the typing of a function's body, this means that the weakening rules are fundamentally the only source of "extra" energy loss and thus are the only source looseness in AARA's cost bounds. (However, the weakening rules are still important for, e.g., keeping the annotations of branches identical in rules like *T-CaseS*, so the weakening rules cannot be removed.)

Because the potential energy held by lists and trees is defined inductively, it might be hard to immediately tell whether the definition successfully assigns polynomial amounts of energy. In the case of lists at least, the following results exist in the literature:[7]

Lemma 3.4.3 (list potential energy [77])**.** *The potential energy of a list v with annotation $\vec{a}$ is a polynomial function of v's length n plus the potential energy of v's elements.*

$$\Phi(v : L(\tau) \mid \vec{a}) = \sum_{i=1}^{D_{max}} \vec{a}_{\mathtt{d}_i} \cdot \binom{n}{i} + \sum_{v' \in v} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}.i})$$

Lemma 3.4.3 shows that the annotations of lists faithfully act as coefficients of linear combinations of binomial coefficients. This property allows AARA to capture polynomial amounts of energy because binomial coefficients are a linear basis for the space of polynomial functions.

Lemma 3.4.4 (tree potential energy [76])**.** *Let v be a tree of height h. The potential energy of v with annotation $\vec{a}$ is a polynomial function the number of nodes n_i at level i of v plus the potential energy of v's elements.*

$$\Phi(v : T(\tau) \mid \vec{a}) = \sum_{i=1}^{h} n_i \cdot \left(\sum_{j=1}^{D_{max}} \vec{a}_{\mathtt{d}'_j} \cdot \binom{i-1}{j-1} \right) + \sum_{v' \in v} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}'.i})$$

The expression for the potential energy of trees in Lemma 3.4.4 is less intuitive than that for lists. Nonetheless, the expression is still a naturally-arising polynomial amount. In particular, the expression coincides with the potential energy of lists for trees where every node includes a leaf, such as a tree of all left nodes. Such trees are morally just lists, so in this sense, the expression

[7]When I write $v' \in v$ for a list v, I treat v as a multiset to allow for repetition.

naturally generalizes the “nice” expression of Lemma 3.4.3. Other work [65] generalizes polynomial resource functions over trees in further ways, but the definition here is sufficient for this thesis.

To conclude this section, I state Lemmas 3.4.5 and 3.4.6, which are simple, related properties that are useful for reasoning about potential energy in AARA. Both these lemmas also hold over entire contexts, but are stated here for single values.

Lemma 3.4.5 (pointwise monotonicity of potential energy)**.**

$$\vec{a} \geq \vec{b} \implies \Phi(v : \tau \mid \vec{a}) \geq \Phi(v : \tau \mid \vec{b})$$

Because the annotation of all zeros assigns zero potential energy, a direct consequence of Lemma 3.4.5 is that nonnegative annotations always assign nonnegative potential energy.

Lemma 3.4.6 (linearity of potential energy)**.**

$$\Phi(v : \tau \mid \vec{a} + \vec{b}) = \Phi(v : \tau \mid \vec{a}) + \Phi(v : \tau \mid \vec{b})$$

In Lemma 3.4.6 the expression $\vec{a} + \vec{b}$ is the pointwise addition of the $\vec{a}$ and $\vec{b}$ where both have the same indices. This lemma expresses another form of the conservation property of Lemma 3.4.1 but for a form of sharing over all indices at once.

3.5 Soundness

The soundness of the AARA type system amounts to the correctness of its energy bookkeeping with respect to actual execution costs. This property is formalized via Theorem 3.5.1 alongside the property that the return value of the expression's evaluation is well-formed.[8] One can see the physicist's method in this soundness theorem. The energy of the initial program environment gives an upper bound on the peak cost. The difference in energy between the initial program environment and that of the progam's return value bounds the net cost.

[8]The well-formedness of the value returned by the expression's evaluation is often elided in AARA literature. This property follows almost entirely for the same reasons as type preservation in the Hindley-Milner type system that AARA is built upon, and therefore the property is not a particularly interesting part of AARA's soundness. However, I take the time to explicitly include it here for completeness.

$$\textsc{V-Nont}\quad \frac{}{\bullet : \tau} \qquad\qquad \Phi(\bullet : \tau \mid \vec{a}) = \infty$$

Figure 3.10: Additional nontermination rules and definitions

Theorem 3.5.1 (polynomial AARA soundness[77]). *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) - \Phi((\texttt{ret} \mapsto v) : (\texttt{ret} : \tau) \mid \vec{b}) \geq p - q$ *(difference bounds net)*

I attribute this theorem to Hoffmann and Hofmann [77] (see [76] for a detailed proof). While my statement here uses a slightly different formalization, there is no major conceptual difference between the two results. Hoffmann and Hofmann prove the soundness theorem by lexicographic induction over the evaluation judgment followed by the typing judgment. This approach to proving soundness will be repeated in later chapters.

As a result of this soundness theorem, one can be assured that AARA's cost analysis is accurate. That is, whenever AARA assigns the type of $\tau \xrightarrow{\vec{a}|\vec{b}} \sigma$ to some function, that function requires no more than $\Phi((\texttt{arg} \mapsto v) : (\texttt{arg} : v) \mid \vec{a})$ resources to run given input value v, and spends no more than $\Phi((\texttt{arg} \mapsto v) : (\texttt{arg} : \tau) \mid \vec{a}) - \Phi((\texttt{ret} \mapsto v') : (\texttt{ret} : \sigma) \mid \vec{b})$ resources in net given output value v'.

3.6 Nontermination

To derive the peak costs of nonterminating computations, other AARA literature usually extends Theorem 3.5.1 over a set nontermination judgments as discussed in Section 2.4. However, I simplify the work of this thesis by using one new "terminating" evaluation judgement given in Figure 2.5. This evaluation simulates nondeterministically aborting computation and returning the dummy value $\bullet$.

To go along with this new evaluation judgment, I now provide the corresponding well-formedness rule and the definition of potential energy for the value $\bullet$ in Figure 3.10. The value $\bullet$ is well-formed at any type and carries infinitely much potential energy. It should be noted that these definitions maintain many of the properties of AARA pointed out before, such as evaluation maintaining well-formedness, the pointwise monotonicity of potential energy (Lemma 3.4.5), and the nonnegativity of potential energy.

Using these rules, nontermination can incorporated into AARA's soundness theorem, and with far less work than the large set of rules from the literature. However, one additional adjustment must be made to the soundness theorem to avoid any issues arising from arithmetic with infinity. This adjustment is simply the arithmetic rearrangement of the expression of the net cost bound to avoid subtraction, and thus avoid the indeterminate expression $\infty - \infty$. Instead of

$$\Phi(V : \Gamma \mid \vec{a}) - \Phi((\mathtt{ret} \mapsto v) : (\mathtt{ret} : \tau) \mid \vec{b}) \geq p - q$$

I use

$$\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((\mathtt{ret} \mapsto v) : (\mathtt{ret} : \tau) \mid \vec{b}) + p$$

With this formulation of net cost bounds, I now show Theorem 3.6.1 to emphasize how the net cost reasoning of Theorem 3.5.1 can elegantly be fully maintained in the presence of the new nontermination rule.

Theorem 3.6.1 (AARA net cost soundness with nontermination)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((\mathtt{ret} \mapsto v) : (\mathtt{ret} : \tau) \mid \vec{b}) + p$ *(difference bounds net)*

Proof. The statement is proven by cases:

- If the derivation of $V \vdash e \Downarrow v \mid (p, q)$ does not use *E-Nont*, then Theorem 3.5.1 suffices.
- If the derivation of $V \vdash e \Downarrow v \mid (p, q)$ does use *E-Nont*, then $q = \infty$ by Lemma 2.4.2, which trivializes the net cost inequality.

□

More generally, the peak cost bounds found by AARA also hold in the presence of the rule *E-Nont*. This fact can be proven with the essentially the same proof cases as are present in the original proof of Theorem 3.5.1 plus one new trivial proof case for *E-Nont*. However, it would not be instructive to reproduce the the original proof here. Instead, I provide only the new proof case here, and I let the soundness proofs of later chapters (particularly Chapter 5) serve as full witnesses of the soundness of AARA with these rules for potentially nonterminating programs.

The new case of the soundness proof goes as follows: Suppose the last rule applied for the evaluation judgment is *E-Nont*. Then the cost behaviour is $(0, \infty)$, so the peak cost bound inequality is an upper bound on 0, and the net cost bound inequality is a lower bound on ∞. Because potential is always nonnegative, the peak cost bound is satisfied. And because ∞ is greater than or equal to everything, the net cost bound is also satisfied.

3.7 Automation

A type-inference algorithm is used to automate the AARA analysis. This algorithm uses standard type-inference techniques to derive a typing skeleton and then adds annotations to that skeleton.

After fixing a max polynomial degree D_{max} for the analysis, the AARA type-inference algorithm takes the following steps:

1. identify contraction and weakening for variables
2. basic type inference
3. collect and solve linear contraints

The first step of type inference is to set up a basic type derivation skeleton. To set up this skeleton, one must first identify all places in the code where variables are used more than once or not at all. Each of these places requires a use of either the contraction rule *T-Contract* or the weakening rule *T-Weak*, respectively. These structural rules are *not* idempotent, so they must be carefully inserted in the correct places in the derivation skeleton. (In contrast, the subtyping rules are idempotent and can be handled by blindly inserting them between the uses of any other typing rules.) With the use of the non-idempotent structural rules identified, a basic type derivation skeleton can be set up that does not use annotations.

The next step of type inference is to infer non-AARA types for the program, without any annotations. Typically this typing is accomplished via some form of unification such as Hindley-Milner type inference [72, 106]. After this step is complete, every subexpression and variable of the program will have been assigned some concrete type.

The final step of type inference is to decorate the type derivation with linear constraints and solve those constraints. Because every subexpression and variable has been assigned a concrete type, the indices annotating those expressions and variables can be computed using $Ind(-)$. After generating fresh annotation vectors with the appropriate indices everywhere on the type derivation skeleton, all that remains is relating those vectors according to the AARA typing rules and solving the resulting linear program. If the AARA typing rules justify any type, this linear progam will find one. To optimize the tightness of that type, AARA type inference uses an objective function which first minimizes the annotations of the type context (especially for energy of high degree), then maximizes the annotations of the expression being typed, (including the associated leftover resources at the index `c`).

Each step of AARA type inference is efficient in practice. In the first step, the uses of weakening and contraction can be identified in linear time. In the second step, Hindley-Milner type inference[72, 106] is sufficient for AARA and efficient in practice.[9] In the final step, the linear progam takes at worst polynomial time in the number of constraints [103], and this number of constraints turns out to be polynomial in both the chosen maximum degree D_{max} and the size of the source code,[10] Thus, the final inference step runs in polynomial time overall, so each step of the inference process is efficient.

Note that AARA type inference cannot always succeed. If AARA could always assign a type, then AARA could be used to decide the undecidable halting problem[135]. For similar reasons, AARA cannot always assign a tight cost bound, even for polynomial-time functions. Nonetheless, if the AARA typing rules can justify *any* types for a given program, this inference algorithm picks one of those types yielding an optimally tight cost bound (in the sense of ”optimal” corresponding to the linear program's optimization).

[9]However, pathological cases do show that Hindley-Milner inference is DEXPTIME-complete [99, 105]

[10]I revise this polynomial complexity in Chapter 8 where some additional language features are introduced.

3.8 Linearity

I take a moment now to emphasize some of the ways that linearity shows up in AARA and how such linearity can be leveraged. I hope that this will prime the reader for later chapters, where leveraging such linear features plays a central role.

While many of the forms of linearity in AARA are interrelated, I identify at least four avenues through which linearity arises. These linear features are what this thesis will leverage to improve AARA.

- logical linearity — AARA's type system has the flavor of linear logic[63]. More specifically, the type system is affine. AARA variables and potential energy can be used *at most* once (affinely), but AARA does not require that they are used *at least* once (relevantly); both are needed for full linearity. Regardless, the fact that a resource-analysis type system has a linear flavor is not really that surprising because linear logic is well established as a logic of resources [1, 29, 63]. AARA's logical linearity plays some role in all this thesis's contributions, but is especially apparent in Chapter 5's adaptation of linear-logic proof-search techniques and in Chapter 9's affine *and relevant* flavor of type system.
- linear progamming — Linear programming automates AARA's type inference and is the key to AARA's success as an automatic cost-analysis tool. Linear programming is both efficient and expressive, which in turn allows AARA to be efficient and expressive. It is a powerful coincidence that the contraction rule of AARA can be expressed with only linear constraints, allowing AARA's otherwise affine type system to type non-affine code. It is an important theme throughtout this thesis to express my contributions' new reasoning principles in terms of linear constraints so that my AARA extensions can continue to be efficienty automated.
- linear recurrences — AARA uses the linear recurrence of Pascal's identity to derive polynomial cost bounds. It is yet another powerful coincidence that such linear recurrences fit into the framework of linear programming so that AARA can infer polynomial cost bounds just as easily as linear cost bounds. This power is expanded in Chapter 6 to express and efficiently derive even more kinds of cost bounds.
- the physcisist's method — The physicist's method that powers AARA's reasoning allows for physical principles to seep into the AARA type system. So far the only principle that has really come into play is conservation of energy. However, both Chapters 5 and 9 show that other interesting physical principles have parallels in the domain of cost analysis, including, e.g., the highly linear domain of quantum mechanics.

It is instructive to point out that AARA itself was born out of leveraging linearity. The original AARA type system was developed by Hofmann and Jost [85] by leveraging the synergistic linearity of the following key ideas:

1. The first key idea was to build their heap analysis upon Hofmann's *linearly typed functional (programming) language* (LFPL) [83]. Programs typeable in this system are guaranteed to require no additional heap memory beyond that provided by the input. The type system ensures this property by explicitly passing around heap rights in the form of tokens represented by $\diamond$. These tokens are introduced to programs through input data stuctures and nowhere else, which ensures that the only heap cells that can be used are those of the

input. Critically, this means such tokens are affine, and this affine nature is the eponymous element of *linearity* in *L*FPL.

2. The second key idea was that numerical-optimization techniques could greatly simplify the handling of LFPL-style tokens. One might initially observe that groups of tokens can be summarized as natural numbers, like representing 3 tokens simply with the number 3. This transformation naturally reduces type inference to NP-hard integer linear programming (ILP). However, Hofmann and Jost's work went a bit further: they leveraged the simplifying linear-programming technique of *linear relaxation* to the ILP, which places the *integer* optimization problem into a *rational* optimization space where efficient linear programming can be applied. Crucially, the rational linear programming solutions were easily adapted into their LFPL-style type system and were just as useful for heap analysis. Thus, type inference could be efficiently automated in polynomial-time via linear programming.

Thus, from the beginning of AARA, leveraging linearity has been important. For Hofmann and Jost, this leveraging of linearity played out in the adaptation of techniques from linear programming into their type system. Later work has continued the tradition by leveraging other forms of linearity, including the introduction of a linear recurrence to represent polynomials [77]. This thesis aims to show that leveraging linearity has not yet been exhausted.

Chapter 4

Related Work

This chapter presents a brief description of some alternative approaches to cost analysis. These alternatives have been organized along three axes that should contextualize the reader's introduction to AARA in Chapter 3. Section 4.1 exhibits other type-based approaches to cost analysis, many of which make use of linear flavors of typing. Section 4.2 describes some techniques that focus on amortized cost analyses specifically, showing that there exist other techniques using the physicist's method. Section 4.3 shows alternative approaches to automatic cost analysis, many of which rely on the existence of some solver.

Each of this thesis's chapters also provides more close comparisons between their contributions and other related work.

4.1 Type Systems

Outside of AARA, there are many other type systems that perform cost analysis. These type systems take various approaches to cost analysis and have various levels of automation. Most are not nearly as automated as AARA, but many exchange such automation for more power and can be used to verify more interesting cost bounds than AARA.

A variety of type systems for cost analysis get their power from implicit computational complexity, wherein the typeability of a program implies that the program (or the function it computes) has a certain complexity. Typically, these type systems operate via some form of logical linearity, wherein a program value can only be used so many times. Such type systems include non-size-increasing types [84], those based on light linear logic [13], those based on bounded linear logic [42, 61], and linear dependent types [11, 41, 43]. AARA descends from such work through LFPL [83].

Modal type systems can also enable cost analysis [44, 118, 124]. Usually, modal type systems perform cost analysis by encoding amounts of resources or costs via grades, which can take the form of numerical annotations. AARA can be expressed this way too [125]. Temporal type systems can also express time usage more directly [45, 92].

Other type systems make use of the power of dependent types, wherein types can depend on program values. These types have already been mentioned for their use in implicit complexity results [11, 41, 43], but their use ranges widely. At one extreme, dependent types might only show up to allow costs to depend on arguments [39, 64]. Such simple dependent type systems often admit some level of automation. At the other extreme, the use of dependent types goes beyond mere cost analysis and takes the form of a general-purpose proof assistant [108, 134]. The CALF system [114] specializes this general approach for the purposes of finding cost bounds by introducing a special modality that distinguishes extensional behaviour (like correctness) and intensional behaviour (like cost). These proof systems are usually too powerful to fully automate their type inference.

Like dependent types, refinement types also allow the relation of types and values. However, refinement types do so in lighter-weight way, making them easier to automate. Systems like Liquid Haskell have used specialized refinement types for checking resource usage [71].

The TiML type system [139] is another type system which has been specialized for automatically verifying time complexity. Similarly to AARA, TiML uses type annotations to facilitate its cost reasoning. However, unlike AARA, these annotations are user-provided and must be discharged by an SMT solver. While this circumstance makes TiML harder to automate, it also affords more expressive power to TiML.

An indirect approach to cost analysis is to track the sizes of data structures. Sized types are capable of such tracking with some level of automation [32, 90, 137]. While data structure size is not directly related to cost, many times costs of interest are parameterized on such sizes. AARA skips reasoning directly about size and instead reasons directly about resource densities in data structures.

4.2 Amortized Analyses

Amortized cost analysis originates from the analysis of algorithms [133] and is a technique by which prepayments smooth over costs. Amortized costs are more natural for reasoning about the accumulated cost of series of operations because they can amortize the costs of rare but expensive operations that might otherwise inflate the bounds found by a more naive method of accumulating costs. Because amortization is so well-suited to composing the costs of series of operations, it also serves as a strong foundation for programmatic cost analysis.

Many different approaches to cost analysis provide amortized analyses. AARA of course does, as do various AARA-based systems [27, 47], some of which are in settings like program logic [10] and term rewriting [107]. There are additionally amortized techniques based on recurrence solving [40], cost relations [58, 59], separation logic [66], ranking functions for lossy vector addition systems [131], and more [15, 56, 104, 113].

Amortized analyses typically approach cost analysis via resource analysis. That is, these analyses track some resource that gets spent, rather than track some cost that gets accumulated. While it usually equivalent to track cost or resources, this distinction does mean that a resource analysis might care more about where resources are located and what form they take. As a result, amortized analyses tend to use the abstraction of resource credits or potential energy. Such analyses often focus on how to move or store such credits where they are needed. Allowing such accounting to happen smoothly is a major theme of this thesis.

4.3 Automatic Approaches

There are many different techniques one could use to automate cost analysis, some of which have already been mentioned in previous sections. In this section, I briefly describe some remaining categories of automatic techniques, of which there are many. The categories I mention here are by no means exhaustive, or even mutually exclusive.

Recurrence Solving One common approach to automatic cost analysis is recurrence solving. This approach aims to extract recursive functional relations between various program properties, including cost, data-structure size, recursion depth, etc. Then these techniques solve for the functions involved, disentangling cost from its relations with other program properties. Such a solution can come easily for certain solvable forms of recurrences, such as linear recurrences[1] or the master theorem [18]. However, recurrence solving in general is not computable, and thus more powerful recurrence-based approaches must rely on the strength of more sophisticated techniques.

The use of recurrence solving for automatic cost analysis goes all the way back to Wegbreit's 1975 work on METRIC [140]. The METRIC system approaches cost analysis by gathering and solving recurrence relations using difference equations. Modern techniques improve upon this early work by using sophisticated solution methods. For example, techniques like PILAT [48] deal with systems of polynomial equations that arise from recurrences by computing Gröbner bases [22]. Other modern techniques gain more power by using SMT solvers to deal with more difficult recurrences [20, 55, 101, 126]. Machine learning has even been used to guess recurrence solutions which can then be checked by an SMT solver [100].

Cost Relations A more directed framework for cost analysis following the Metric framework [140] is the use of *cost relations*, which are specialized recursive equations and constraints describing the cost behaviours of programs. Tools like CoFloCo [59], and PUBS [5] have been developed for the purposes of solving such cost relations. Other tools use cost-relation solvers as part of their cost analysis algorithms [4, 7, 9]. Many times cost relations are combined with other techniques, like the ranking functions I discuss in the following paragraphs.

Ranking Functions Ranking functions are a key method of proving program termination. Ranking functions operate by identifying decreasing measures that must eventually reach zero. Tools for the inference of ranking functions have been adapted to reason about computational complexity by translating ranking functions into bounds, often in combination with other techniques [5, 8, 21, 31, 130, 131, 141].

Ranking functions are particularly prevalent for proving the termination of term-rewrite systems, which describe computation via string derivations. Some automatic complexity analysis tools in this domain find ranking functions using specialized term-rewriting techniques like the dependency-pair method [12, 62, 73, 115].

[1]See Chapter 6 for linear recurrences being used in AARA.

Invariant Generation An alternative method of cost analysis treats a program's cost as an invariant. Such an invariant might relate the size of a program's input with the value of some step counter. Invariant-finding techniques can then be employed to obtain cost bounds. Invariants can either be directly related to cost or leveraged through techniques like Speed [69].

Some techniques for finding invariants come from linear algebra. Such techniques are based upon the invariant kernels or eigenspaces of linear (or affine) program operations [49, 97].

Finally, abstract interpretation is a major technique for finding program invariants. I discuss abstract interpretation more in the following paragraphs.

Abstract Interpretation Abstract interpretation is a technique for approximating program behaviour, especially behaviour like the aforementioned invariants. Abstract interpretation typically operates using iterative techniques to approach fixed points in some abstraction of program behaviour [37, 54]. By selecting the correct abstract domain, invariants such as linear and polynomial (in)equalities can be found effectively [23, 38, 110, 128].

Abstract interpration is often combined with other techniques to make a complete cost analysis. Sometimes the role of abstract interpretation in such tools is just to simplify the problem enough enough that other kinds of solvers can take over [68, 69]. Other times abstract domains can more directly aid in cost in reasoning, such as combining size-change abstraction with ranking functions [141].

Program Logic Some specialized program logics have been introduced for the purposes of automatic cost analysis rather than mere verification. Some example of automated logics include an AARA-based separation logic [10], an energy-aware Hoare logic that builds off of other loop analyses [98], and the separation-logic tool Infer, which is powered by bi-abduction [24].

Chapter 5

Remainder Contexts

This chapter lays out *remainder contexts*,[1] which are this thesis's first extension to the AARA type system. Remainder contexts are additional type contexts that capture the remaining energy of the context *after* program execution. These new contexts allow AARA to better conserve reusable resources like memory. Further, because remainder contexts only require linear constraints, AARA type inference can be efficiently automated via linear programming as before.

Remainder contexts operate through a combination of principles originating from linear logic and quantum computing. Remainder contexts are AARA's analogues of the I/O contexts from linear-logic proof search [28, 75], and like those I/O contexts, they allow for deeper reasoning about leftover resources. Then, to better reuse those resources, remainder contexts engage in a form of *uncomputation* [17]. Reversible computing uses uncomputation to recover and reuse bits assigned as temporary memory, whereas AARA uses uncomputation to recover and reuse leftover potential energy assigned to temporary variables. Together, these ideas allow AARA to recover leftover resources more effectively, which is the crux of working with reusable resources like memory.

As a result of introducing remainder contexts, AARA's structural rules no longer make use of *T-Contract* or *T-Weak*. This change makes the type-inference process simpler because the algorithm no longer needs to identify the uses of such rules (step one in Section 3.7).

5.1 The Problem: Keeping the Change

For AARA to provide tight cost bounds, it is critical that its type system is able to conserve potential energy as much as possible. However, there are various ways that the type system can lose energy. One source of loss in particular harms AARA's ability to reason about reusable resources like memory: AARA does not recover potential energy left on temporary data structures. If potential energy were money,[2] it would be as if AARA (over)pays with a large bill and does not bother to recieve the change. As a result, such potential energy is lost rather than reused, inflating the cost bounds found by AARA.

[1] I first published remainder contexts in [96] in service of the work I will present separately in Chapter 9.

[2] In fact, potential energy might literally be money if money is the resource tracked by ticks.

```
   let a = len lst
   let b = mem (5, lst)
```

Figure 5.1: Code sequentially calling `len` and `mem`

```
fun len lst = case lst of
  | [] -> 0
  | x::xs -> let () = tick{1} in
    let subsoln = len xs in
    let () = tick{-1} in
    1 + subsoln

  fun mem (y, lst) = case lst of
  | [] -> false
  | x::xs -> if x = y then true
    else let () = tick{1} in
      let subsoln = len xs in
      let () = tick{-1} in
      subsoln
```

Figure 5.2: Code for `len` and `mem` with ticks for naive stack frames

To see how AARA fails to keep the change, consider the call-stack usage[3] for sequential calls to functions in Figure 5.1. Each of `len` and `mem` are functions that operate by recursing over their input lists and thus make a worst-case number of recursive calls equal to their inputs' lengths (plus one to account for the initial non-recursive call). One can therefore expect these functions might each use a number of stack frames equal to the length of `lst` plus one. Critically, however, these stack frames are freed after each function returns, so the stack frames for the call to `len` can be reused by the call to `mem`. Thus, in total, the code of Figure 5.1 should require a peak cost of $|\texttt{lst}| + 1$ call stack frames to run. The code should also require a net cost of 0 stack frames because all stack frames are returned.

The cost bounds found by AARA do not agree with this analysis. After instrumenting the code for `len` with ticks to count stack frames (Figure 5.2), AARA assigns `len` the annotated type $L^1(\mathbb{Z}) \to \mathbb{Z}$. This type does show that one stack frame per element of the input list is required, but does not indicate in any way that these stack frame are later returned. AARA types `mem` similarly. As a result, AARA cannot reuse these stack frames in its bookkeeping and finds that each function call in Figure 5.1 requires a fresh set of stack frames to run. The peak cost bound found by AARA is $2 \cdot |\texttt{lst}| + 1$ stack frames[4] and the net cost bound is $2 \cdot |\texttt{lst}|$.

The situation is actually worse than it might appear. AARA's bounds can be made arbitrarily

[3]To keep examples simple, this work only considers *naive* call stack usage, wherein every function call causes a new call stack frame to be allocated. This work ignores optimizations like tail-call optimization, and this work ignores the possibility of memory leaks where the call stack is not freed after a function returns.

[4]This bound is *not* $2 \cdot |\texttt{lst}| + 2$ because AARA actually can reuse the single stack frame needed for the initial function calls without any alteration.

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \otimes R$$

Figure 5.3: Right rule for multiplicative product

loose simply by calling more functions on the list `lst`. If k such sequential function calls are made, the calls still only require a peak cost of $|\texttt{lst}| + 1$ stack frames due to reuse, but AARA finds a peak cost bound of $k \cdot |\texttt{lst}| + 1$, which is about k times worse. The net cost situation is worse still: in actuality, k sequential function calls incur a net cost of 0 stack frames because all stack frames are returned, but AARA finds a net cost bound of $k \cdot |\texttt{lst}|$. And if calls are made to superlinear-cost functions rather than linear-cost functions like `len` and `mem`, these bounds will also become superlinear.

Thus, as it stands, AARA is poorly suited for deriving cost bounds for reusable resources like memory. AARA's analysis does not take advantage of when such resources could be reused— does not keep the change— and therefore often finds much looser cost bounds than the actual cost. This is the problem addressed by the current chapter of this thesis.

5.2 The Linear Ideas: I/O Contexts and Uncomputation

To address the problem described in Section 5.1, I follow the theme of this thesis and look to how similar problems have been addressed in other linear domains. For this problem, I look to practices in both linear logic and reversible computation. Linear logic proof search can be made more efficient through the use of *I/O contexts* to organize leftover propositions [28, 75]; a similar construction allows AARA to express leftover potential energy. Reversible computing uses *uncomputation* to recover and reuse bits assigned as temporary memory [17]; a similar process allows AARA to recover leftover potential energy from temporary variables. This section describes both of these techniques in more detail to give a taste of how the patterns of remainder contexts naturally arise in other linear settings.

5.2.1 I/O Contexts

The basic inference rules for linear logic [63] are not so easy to use algorithmically. The problem stems from rules for the multiplicative connectives such as the sequent calculus rule in Figure 5.3 for the multiplicative product $\otimes$. Linear logic does not allow the free duplication (contraction) of propositions in its logical context, so it is critical that proofs properly allocate such propositions between premiss judgments. However, there are $2^{|\Gamma, \Delta|}$ different ways to split up the logical context Γ, Δ between the two premisses in the rule $\otimes R$. Thus any linear logic proof search procedure that naively applies the rule $\otimes R$ may have to backtrack to the application of this rule exponentially many times to try each way of splitting the context.

I/O contexts provide a different formulation of rules to mitigate such expensive backtracking during proof search. An example proof rule using I/O contexts is given in Figure 5.4, where the rightmost symbol in each judgment is the new logical context for leftover propositions. The key

$$\frac{\Gamma \vdash A \mid \Omega \quad \Omega \vdash B \mid \Delta}{\Gamma \vdash A \otimes B \mid \Delta} \otimes R$$

Figure 5.4: Right rule for multiplicative product using I/O contexts

to using I/O contexts is that such rules do not split the logical context at all. Instead, the rules pass the entire context to their first premiss, where I/O contexts then are used to dynamically track which propositions gets used and which propositions are leftover. The leftovers from deriving the first premiss may then be passed wholesale to the second premiss.

The dynamic tracking enabled by I/O contexts is a more efficient because it provides a flexible, lazy way of determining where propositions get used in linear logic proofs. Rules without I/O contexts must eagerly commit to how propositions get used through splitting their contexts, and this is a major source of backtracking during proof search. While using I/O contexts does not completely eliminate the need for backtracking—there may be multiple ways to derive the first premiss— it does have the effect that only those context splits satisfying the first premiss are ever considered, cutting down the proof search space significantly. As a result, linear logic proof search is more efficient when using such I/O contexts.

The intention of adapting I/O contexts to AARA is to gain the ability to reason about leftovers effectively. However, the adaptation is not completely trivial. In AARA, these leftovers take the form of potential energy rather than propositions, and potential energy is a continuous quantity that can be split up, whereas propositions are used discretely and completely. The programmatic context of AARA also means that AARA can only easily reason about the data structures existing in a given program context, whereas logical propositions can be introduced and passed around more freely. Nonetheless, both of these hurdles can be overcome, especially with the help of uncomputation.

5.2.2 Uncomputation

The physical concerns of computation come up against various limitations. Some of the most important concerns are about information-theoretical physical consequences of irreversible computation. Physics typically assumes that all phenomena are reversible so that information can never be destroyed and symmetrically can never be created—in other words, information is logically linear. However, irreversible computations, like the non-invertible constant function $\lambda x.\, 0$, are ubiquitous in computer science. The study of physical computation has therefore introduced special techniques to reconcile the linearity of physics and the nonlinearity of general computation.

One limitation of physical computation is the amount of energy dissipated through irreversible computation [102]. The energy bottleneck is actually the creation of entropy through the erasure of memory: each bit of memory at room temperature requires approximately $3 \cdot 10^{-21}$ Joules of energy to erase. This energy cost arises because, while the memory might be erased, the information it represents cannot be. Instead, that information is sent out into the environment in a form such as heat, and any computational device that pumps out heat must take in energy. Unfortunately, clearing memory is incredibly common in computation because memory is usually reused, especially between different computations. It therefore would appear that general-purpose computation is inherently entropic and there are limits on how energy efficient it can be.

To address such concerns, Bennett showed how to perform general-purpose computation in a fully reversible manner, allowing the plausibility of arbitrarily energy-efficient computation [17]. The key to this result is to record the intermediate computation states necessary to perform *uncomputation*.[5] Rather than *erase* bits, uncomputation *reverses* the process used to compute those bits. This process leaves the bits back in the state they started in, changing no information content along the way.

One simple way to make the computation of any function f reversible is to lift its action to pairs of inputs and outputs. Specifically, if $f(x) = y$ for bit strings x and y, then the function $g(x, z) = (x, z \oplus f(x))$ simulates f, where z is arbitrary and $\oplus$ is pointwise exclusive-or. This choice of g is then its own inverse. This chapter makes a similar change to function types, pairing up arguments with their returns.

Uncomputation has only gained stronger interest with the advent of modern quantum computation. Quantum mechanics has much of the same linear informational concerns, except with higher computational stakes. Erasing quantum bits (qbits) willy-nilly does not simply have an energy cost, but rather completely ruins the computation. This ruination occurs because the benefits of quantum computation rely on qbits existing in certain correlated states, i.e., on qbits being *entangled*. Erasing an entangled qbit collapses the entanglement, ruining the quantum correlations for qbits that were not erased. To avoid this problem, uncomputation is used pervasively. High level quantum languages like QCL[117] and Silq[19] even automatically use uncomputation to remove temporary variables from scope at the end of their lifetimes.

The use of uncomputation in AARA is also for handling temporary variables at the end of their lifetimes. This uncomputation is not actually performed by the program execution, but simulated in the potential energy bookkeeping. Such simulation returns leftover potential energy carried by temporary variables back to the data structures involved in their creation. As a result, less potential energy is lost to being stranded on such variables, resulting in tighter cost bounds. While not every operation in the language is uncomputed in this manner, enough potential energy can be recovered to obtain tighter cost bounds for reusable resources.

This uncomputation could not be performed without also incorporating the leftover tracking of I/O contexts. Such leftovers are what is uncomputed. At the same time, tracking the leftover potential energy would not be meaningful unless it could be recovered. Thus, both ideas are necessary for the success of remainder contexts.

5.3 Type System

I now proceed to adapt the ideas of I/O contexts and uncomputation into AARA via remainder contexts. The resulting type system is able to maintain leftover potential energy on data structures, just as I/O contexts maintain leftover propositions. The type system is then further able to

[5]Uncomputation is not to be confused with uncomputability. The former is the undoing of computation, and the latter is the inability to algorithmically create.

recover that potential energy for reuse, just as uncomputation recovers memory for reuse. My adaptations require only a few systematic changes to the type system, which I explain in this section.

5.3.1 Types

There is precisely one change to the types represented in the AARA system with remainder contexts: function types have slightly augmented annotations. In Chapter 3, the function type $\tau \xrightarrow{\vec{a}|\vec{b}} \sigma$ required the domain of the annotation map $\vec{b}$ to be $\{\mathtt{c}\} \cup \mathtt{ret}.\mathit{Ind}(\sigma)$. With remainder contexts, the required domain is instead $\{\mathtt{c}\} \cup \mathtt{ret}.\mathit{Ind}(\sigma) \cup \mathtt{arg}.\mathit{Ind}(\tau)$. This change just adds indices for the function argument to the return's annotations.

The intention of these new function indices to represent the amount of potential energy leftover on the function argument after the function is run. This notion is formalized by the typing rules of section 5.3.2, where such leftover tracking is pervasive.

To express these new function types conveniently in prose, the function notation is extended so that annotation for the return is added onto the end. Such a function type might look like $\langle L^3(\mathbb{Z});\, 1\rangle \to \langle \mathbb{Z};\, 4\rangle \sim L^1(\mathbb{Z})$, where $L^1(\mathbb{Z})$ has the same base type as the argument, just with different remainder annotations. This type indicates a function taking an integer list with 3 units of energy per element and 1 unit of free energy as input and returning an integer alongside 4 units of free energy, where 1 unit of energy per element is also restored to the input after running.

5.3.2 Typing Rules

The remainder context typing rules are given across Figures 5.5 to 5.7. These new typing rules make use of a similar typing judgment to that used in Chapter 3, except the judgment has an additional set of annotations for the type context. This extra set of annotations lets the typing judgment act in a way reminiscent of Hoare triples[74], or a two-vocabulary relation over annotation indices, or (most pertinently) I/O contexts.

The new typing judgment is:

$$\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$$

This typing judgment means that, in type context Γ annotated by $\vec{a} \geq 0$, the expression e is typed τ, where both τ and the leftover energy *of the whole type context* Γ are annotated by $\vec{b} \geq 0$. To be explicit, $\vec{a}$ is indexed by $\mathit{Ind}(\Gamma)$ and $\vec{b}$ is indexed by $\mathit{Ind}(\Gamma, \mathtt{ret} : \tau)$.

The only difference between this typing judgment and that of Chapter 3 is that here $\vec{b}$ also annotates the type context Γ. This additional annotation of Γ is how I/O contexts show up and is where uncomputation occurs. For convenience, I refer to $\vec{a}$ as the *initial* annotation and $\vec{b}$ as the *remainder* annotation.

Unlike the rules of Section 3.2, remainder contexts do not require any typing rules for the contraction or weakening of variables. Instead, the only structural rule used here is the subtyping rule *R-Sub* of Figure 5.5, which as before is a weaking rule for potential energy. In both cases, the reason that the other structural rules are no longer necessary is that the remaining rules perform their functions better. The reason that variable contraction is no longer necessary is that remainder contexts have the same lazy context usage of I/O contexts, and therefore the eager duplication

$$\textsc{R-Sub}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a} \geq \vec{a'} \qquad \vec{b} \leq \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

Figure 5.5: Remainder context structural typing rule

of variables using the rule *T-Contract* is unnecessary. The reason that variable weakening is no longer necessary is that the rule *T-Weak* does not conserve potential energy as well as remainder contexts' syntax-directed uncomputation.

The behaviour of I/O contexts shows most clearly in the rule *R-Let*. Because the AARA language is in let-normal form, the previous let rule *T-Let* was the only rule displaying the linear multiplicative connective behaviour of splitting the context. The new version with remainder contexts, *R-Let*, exhibits the characteristic I/O context behaviour of passing the whole type context to the first premiss (annotated by $\vec{a}$), then passing all the leftovers to the second premiss (annotated by $\vec{c}$).

To see how I/O contexts' lazy context usage comes into play, one can look at how variables get used in the typing rules. Whenever a variable x is used, the potential energy on x is split between the use of x and the remainder left on x. This splitting is accomplished via sharing and is most clear in the rule *R-Var*, where no other complicated behaviour occurs. Because this splitting occurs lazily just before x gets used, it is capable of more flexibility than the eager splitting performed by the rule *T-Contract*. Remainder contexts are the perfect way to capture this flexibility.[6]

The uncomputation of remainder contexts is the final piece of the puzzle, and this feature can be seen in any destructor expression. For example, look at the rule *R-CaseP*. This rule performs exactly the same operation on the annotation vectors $\vec{a}$ and $\vec{b}$ for its initial and remainder annotations. For the initial annotation, this transformation has the effect of annotating the deconstructed parts of a data structure from the data structure's initial annotations, as might be expected. However, for the remainder annotation, this transformation has the effect of annotating the *reconstructed* original data structure from the remainder annotations of its parts. This perfect symmetry between destruction and reconstruction allows leftover potential energy to be recaptured on data structures that remain in scope.

Uncomputation makes the new remainder context function types a natural reification of the typing rules. Because the typing rules have leftovers, it is only natural that the function types do. In particular, the rule for typing functions *R-Fun* already uncomputes the formal argument parameter x in its premiss, so maximal potential energy can be reclaimed by having function types record that leftover. This reclaimation plays out in the application rule *R-App*, where sharing is used to restore leftover potential energy to the argument.

Note that not every rule performs uncomputation in this remainder context system. Firstly, only temporary variables are ever uncomputed. Thus, for instance, expressions that construct

[6]A similar flexible reasoning principle for Hoare-triples might be something like separation logic's frame rule [120].

R-Var

$$\frac{}{\Gamma, x : \tau \mid \Upsilon_x^{x,\mathtt{ret}}(\vec{a}) \vdash x : \tau \mid \vec{a}}$$

R-Let

$$\frac{\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c} \qquad \Gamma, x : \sigma \mid [x/\mathtt{ret}]\vec{c} \vdash e_2 : \tau \mid \vec{b}, \vec{d}}{\Gamma \mid \vec{a} \vdash \mathtt{let}\ x \ = \ e_1\ \mathtt{in}\ e_2 : \tau \mid \vec{b}}$$

R-Fun

$$\frac{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid 0 \cdot \vec{a}, [x/\mathtt{arg}]\vec{c} \vdash e : \sigma \mid 0 \cdot \vec{a}, [x/\mathtt{arg}]\vec{d}}{\Gamma \mid \vec{a}, \vec{b} \vdash \mathtt{fun}\ f\ x \ = \ e : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid \vec{a}, \vec{b}}$$

R-App

$$\frac{}{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{b}) \vdash f\ x : \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})}$$

R-Tick

$$\frac{\vec{a}_{\mathtt{c}} = \vec{b}_{\mathtt{c}} + r \qquad \forall i \neq \mathtt{c}.\, \vec{a}_i = \vec{b}_i}{\Gamma \mid \vec{a} \vdash \mathtt{tick}\{r\} : \mathbb{1} \mid \vec{b}}$$

R-Pair

$$\frac{}{\Gamma, x : \tau, y : \sigma \mid \Upsilon_x^{x,\mathtt{ret.1^{st}}}(\Upsilon_y^{y,\mathtt{ret.2^{nd}}}(\vec{a})) \vdash \langle x,\ y \rangle : \tau \otimes \sigma \mid \vec{a}}$$

R-CaseP

$$\frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \vec{a} \vdash e : \tau \mid \vec{b}}{\Gamma, x : \sigma \otimes \rho \mid \Upsilon_{x.1^{\mathtt{st}}}^{y,x.1^{\mathtt{st}}}(\Upsilon_{x.2^{\mathtt{nd}}}^{z,x.2^{\mathtt{nd}}}(\vec{a})) \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\ z \rangle \to e : \tau \mid \Upsilon_{x.1^{\mathtt{st}}}^{y,x.1^{\mathtt{st}}}(\Upsilon_{x.2^{\mathtt{nd}}}^{z,x.2^{\mathtt{nd}}}(\vec{b}))}$$

R-SumL

$$\frac{}{\Gamma, x : \tau \mid \Upsilon_x^{x,\mathtt{ret.l}}(\vec{a}) \vdash \mathtt{l}(x) : \tau \oplus \sigma \mid \vec{a}, \vec{b}}$$

R-SumR

$$\frac{}{\Gamma, x : \sigma \mid \Upsilon_x^{x,\mathtt{ret.r}}(\vec{a}) \vdash \mathtt{r}(x) : \tau \oplus \sigma \mid \vec{a}, \vec{b}}$$

R-CaseS

$$\frac{\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \vec{a}, \vec{b}, \vec{c'} \vdash e_1 : \tau \mid \vec{d}, \vec{e}, \vec{f'} \qquad \Gamma, x : \sigma \oplus \rho, z : \rho \mid \vec{a}, \vec{b'}, \vec{c} \vdash e_2 : \tau \mid \vec{d}, \vec{e'}, \vec{f}}{\Gamma, x : \sigma \oplus \rho \mid \vec{a}, \Upsilon_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{b}), \Upsilon_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{c}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid \vec{d}, \Upsilon_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{e}), \Upsilon_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{f})}$$

Figure 5.6: Remainder context typing rules 1

R-NIL

$$\frac{}{\Gamma \mid \vec{a} \vdash [\,] : L(\tau) \mid \vec{a}, \vec{b}}$$

R-CONS

$$\frac{}{\Gamma, x : \tau, y : L(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\lhd_{x',y'}^{\texttt{ret}}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

R-CASEL

$$\frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{y,z}^{x'}(\vec{c}, \vec{d})\end{array}}{\Gamma, x : L(\sigma) \mid \vec{a}, \curlyvee_x^{x,x'}(\vec{b}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}, \curlyvee_x^{x,x'}(\vec{d})}$$

R-LEAF

$$\frac{}{\Gamma \mid \vec{a} \vdash \texttt{leaf} : T(\tau) \mid \vec{a}, \vec{b}}$$

R-NODE

$$\frac{}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\lhd_{x',y',z'}^{\texttt{ret}}(\vec{a})))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

R-CASET

$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{x,y,z}^{t'}(\vec{c}, \vec{d})\end{array}}{\Gamma, t : T(\sigma) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}) \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d})}$$

Figure 5.7: Remainder context typing rules 2

```
1  (* lst:1,0 *) fun len lst = case lst of   (* lst:1,0 *)
2  (* []: 1,0 *)   | [] -> 0                 (* lst:1,0 *)
3  (* xs: 1,1 *)   | x::xs ->                (* lst:1,0 *)
4  (* xs: 1,0 *)     let () = tick{1} in     (* xs: 1,1 *)
5  (* xs: 1,0 *)     let subsoln = len xs in (* xs: 1,1 *)
6  (* xs: 1,1 *)     let () = tick{-1} in    (* xs: 1,1 *)
7  (* xs: 1,1 *)     1 + subsoln             (* xs: 1,1 *)
```

Figure 5.8: Code for `len` with energy comments

data structures do not simulate the subsequent destruction of those data structures. Then the only rule without uncomputation is the rule *R-Let*. This rule creates a binding for the variable x from the expression e_1 but never uncomputes x despite x having a clear scope. The problem here is that it is not clear how to properly invert the computation of e_1 to determine how to transform the remainder. Despite these exceptions, there is still enough uncomputation throughout the system to allow many functions to be meaningfully assigned leftover potential energy on their inputs, improving the AARA analysis.

Example 5.3.1. Recall the code for the length function `len` in Figure 5.2. Using the remainder context type system, `len` can be typed as $L^1(\mathbb{Z}) \to \mathbb{Z} \sim L^1(\mathbb{Z})$, which properly shows that all stack frames are returned.

This typing is witnessed by the amounts of energy annotated in Figure 5.8. For notational convenience, annotations belonging to the initial context are commented on the left of the code, and annotations belonging to the remainder context on the right. The initial context's comments go down as the computation progresses, and the remainder context's comments go up to reverse the process. Note that additionally, I abbreviate "x units of linear energy on list y and z units of free energy" as just $y : x, z$.

The key lines to look at in Figure 5.8 are as follows, starting with the initial annotations. At line 3, one unit of free energy is released through the pattern match of the argument `lst` and all energy is stored on `xs`. Subsequent lines do not change the amount of energy on `xs`, particularly line 5, which leaves `xs` unchanged due to the remainder matching the argument ($L^1(\mathbb{Z})$) in the type of `len`. For the remainder annotations, no changes are made to the energy going up from line 7 until line 3. This lack of change is because the action of ticks are not uncomputed; only data structures are. Then at line 3, the free energy and `xs` are repackaged back into the argument `lst`, leaving the argument holding the same linear energy it begain with.

Sequential calls to functions of this type (like in Figure 5.1) can reuse the resources left on the function argument.

5.4 Soundness

The soundness of AARA with remainder contexts is similar to Theorems 3.5.1 and 3.6.1 in that the initial potential energy of a context bounds the peak cost of evaluation, and the difference between initial and final energies bounds the net cost. However, the notion of final energy now takes into account the new presence of leftover potential energy in the context by adding it to the potential energy of returned value. This change results in Theorem 5.4.1, where the consequent concerning the net cost is changed from Theorems 3.5.1 and 3.6.1. (Also note that Theorem 5.4.1's proof takes into account potentially non-terminating execution.)

Theorem 5.4.1 (remainder context soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p$ *(diff. bounds net)*

Proof. The soundness proof proceeds by lexicographic induction over the derivation of the evaluation judgment followed by the typing judgment.

Because this proof has many cases, here are some points and patterns of interest:

- First this proof deals with non-syntax-directed rules so that each subsequent proof case only needs to cover syntax-directed options. Otherwise there would always be multiple cases for how to match up typing and evaluation judgments to a given expression.
- The tick and let expressions are the only expressions with interesting cost behaviour. These proof cases show the essence of how cost is paid and composed, respectively.
- The function application case makes nontrivial use of the well-formedness of function values to obtain a typing judgment for the function being applied. This judgment is necessary to apply the inductive hypothesis.
- The construction of every data structure has cost behaviour $(0, 0)$ and perfectly conserves potential, so the cost bounds follow directly. Specifically, the peak cost is always satisfied here because potential energy is nonnegative, and the net cost bound is always satisfied with a tight equality.
- The destruction of every data type (except for function types) has the same cost behaviour as its premisses. These cases all follow from applying the inductive hypothesis and then, similarly to the construction cases, using the facts that sharing and data structure destruction perfectly conserve potential.

Now each case in the induction is given in more detail:

R-Sub This case deals with the structural typing rule *R-Sub* so that future considerations of typing judgment derivation structure need not consider the case that the derivation ends with the application of *R-Sub*.

Suppose the last rule applied for the typing judgment is *R-Sub*.

$$\textsc{R-Sub}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a} \geq \vec{a'} \qquad \vec{b} \leq \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

Then the premisses of this rule hold by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'}$ to learn:

(1) $v : \tau$
(2) $\Phi(V : \Gamma \mid \vec{a'}) \geq p$
(3) $\Phi(V : \Gamma \mid \vec{a'}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from inequalities (2) and (3) by applying the pointwise monotonicity of potential energy (Lemma 3.4.5) alongside the pointwise annotation inequalities $\vec{a} \geq \vec{a'}$ and $\vec{b} \leq \vec{b'}$.

E-Nont Suppose the last rule applied for the evaluation judgment is *E-Nont*.

$$\textsc{E-Nont}\quad \frac{}{V \vdash e \Downarrow \bullet \mid (0, \infty)}$$

Then $p = 0$, $q = \infty$, and $v = \bullet$. Because $\bullet : \tau$ by *V-Nont*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because ∞ is greater than or equal to anything, the net cost bound also satisfied.

E-Tick Suppose the last rule applied for the evaluation judgment is *E-Tick*.

$$\textsc{E-Tick}\quad \frac{}{V \vdash \texttt{tick}\{r\} \Downarrow \langle\rangle \mid (\max(0, r), \max(0, -r))}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Tick}\quad \frac{\vec{a}_{\texttt{c}} = \vec{b}_{\texttt{c}} + r \qquad \forall i \neq \texttt{c}.\, \vec{a}_i = \vec{b}_i}{\Gamma \mid \vec{a} \vdash \texttt{tick}\{r\} : \mathbb{1} \mid \vec{b}}$$

The premisses of this rule hold by inversion.

Because $\langle\rangle : \mathbb{1}$ by *V-Unit*, the needed well-formedness judgment holds.

Then the following inequalities confirm the peak cost bound:

$$\begin{aligned} \Phi(V : \Gamma \mid \vec{a}) &\geq \vec{a}_{\texttt{c}} && \textit{def} \\ &= \vec{b}_{\texttt{c}} + r && \vec{a}_{\texttt{c}} = \vec{b}_{\texttt{c}} + r \\ &\geq \max(0, r) && \vec{a}, \vec{b} \geq 0 \end{aligned}$$

Finally, let $s = \sum_{x \in \texttt{dom}(\Gamma)} \Phi(V(x) : \Gamma(x) \mid \lambda i.\, \vec{a}_{x.i})$. Then the following inequalities confirm the net cost bound.

$$\begin{aligned}
\Phi(V : \Gamma \mid \vec{a}) + \max(0, -r) &= \vec{a}_{\mathsf{c}} + s + \max(0, -r) && \textit{def} \\
&= \vec{b}_{\mathsf{c}} + r + s + \max(0, -r) && \vec{a}_{\mathsf{c}} = \vec{b}_{\mathsf{c}} + r \\
&= \vec{b}_{\mathsf{c}} + s + \max(0, r) && \textit{algebra} \\
&= \vec{b}_{\mathsf{c}} + s + \Phi(\langle\rangle : \mathbb{1} \mid \cdot) + \max(0, r) && 0 \textit{ potential} \\
&= \Phi((V, \texttt{ret} \mapsto \langle\rangle) : (\Gamma, \texttt{ret} : \mathbb{1}) \mid \vec{b}) + \max(0, r) && \textit{def}
\end{aligned}$$

E-Var Suppose the last rule applied for the evaluation judgment is *E-Var*

$$\textsc{E-Var}\quad \frac{}{V, x \mapsto v \vdash x \Downarrow v \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Var}\quad \frac{}{\Gamma, x : \tau \mid \curlyvee_x^{x,\texttt{ret}}(\vec{a}) \vdash x : \tau \mid \vec{a}}$$

Then $p = q = 0$ and $(V, x \mapsto v) : (\Gamma, x : \tau)$. Because $v : \tau$ follows from inverting *V-Context*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because sharing perfectly conserves potential (Lemma 3.4.1), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \curlyvee_x^{x,\texttt{ret}}(\vec{a})) = \Phi((V, x \mapsto v, \texttt{ret} \mapsto v) : (\Gamma, x : \tau, \texttt{ret} : \tau) \mid \vec{a})$$

E-Let Suppose the last rule applied for the evaluation judgment is *E-Let*.

$$\textsc{E-Let}\quad \frac{V \vdash e_1 \Downarrow v' \mid (p, q) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)}{V \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v \mid (p + \max(0, r - q), s + \max(0, q - r))}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Let}\quad \frac{\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c} \qquad \Gamma, x : \sigma \mid [x/\texttt{ret}]\vec{c} \vdash e_2 : \tau \mid \vec{b}, \vec{d}}{\Gamma \mid \vec{a} \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau \mid \vec{b}}$$

The premisses of both of these rules hold by inversion.

Because $V : \Gamma$ holds by assumption, the inductive hypothesis can be applied with the judgments $V \vdash e_1 \Downarrow v' \mid (p, q)$ and $\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c}$ to learn:

(1) $v' : \sigma$

(2) $\Phi(V : \Gamma \mid \vec{a}) \geq p$

(3) $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p$

Because $v' : \sigma$ holds as (1) from the previous induction and both $V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)$ and $\Gamma, x : \sigma \mid [x/\texttt{ret}]\vec{c} \vdash e_2 : \tau \mid \vec{b}, \vec{d}$ hold from inversion, the inductive hypothesis can be applied again to learn:

(4) $v : \tau$
(5) $\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) \geq r$
(6) $\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + s \geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma) \mid \vec{b}, \vec{d}) + r$

The well-formedness judgment (4) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, proceed by cases on whether $q \geq r$.

If $q \geq r$, then the cost behaviour to consider is $(p, s + (q - r))$. Then (2) confirms the peak cost bound, and the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) + s + (q - r) \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p + s - r && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + p + s - r && \textit{relabelling} \\
&\geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma, \texttt{ret} : \tau) \mid \vec{b}, \vec{d}) + p && (6) \\
&\geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p && \textit{energy} \geq 0
\end{aligned}$$

If $q < r$, then the cost behaviour to consider is $(p + (r - q), s)$, and $q \neq \infty$ (so can be subtracted). Then the following inequalities confirm the peak cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p - q && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + p - q && \textit{relabelling} \\
&\geq p + (r - q) && (5)
\end{aligned}$$

And finally, the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) + s \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + s + p - q && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + s + p - q && \textit{relabelling} \\
&\geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma, \texttt{ret} : \tau) \mid \vec{b}, \vec{d}) + p + (r - q) && (6) \\
&\geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p + (r - q) && \textit{energy} \geq 0
\end{aligned}$$

E-Fun Suppose the last rule applied for the evaluation judgment is *E-Fun*.

$$\textsc{E-Fun}\quad \frac{}{V \vdash \texttt{fun}\ f\ x\ =\ e \Downarrow \texttt{C}(V;\ f,\ x.\, e) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Fun}\quad \frac{\Gamma, x : \tau, f : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma \mid 0 \cdot \vec{a}, [x/\texttt{arg}]\vec{c} \vdash e : \sigma \mid 0 \cdot \vec{a}, [x/\texttt{arg}]\vec{d}}{\Gamma \mid \vec{a}, \vec{b} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma \mid \vec{a}, \vec{b}}$$

The assumed typing judgment for the expression being evaluated therefore takes the form of this rule's conclusion.

Because $\mathtt{C}(V;\ f,\ x.\,e) : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma$ follows from *V-Fun* and the assumed typing judgment, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because the initial and remainder annotations are identical and functions carry no potential energy (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \vec{a}, \vec{b}) = \Phi((V, \texttt{ret} \mapsto \mathtt{C}(V;\ f,\ x.\,e)) : (\Gamma, \texttt{ret} : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma) \mid \vec{a}, \vec{b})$$

E-App Suppose the last rule applied for the evaluation judgment is *E-App*.

$$\textsc{E-App}\quad \frac{V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e) \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e) \vdash f\ x \Downarrow v \mid (p, q)}$$

Then this rule's premiss holds by inversion and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-App}\quad \frac{}{\Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \Upsilon_x^{x,\texttt{arg}}(\vec{a}, \vec{b}) \vdash f\ x : \sigma \mid \Upsilon_x^{x,\texttt{arg}}(\vec{a}, \vec{c})}$$

Because $(V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma)$ by assumption, the rule *V-Context* can be inverted to learn $\mathtt{C}(V';\ g,\ y.\,e) : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma$. Then further, the rule *V-Fun* can be inverted to learn that this function body can be typed in some context Γ' where $V' : \Gamma'$. Using *V-Context*, one can then use this well-formedness judgment to derive

$$(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma)$$

Now inspect the derivation of the type of the function closure's body. Only structural rules (like *R-Sub*) and *R-Fun* can conclude a typing derivation for a function, and the application a structural rule itself requires another typing derivation for the same function. Thus it can be shown by induction that the typing derivation must conclude by the rule *R-Fun* followed by some number of uses of structural rules. The typing derivation therefore contains the following rule application:

$$\textsc{R-Fun}\quad \frac{\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{b} \vdash e : \sigma \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{c}}{\Gamma' \mid \vec{a'}, \vec{b'} \vdash \texttt{fun}\ g\ y\ =\ e : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \vec{a'}, \vec{b'}}$$

This rule's premiss holds by inversion.

Each of the following judgments have now been found:

- $V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (p, q)$
- $(V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$
- $\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{b} \vdash e : \sigma \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{c}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \sigma$

(2) $\Phi((V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{b}) \geq p$

(3) $\Phi((V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{b}) + q$
$\geq \Phi((V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e), \texttt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \texttt{ret} : \sigma) \mid 0 \cdot \vec{a'}, [y/\texttt{arg}]\vec{c}) + p$

The well-formedness judgment (1) $v : \sigma$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, first simplify inequalities (2) and (3) into inequalities (4) and (5), respectively, by removing bindings that carry no potential energy:

(4) $\Phi((y \mapsto v') : (y : \tau) \mid [y/\texttt{arg}]\vec{b}) \geq p$

(5) $\Phi((y \mapsto v') : (y : \tau) \mid [y/\texttt{arg}]\vec{b}) + q \geq \Phi((y \mapsto v', \texttt{ret} \mapsto v) : (y : \tau, \texttt{ret} : \sigma) \mid [y/\texttt{arg}]\vec{c}) + p$

Now let $r = \Phi((V, f \mapsto \texttt{C}(V';\ g, y.\, e)) : (\Gamma, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \vec{a})$. Then the following inequalities confirm the peak cost bound:

$$\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \curlyvee_x^{x,\texttt{arg}}(\vec{a}, \vec{b})) && \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \curlyvee_x^{x,\texttt{arg}}(\vec{a}, \vec{b})) && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((\texttt{arg} \mapsto v') : (\texttt{arg} : \tau) \mid \vec{b}) && \textit{Lemma 3.4.1} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((y \mapsto v') : (y : \tau) \mid [y/\texttt{arg}]\vec{b}) && \textit{relabelling} \\
&\geq \Phi((y \mapsto v') : (y : \tau) \mid [y/\texttt{arg}]\vec{b}) && \textit{algebra} \\
&\geq p && (4)
\end{aligned}$$

And the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g, y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \curlyvee_x^{x,\mathtt{arg}}(\vec{a}, \vec{b})) + q \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \curlyvee_x^{x,\mathtt{arg}}(\vec{a}, \vec{b})) + q && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((\mathtt{arg} \mapsto v') : (\mathtt{arg} : \tau) \mid \vec{b}) + q && \textit{Lemma 3.4.1} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + q && \textit{relabelling} \\
&\geq r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid [y/\mathtt{arg}]\vec{c}) + p && (5) \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \vec{a}) + \Phi((\mathtt{arg} \mapsto v', \mathtt{ret} \mapsto v) : (\mathtt{arg} : \tau, \mathtt{ret} : \sigma) \mid \vec{c}) + p && \textit{relabelling} \\
&= r + \Phi((x \mapsto v', \mathtt{ret} \mapsto v) : (x : \tau, \mathtt{ret} : \sigma) \mid \curlyvee_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})) + p && \textit{Lemma 3.4.1} \\
&= \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(C';\ g, y.\, e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid \curlyvee_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})) + p && \textit{def}
\end{aligned}$$

E-Pair Suppose the last rule applied for the evaluation judgment is *E-Pair*.

$$\textsc{E-Pair}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash \langle x,\ y\rangle \Downarrow \langle v_1,\ v_2\rangle \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Pair}\quad \frac{}{\Gamma, x : \tau, y : \sigma \mid \curlyvee_x^{x,\mathtt{ret.1^{st}}}(\curlyvee_y^{y,\mathtt{ret.2^{nd}}}(\vec{a})) \vdash \langle x,\ y\rangle : \tau \otimes \sigma \mid\ \vec{a}}$$

Because $\langle v_1,\ v_2\rangle : \tau \otimes \sigma$ follows from *V-Pair* and the assumed well-formedness judgment $(V, x_1 \mapsto v_1, x_2 \mapsto v_2) : (\Gamma, x : \tau, y : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a pair is the sum of its parts' (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\begin{aligned}
&\Phi(V, x \mapsto v_1, y \mapsto v_2 : \Gamma, x : \tau, y : \sigma \mid \curlyvee_x^{x,\mathtt{ret.1^{st}}}(\curlyvee_y^{y,\mathtt{ret.2^{nd}}}(\vec{a}))) \\
&= \Phi(V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto \langle v_1,\ v_2\rangle : \Gamma, x : \tau, y : \sigma, \mathtt{ret} : \tau \otimes \sigma \mid \vec{a})
\end{aligned}$$

E-CaseP Suppose the last rule applied for the evaluation judgment is *E-CaseP*.

$$\textsc{E-CaseP}\quad \frac{V, x \mapsto \langle v_1,\ v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto \langle v_1,\ v_2\rangle \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\ z\rangle \to e \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-CaseP}\quad \frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \vec{a} \vdash e : \tau \mid\ \vec{b}}{\Gamma, x : \sigma \otimes \rho \mid \curlyvee_{x.1^{\mathrm{st}}}^{y,x.1^{\mathrm{st}}}(\curlyvee_{x.2^{\mathrm{nd}}}^{z,x.2^{\mathrm{nd}}}(\vec{a})) \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\ z\rangle \to e : \tau \mid\ \curlyvee_{x.1^{\mathrm{st}}}^{y,x.1^{\mathrm{st}}}(\curlyvee_{x.2^{\mathrm{nd}}}^{z,x.2^{\mathrm{nd}}}(\vec{b}))}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho)$ by assumption, the rule *V-Context* can be inverted to learn $\langle v_1,\, v_2\rangle : \sigma \otimes \rho$. Then further, the rule *V-Pair* can be inverted to learn both $v_1 : \sigma$ and $v_2 : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$$

Each of the following judgments has now been found:

- $V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)$
- $(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$
- $\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \vec{a} \vdash e : \tau \mid \vec{b}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \vec{a}) \geq p$

(3) $\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \vec{a}) + q$
$\geq \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid \vec{b}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a pair is the sum of its parts' (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \vec{a})$$
$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho) \mid \curlyvee_{x.\texttt{1}^{\texttt{st}}}^{y, x.\texttt{1}^{\texttt{st}}}(\curlyvee_{x.\texttt{2}^{\texttt{nd}}}^{z, x.\texttt{2}^{\texttt{nd}}}(\vec{a})))$$

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid \vec{b})$$
$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, \texttt{ret} : \tau) \mid \curlyvee_{x.\texttt{1}^{\texttt{st}}}^{y, x.\texttt{1}^{\texttt{st}}}(\curlyvee_{x.\texttt{2}^{\texttt{nd}}}^{z, x.\texttt{2}^{\texttt{nd}}}(\vec{b})))$$

E-SumL Suppose the last rule applied for the evaluation judgment is *E-SumL*.

$$\textsc{E-SumL}\quad \frac{}{V, x \mapsto v \vdash \texttt{l}(x) \Downarrow \texttt{l}(v) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-SumL}\quad \frac{}{\Gamma, x : \tau \mid \curlyvee_{x}^{x, \texttt{ret.l}}(\vec{a}) \vdash \texttt{l}(x) : \tau \oplus \sigma \mid \vec{a}, \vec{b}}$$

Because $\texttt{l}(v) : \tau \oplus \sigma$ follows from *V-SumL* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \tau)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant is that of its tagged value (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\begin{aligned}&\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \curlyvee_x^{x,\texttt{ret.l}}(\vec{a}))\\ &= \Phi((V, x \mapsto v, \texttt{ret} \mapsto \texttt{l}(v)) : (\Gamma, x : \tau, \texttt{ret} \mapsto \tau \oplus \sigma) \mid \vec{a}, \vec{b})\end{aligned}$$

E-SumR Suppose the last rule applied for the evaluation judgment is *E-SumR*.

$$\textsc{E-SumR}\quad \frac{}{V, x \mapsto v \vdash \texttt{r}(x) \Downarrow \texttt{r}(v) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-SumR}\quad \frac{}{\Gamma, x : \sigma \mid \curlyvee_x^{x,\texttt{ret.r}}(\vec{a}) \vdash \texttt{r}(x) : \tau \oplus \sigma \mid \vec{a}, \vec{b}}$$

Because $\texttt{r}(v) : \tau \oplus \sigma$ follows from *V-SumR* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant is that of its tagged value (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\begin{aligned}&\Phi((V, x \mapsto v) : (\Gamma, x : \sigma) \mid \curlyvee_x^{x,\texttt{ret.r}}(\vec{a}))\\ &= \Phi((V, x \mapsto v, \texttt{ret} \mapsto \texttt{r}(v)) : (\Gamma, x : \sigma, \texttt{ret} \mapsto \tau \oplus \sigma) \mid \vec{a}, \vec{b})\end{aligned}$$

E-CaseS-L Suppose the last rule applied for the evaluation judgment is *E-CaseS-L*.

$$\textsc{E-CaseS-L}\quad \frac{V, x \mapsto \texttt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)}{V, x \mapsto \texttt{l}(v') \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-CaseS}\quad \frac{\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \vec{a}, \vec{b}, \vec{c'} \vdash e_1 : \tau \mid \vec{d}, \vec{e}, \vec{f'} \qquad \Gamma, x : \sigma \oplus \rho, z : \rho \mid \vec{a}, \vec{b'}, \vec{c} \vdash e_2 : \tau \mid \vec{d}, \vec{e'}, \vec{f}}{\Gamma, x : \sigma \oplus \rho \mid \vec{a}, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(\vec{b}), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(\vec{c}) \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 : \tau \mid \vec{d}, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(\vec{e}), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(\vec{f})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \texttt{l}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \sigma$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$
- $\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \vec{a}, \vec{b}, \vec{c'} \vdash e_1 : \tau \mid \vec{d}, \vec{e}, \vec{f'}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \vec{a}, \vec{b}, \vec{c'}) \geq p$
(3) $\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \vec{a}, \vec{b}, \vec{c'}) + q$
$\geq \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid \vec{d}, \vec{e}, \vec{f'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant is that of its tagged value (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \vec{a}, \vec{b}, \vec{c'})$$

$$= \Phi((V, x \mapsto \mathtt{l}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \vec{a}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{b}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{c}))$$

$$\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid \vec{d}, \vec{e}, \vec{f'})$$
$$= \Phi((V, x \mapsto \mathtt{l}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \mathtt{ret} : \tau) \mid \vec{d}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{e}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{f}))$$

E-CaseS-R Suppose the last rule applied for the evaluation judgment is *E-CaseS-R*.

E-CaseS-R
$$\frac{V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_2 \Downarrow v \mid (p, q)}{V, x_s \mapsto \mathtt{r}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

R-CaseS
$$\frac{\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \vec{a}, \vec{b}, \vec{c'} \vdash e_1 : \tau \mid \vec{d}, \vec{e}, \vec{f'} \qquad \Gamma, x : \sigma \oplus \rho, z : \rho \mid \vec{a}, \vec{b'}, \vec{c} \vdash e_2 : \tau \mid \vec{d}, \vec{e'}, \vec{f}}{\Gamma, x : \sigma \oplus \rho \mid \vec{a}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{b}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{c}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid \vec{d}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{e}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{f})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$

- $\Gamma, x : \sigma \oplus \rho, z : \rho \mid \vec{a}, \vec{b'}, \vec{c} \vdash e_2 : \tau \mid \vec{d}, \vec{e'}, \vec{f}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \vec{a}, \vec{b'}, \vec{c}) \geq p$

(3) $\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \vec{a}, \vec{b'}, \vec{c}) + q$
$\geq \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid \vec{d}, \vec{e'}, \vec{f}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant is that of its tagged value (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \vec{a}, \vec{b'}, \vec{c})$$

$$= \Phi((V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \vec{a}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{b}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{c}))$$

$$\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid \vec{d}, \vec{e'}, \vec{f})$$

$$= \Phi((V, x \mapsto \mathtt{r}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \mathtt{ret} : \tau) \mid \vec{d}, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(\vec{e}), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(\vec{f}))$$

E-Nil Suppose the last rule applied for the evaluation judgment is *E-Nil*.

$$\textsc{E-Nil}\quad \frac{}{V \vdash [\,] \Downarrow [\,] \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Nil}\quad \frac{}{\Gamma \mid \vec{a} \vdash [\,] : L(\tau) \mid \vec{a}, \vec{b}}$$

Because $[\,] : L(\tau)$ follows from *V-Nil*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because the initial and remainder annotations are identical except for the empty list annotations $\vec{b}$ and empty lists carry no energy regardless of annotation (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \mathtt{ret} \mapsto [\,]) : (\Gamma, \mathtt{ret} : L(\tau)) \mid \vec{a}, \vec{b})$$

E-Cons Suppose the last rule applied for the evaluation judgment is *E-Cons*.

$$\textsc{E-Cons}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-Cons}\quad \frac{}{\Gamma, x : \tau, y : L(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\lhd_{x',y'}^{\texttt{ret}}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

Because $v_1 :: v_2 : L(\tau)$ follows from *V-Cons* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 3.4.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\lhd_{x',y'}^{\texttt{ret}}(\vec{a}))))$$
$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, \texttt{ret} \mapsto v_1 :: v_2) : (\Gamma, x : \tau, y : L(\tau), \texttt{ret} : L(\tau)) \mid \vec{a})$$

E-CaseL-Nil Suppose the last rule applied for the evaluation judgment is *E-CaseL-Nil*.

$$\textsc{E-CaseL-Nil}\quad \frac{V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)}{V, x \mapsto [\,] \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-CaseL}\quad \frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{y,z}^{x'}(\vec{c}, \vec{d})\end{array}}{\Gamma, x : L(\sigma) \mid \vec{a}, \curlyvee_x^{x,x'}(\vec{b}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}, \curlyvee_x^{x,x'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)$
- $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$
- $\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \vec{b'}) \geq p$

(3) $\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \vec{b'}) + q$
$\geq \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \vec{d'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because empty lists carry no potential energy regardless of annotation (Figure 3.9) both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \vec{b'}) = \Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \curlyvee_x^{x,x'}(\vec{b}))$$

$$\Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \vec{d'})$$
$$= \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee_x^{x,x'}(\vec{d}))$$

E-CaseL-Cons Suppose the last rule applied for the evaluation judgment is *E-CaseL-Cons*.

$$\textsc{E-CaseL-Cons}\quad \frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)}{V, x \mapsto v_1 :: v_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-CaseL}\quad \frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{y,z}^{x'}(\vec{c}, \vec{d})\end{array}}{\Gamma, x : L(\sigma) \mid \vec{a}, \curlyvee_x^{x,x'}(\vec{b}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}, \curlyvee_x^{x,x'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 : L(\sigma)$. Then further, the rule *V-Cons* can be inverted to learn both $v_1 : \sigma$ and $v_2 : L(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$$

Each of the following judgments has now been found:

- $V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$
- $\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{y,z}^{x'}(\vec{c}, \vec{d})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b})) \geq p$
(3) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \lhd_{y,z}^{x'}(\vec{a}, \vec{b})) + q$
$\geq \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \lhd_{y,z}^{x'}(\vec{c}, \vec{d})) + p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 3.4.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \lhd^{x'}_{y,z}(\vec{a}, \vec{b}))$$

$$= \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \curlyvee^{x,x'}_{x}(\vec{b}))$$

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \lhd^{x'}_{y,z}(\vec{c}, \vec{d}))$$

$$= \Phi((V, x \mapsto v_1 :: v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee^{x,x'}_{x}(\vec{d}))$$

E-Leaf Suppose the last rule applied for the evaluation judgment is *E-Leaf*.

$$\frac{\textsc{E-Leaf}}{V \vdash \texttt{leaf} \Downarrow \texttt{leaf} \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{R-Leaf}}{\Gamma \mid \vec{a} \vdash \texttt{leaf} : T(\tau) \mid \vec{a}, \vec{b}}$$

Because $\texttt{leaf} : T(\tau)$ follows from *V-Leaf*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because the initial and remainder annotations are identical except for the leaf annotations $\vec{b}$ and leaves carry no energy regardless of annotation (Figure 3.9), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \texttt{ret} \mapsto \texttt{leaf}) : (\Gamma, \texttt{ret} : T(\tau)) \mid \vec{a}, \vec{b})$$

E-Node Suppose the last rule applied for the evaluation judgment is *E-Node*.

$$\frac{\textsc{E-Node}}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\ y,\ z) \Downarrow \texttt{node}(v_1,\ v_2,\ v_3) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{R-Node}}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee^{x,x'}_{x}(\curlyvee^{y,y'}_{y}(\curlyvee^{z,z'}_{z}(\lhd^{\texttt{ret}}_{x',y',z'}(\vec{a})))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

Because $\texttt{node}(v_1,\, v_2,\, v_3) : T(\tau)$ follows from *V-Node* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 3.4.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\lhd^{\texttt{ret}}_{x',y',z'}(\vec{a})))))$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto \texttt{node}(v_1,\, v_2,\, v_3)) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau), \texttt{ret} : T(\tau)) \mid \vec{a})$$

E-CaseT-Leaf Suppose the last rule applied for the evaluation judgment is *E-CaseT-Leaf*.

$$\textsc{E-CaseT-Leaf}\quad \frac{V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{leaf} \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\, y,\, z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-CaseT}\quad \frac{\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \qquad \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \lhd^{t'}_{x,y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd^{t'}_{x,y,z}(\vec{c}, \vec{d})}{\Gamma, t : T(\sigma) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\, y,\, z) \to e_2 : \tau \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$
- $\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \vec{b'}) \geq p$
(3) $\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \vec{b'}) + q$
$\geq \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \vec{d'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because leaves carry no potential energy regardless of annotation (Figure 3.9) both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \vec{b'}) = \Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}))$$

$$\Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \vec{d'})$$
$$= \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d}))$$

E-CaseT-Node Suppose the last rule applied for the evaluation judgment is *E-CaseT-Node*.

E-CASET-NODE
$$\frac{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

R-CASET
$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{x,y,z}^{t'}(\vec{c}, \vec{d})\end{array}}{\Gamma, t : T(\sigma) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 v_3 : T(\sigma)$. Then further, the rule *V-Node* can be inverted to learn all of $v_1 : T(\sigma)$, $v_2 : \sigma$, and $v_3 : T(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$$

Each of the following judgments has now been found:

- $V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$
- $\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \lhd_{x,y,z}^{t'}(\vec{c}, \vec{d})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b})) \geq p$

(3) $\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b})) + q$
$\geq \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid \lhd_{x,y,z}^{t'}(\vec{c}, \vec{d}))$
$+ p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 3.4.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \lhd_{x,y,z}^{t'}(\vec{a}, \vec{b}))$$

$$= \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}))$$

$$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid \lhd^{t'}_{x,y,z}(\vec{c}, \vec{d}))$$

$$= \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee^{t,t'}_{t}(\vec{d}))$$

□

5.5 Automation

One of the key benefits of remainder contexts is their effect on AARA-style type inference. Because remainder contexts only require linear relations between annotations, remainder context type inference can be automated via linear program in much the same way as described in Section 3.7. However, remainder contexts also make the type inference simpler. This simplicity arises because remainder contexts remove the need for weakening and contraction rules, and these rules require a variable-usage analysis during typing inference. Thus, type inference for remainder contexts is a simple two step process:

1. basic type inference
2. collect and solve linear contraints

While basic type inference is the theoretical complexity bottleneck as described in Section 3.7, the linear constraint solving is the more practically relevant bottleneck. For AARA as described in Chapter 3, linear programming takes polynomial time in the size of the source code. This circumstance is still the case for remainder contexts, even though remainder contexts generate more constraints than Chapter 3's AARA. At worst, the number of constraints is only increased by a factor of around 2 due to mirroring the relations of the initial and remainder annotations in rules that perform uncomputation. As a result, remainder contexts still induce polynomial time constraint solving, and therefore remainder contexts continue to enable efficient type inference.

5.6 Non-Recursive Functions

Using remainder contexts, there is one additional trick one can consider: in certain circumstances, *non-recursive* function closures can be allowed to capture potential energy from their environment. Right now, AARA requires that function closures capture no potential, as can be gleaned from the 0 scalar in *R-Fun* and the function energy definition in Figure 3.9. The result of relaxing this requirement is that some functions can be typed as if they have a lower peak cost. This typing flexibility has some utility and might be of use for future research. Thus I lay out the details of such typing in this section. However, this flexibility is also an unecessary complication for the remainder of this thesis, and thus I limit its description to this section.

This new typing power is useful for analyzing code like Figure 5.9. As it stands, the typing rules provided so far can type the application `f` b as $\langle \mathbb{1};\ 2\rangle \to \langle \mathbb{1};\ 2\rangle \sim \mathbb{1}$. This type signals that 2 additional units of energy are needed to run the function. Using the new rules provided in this section, the application `f` b can instead be typed as $\langle \mathbb{1};\ 0\rangle \to \langle \mathbb{1};\ 0\rangle \sim \mathbb{1}$, which signals

```
fun f b = case b of
  | inl(_) -> fun g _ = tick{0}
  | inr(_) -> let _ = tick{-2} in
    fun h _ = let _ = tick{2} in
      tick{-2}

fun getLen _ = len lst
```

Figure 5.9: Code that can make use of energy-capturing non-recursive function typing

$$\textsc{R-NonRecFun}\quad \frac{\Gamma, x : \tau \mid \Upsilon^{y;\mathtt{c}}_{\mathtt{c}}([y/\mathtt{c}]\vec{b}, [x/\mathtt{arg}]\vec{c}) \vdash e : \sigma \mid \Upsilon^{y;\mathtt{c}}_{\mathtt{c}}([y/\mathtt{c}]\vec{b}, [x/\mathtt{arg}]\vec{d})}{\Gamma \mid \vec{a} + \vec{b} \vdash \mathtt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{c}\mid\vec{d}} \sigma \mid \vec{a}}$$

$$\Phi(\mathtt{C}(V;\ f,\ x.\,e) : \tau \xrightarrow{\vec{c}\mid\vec{d}} \sigma \mid \vec{a})$$

$$= \begin{cases} 0 & f \in e \\ \min_{\vec{b}} \Phi(V : \Gamma \mid \vec{b})\ \ s.t.\ \ \Gamma, x : \tau \mid \Upsilon^{y;\mathtt{c}}_{\mathtt{c}}([y/\mathtt{c}]\vec{b}, [x/\mathtt{arg}]\vec{c}) \vdash e : \sigma \mid \Upsilon^{y;\mathtt{c}}_{\mathtt{c}}([y/\mathtt{c}]\vec{b}, [x/\mathtt{arg}]\vec{d}) & f \notin e \end{cases}$$

Figure 5.10: Typing rule and energy definition for non-recursive functions

that *no* additional energy is needed to run the function. Then assuming `len` has a type like $L^1(\mathbb{Z}) \to \mathbb{Z} \sim L^1(\mathbb{Z})$ and `lst` has a type like $L^1(\mathbb{Z})$, the function `getLen` cannot be typed at all by the rules provided so far. However, with the new rules provided in this section, `getLen` could be typed as $\mathbb{1} \to \mathbb{Z} \sim \mathbb{1}$. In these ways, the rules of this section allow for tighter and more robust AARA-style cost analysis in the presence of non-recursive functions.

To unlock this power in AARA, the typing rules and energy definition need some small updates. The new typing rule and energy definition for non-recursive functions are given in Figure 5.10. Each of these formalisms are explained in the following paragraphs.

The typing rule *R-NonRecFun* types a non-recursive function by first splitting the entire initial context annotation pointwise into $\vec{a}$ and $\vec{b}$ and then using the annotation $\vec{b}$ to help type the function body. The caveat is that the function body must leave all of $\vec{b}$ as remainder. This set up simulates setting aside a personal store of *fully reusable* resources annotated by $\vec{b}$ for the function. With these resources set aside, such a function can be run arbitrarily often without requiring new resources. This set up is a more permissive alternative to requiring that zero energy is captured from the environment while still ensuring that the closure does not exhaust its resources over many function executions.

The energy of a function then reflects that some energy might be set aside in the function closure. For recursive functions f (i.e., those where f occurs in the body e), the energy is still 0. Alternatively, for non-recursive functions, this energy is equal to the minimum potential energy of the closure $\Phi(V : \Gamma \mid \vec{b})$ over any annotation $\vec{b} \geq 0$ such that the function body can still be

```
  fun f x = let _ = tick {-1} in
    let ret = f x in
    let _ = tick{1} in
    ret
```

Figure 5.11: Problem case for recursive functions capturing energy

typed. This amount of energy is necessary to cover the peak costs of running such functions.

As a result of these changes, the interpretation of function types is slightly more nuanced than before. The function type $\tau \xrightarrow{\vec{a}|\vec{b}} \sigma$ previously indicated a peak execution cost bound of $\Phi((\texttt{arg} \mapsto v) : (\texttt{arg} : \tau) \mid \vec{a})$ to run the function on input v. In this new setting, this interpretation is still essentially the same, except that the bounded cost must be understood as the peak *marginal* execution cost. This intepretation takes into account that some cost might have been "prepaid" through energy captured in the function closure.

One might wonder why this section restricts its upgrades to non-recursive functions. This restriction is imposed because the execution of a recursive function closure $\texttt{C}(V;\ f,\ x.\,e)$ creates a program context in which *both* V and the closure itself are live. The problem is that this circumstance unsoundly duplicates energy since it allows the program to access to V's captured energy twice, both through the closure and through V itself. This duplication can be abused through code like Figure 5.11 where capturing just 1 unit of energy would appear to successfully prepay an infinitely large peak cost. Thus, without further adjustment, the ideas of this section are restricted to non-recursive functions.

I now conclude this section by providing the missing soundness proof cases for the new typing rule and energy definition. As a result, Theorem 5.4.1 continues to hold in their presence. The key new ideas used in these proof cases concern carefully accounting for the energy of functions, which now may be nonzero.

Non-Recursive E-Fun Suppose the last rule applied for the evaluation judgment is *E-Fun*.

$$\textsc{E-Fun}\quad \frac{}{V \vdash \texttt{fun}\ f\ x\ =\ e \Downarrow \texttt{C}(V;\ f,\ x.\,e) \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-NonRecFun}\quad \frac{\Gamma, x : \tau \mid \Upsilon^{y,\texttt{c}}_{\texttt{c}}([y/\texttt{c}]\vec{b}, [x/\texttt{arg}]\vec{c}) \vdash e : \sigma \mid \Upsilon^{y,\texttt{c}}_{\texttt{c}}([y/\texttt{c}]\vec{b}, [x/\texttt{arg}]\vec{d})}{\Gamma \mid \vec{a} + \vec{b} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid \vec{a}}$$

This rule's premiss holds by inversion and the assumed typing judgment for the expression being evaluated therefore takes the form of this rule's conclusion.

Because $\texttt{C}(V;\ f,\ x.\,e) : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma$ follows from *V-Fun* and the assumed typing judgment, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, the following inequalities confirm that the net cost bound is satisfied:

$$\begin{aligned}
\Phi(V : \Gamma \mid \vec{a} + \vec{b}) &= \Phi(V : \Gamma \mid \vec{a}) + \Phi(V : \Gamma \mid \vec{b}) && \textit{Lemma } 3.4.6 \\
&\geq \Phi(V : \Gamma \mid \vec{a}) + \Phi(\mathtt{C}(V;\ f,\ x.\, e) : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid \cdot) && \textit{def} \\
&= \Phi((V, \mathtt{ret} \mapsto \mathtt{C}(V;\ f,\ x.\, e)) : (\Gamma, \mathtt{ret} : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma) \mid \vec{a}) && \textit{def}
\end{aligned}$$

Non-Recursive E-App Suppose the last rule applied for the evaluation judgment is *E-App*.

$$\textsc{E-App}\quad \frac{V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash f\ x \Downarrow v \mid (p, q)}$$

Then this rule's premiss holds by inversion and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{R-App}\quad \frac{}{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{b}) \vdash f\ x : \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})}$$

Because $(V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$ by assumption, the rule *V-Context* can be inverted to learn $\mathtt{C}(V';\ g,\ y.\, e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma$. Then further, the rule *V-Fun* can be inverted to learn that this function body can be typed in some context Γ' where $V' : \Gamma'$. Using *V-Context*, one can then use this well-formedness judgment to derive

$$(V', y \mapsto v') : (\Gamma', y : \tau)$$

Now inspect the derivation of the type of the function closure's body. Only the rules *R-Fun*, *R-Sub*, and now *R-NonRecFun* can conclude a the typing derivation for a function, and the application *R-Sub* itself requires another typing derivation for the same function. Thus it can be shown by induction that the typing derivation must conclude by the rule *R-Fun* or *R-NonRecFun* followed by some number of uses of the rule *R-Sub*. As the case for *R-Fun* has already been considered, all that remains to be considered is that the typing derivation contains the following rule application:

$$\textsc{R-NonRecFun}\quad \frac{\Gamma', y : \tau \mid \Upsilon_{\mathtt{c}}^{z,\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid \Upsilon_{\mathtt{c}}^{z,\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{c})}{\Gamma' \mid \vec{a'} + \vec{b'} \vdash \mathtt{fun}\ g\ y\ =\ e : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \vec{a'}}$$

Because any derivation exist at all that uses this rule in this way, choose the derivation that minimizes the value of $\Phi(V' : \Gamma' \mid \vec{b'})$ without loss of generality. This rule's premiss holds by inversion.

Because *R-NonRecFun* types the expression e, the function g cannot be used in e. Thus the for g can be weakened away in the *E-App*'s premiss to obtain

$$V', y \mapsto v' \vdash e \Downarrow v \mid (p, q)$$

Each of the following judgments have now been found:

- $V', y \mapsto v' \vdash e \Downarrow v \mid (p, q)$
- $(V', y \mapsto v') : (\Gamma', y : \tau)$
- $\Gamma', y : \tau \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid \ \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{c})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \sigma$

(2) $\Phi((V', y \mapsto v') : (\Gamma', y : \tau) \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b})) \geq p$

(3) $\Phi((V', y \mapsto v') : (\Gamma', y : \tau) \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b})) + q$
$\geq \Phi((V', y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, \mathtt{ret} : \sigma) \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{c})) + p$

The well-formedness judgment (1) $v : \sigma$ is what this case needs, so only this case's cost bounds remain to be proven.

Let $r = \Phi(V : \Gamma \mid \vec{a})$ and $s = \Phi((x \mapsto v') : (x : \tau) \mid \vec{a})$. Then the following inequalities confirm the peak cost bound:

$$\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \Upsilon^{x,\mathtt{arg}}_{x}(\vec{a}, \vec{b})) \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon^{x,\mathtt{arg}}_{x}(\vec{a}, \vec{b})) + \Phi(\mathtt{C}(V';\ g,\ y.\, e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \cdot) && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon^{x,\mathtt{arg}}_{x}(\vec{a}, \vec{b})) + \Phi(V' : \Gamma' \mid \vec{b'}) && \textit{def} \\
&= r + s + \Phi((\mathtt{arg} \mapsto v') : (\mathtt{arg} : \tau) \mid \vec{b}) + \Phi(V' : \Gamma' \mid \vec{b'}) && \textit{Lemma 3.4.1} \\
&= r + s + \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + \Phi(V' : \Gamma' \mid \vec{b'}) && \textit{relabelling} \\
&= r + s + \Phi((V', y \mapsto v') : (\Gamma', y : \tau) \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b})) && \textit{def}, \textit{Lemma 3.4.1} \\
&\geq \Phi((V', y \mapsto v') : (\Gamma', y : \tau) \mid \Upsilon^{z;\mathtt{c}}_{\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b})) && \textit{algebra} \\
&\geq p && (2)
\end{aligned}$$

And the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{b})) + q \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{b})) + \Phi(\mathtt{C}(V';\ g,\ y.\, e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \cdot) + q && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{b})) + \Phi(V' : \Gamma' \mid \vec{b'}) + q && \textit{def} \\
&= r + s + \Phi((\mathtt{arg} \mapsto v') : (\mathtt{arg} : \tau) \mid \vec{b}) + \Phi(V' : \Gamma' \mid \vec{b'}) + q && \textit{Lemma } 3.4.1 \\
&= r + s + \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + \Phi(V' : \Gamma' \mid \vec{b'}) + q && \textit{relabelling} \\
&= r + s + \Phi((V', y \mapsto v') : (\Gamma', y : \tau) \mid \Upsilon_{\mathtt{c}}^{z,\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{b})) + q && \textit{def}, \textit{Lemma } 3.4.1 \\
&\geq r + s + \Phi((V', y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, \mathtt{ret} : \sigma) \mid \Upsilon_{\mathtt{c}}^{z,\mathtt{c}}([z/\mathtt{c}]\vec{b'}, [y/\mathtt{arg}]\vec{c})) + p && (3) \\
&= r + s + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid [y/\mathtt{arg}]\vec{c}) + \Phi(V' : \Gamma' \mid \vec{b'}) + p && \textit{def}, \textit{Lemma } 3.4.1 \\
&= r + s + \Phi((\mathtt{arg} \mapsto v', \mathtt{ret} \mapsto v) : (\mathtt{arg} : \tau, \mathtt{ret} : \sigma) \mid \vec{c}) + \Phi(V' : \Gamma' \mid \vec{b'}) + p && \textit{relabelling} \\
&= r + \Phi((x \mapsto v', \mathtt{ret} \mapsto v) : (x : \tau, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})) + \Phi(V' : \Gamma' \mid \vec{b'}) + p && \textit{Lemma } 3.4.1 \\
&= r + \Phi((x \mapsto v', \mathtt{ret} \mapsto v) : (x : \tau, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})) + \Phi(\mathtt{C}(V';\ g,\ y.\, e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \cdot) + p && \textit{def} \\
&= \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(C';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a}, \vec{c})) + p && \textit{def}
\end{aligned}$$

5.7 Related Work

While remainder contexts are novel in their construction from I/O contexts and uncomputation, other work in AARA has made similar constructions. I take a moment here to compare and contrast this similar work. I follow this with a discussion of how program logics compare with remainder contexts. Otherwise, related work has already been discussed earlier in this chapter, particularly in Section 5.2.

Parallel Leftovers

Hoffmann and Shao develop a similar judgment for capturing leftovers in their work on extending AARA to parallel programs [79]. Just like remainder contexts, their judgment provides an additional annotation for the context which represent the leftover resources after evaluation. The purpose of their judgment is to help reconcile the costs of parallel evaluation against the forking context's single shared pool of potential energy. The authors also take advantage of their leftover judgment to type let expressions in the same way as *R-Let* (though they do not get rid of structural rules for weakening and contraction).

Nonetheless, remainder contexts differ significantly from Hoffmann and Shao's development. The main differences boil down to the fact that their system does not try to reuse resources at all—it is simply convenient to have a small amount of leftover reasoning for their parallel cost analysis. As a result, their work does not include features such as remainder contexts' new function types or uncomputation.

Give-Back Annotations

Campbell develops a *give-back* system for the purpose of recapturing leftover potential energy in AARA [25]. This purpose is almost the same as that of remainder contexts, except that the give-back system is more specialized to reclaiming memory in particular. As a result, these two systems do share many similarities, even though they operate by very different means.

While remainder contexts change the function types, the give-back system changes the non-function types. These give-back types come with two sets of annotations that are similar to—but distinct from— the two sets of annotations in the remainder-context typing rules. The key distinction is that give-back typing provides two sets of annotations for the expression being typed, while remainder contexts only provide one. The additional give-back annotation represents the amount of potential energy that might be restored to the expression's value after its later uses in computation. The conditions under which such potential energy is reclaimed roughly amount to the value no longer being used in the computation, but the precise conditions are different than those used by remainder contexts' uncomputation. For example, the give-back system reasoning extends through let expressions, which remainder contexts do not do.

Give-back reasoning also comes with more analysis overhead than remainder contexts. To properly formalize the reclaimation of potential energy, the give-back system makes use of multiple auxiliary analyses including heap separation and benign sharing. The give-back system type inference also still needs the variable analysis described in Section 3.7 to determine how to use its weakening and contraction rules—these rules are still needed because the give-back system uses different leftover reasoning than remainder contexts.

Program Logics

Remainder contexts do have some similarity to Hoare logic triples [74]. These triples take a form like $\{P\}e\{Q\}$ and mean that, whenever the precondition P holds, the postcondition Q will hold upon the termination of the expression e's execution. Initial and remainder contexts have a similar sort of interpration.

It is possible that remainder contexts could be recast as a program logic. Interestingly, multiple lines of work have already developed AARA-inspired program logics [10, 27]. However, these program logics have been targeted at imperative programs rather than functional progams. As a result, they focus on integers and pointers, and these do not benefit from uncomputation to the same extent that data structures do.

Chapter 6

Exponentials and Linear Recurrences

This chapter lays out a recipe for integrating exponential resource functions into AARA [95], forming this thesis's second contribution to the AARA type system. These resource functions are key to bolstering a blindspot in the polynomial AARA analysis: functions making multiple recursive calls. Such code patterns are natural in functional programming, especially for brute-force algorithms, but could not before be analyzed by AARA. Automatically providing exponential upper bounds on cost is the focus of this chapter's work, but these bounds may also be of interest for other reasons, such as analyzing *lower* cost bounds[112] to find complexity bugs—such other applications will not be explored in this chapter, however.

Exponential functions are represented using Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$ [132], which count the number of ways to divide n elements into k partitions. These combinatorial functions follow a linear recurrence similar to Pascal's identity. As a result, these new resource functions only require linear constraints, so AARA type inference can be efficiently automated via linear programming as before. It is also proven that Stirling numbers are the "best" way of representing exponential functions.

In fact, this chapter contributes more generally than just exponential resource functions. The main AARA system developed in this chapter is set up to be parameterized by a linear recurrence defining some basis of resource functions. Given the recurrence for Pascal's identity, it recovers the polynomial system, and, given the recurrence for Stirling numbers, it generates an exponential system. Thus, this setup uniformly represents both polynomials, exponentials, and more.

This chapter also includes some additional results concerning how to combine ("mix") different resource functions. In particular, *demotion* is provided as a special optimization for integrating exponential and polynomial resource functions.

Notably, the contributions of this chapter are orthogonal to the remainder contexts of the previous chapter, Chapter 5, and therefore could be implemented in AARA without remainder contexts. However, this chapter builds its contributions on top of remainder contexts to present a more coherent and complete AARA system.

```
fun d0 lst = tick{1}

fun d1 lst = case lst of
  | [] -> ()
  | _::t -> let _ = d0 t in d1 t

fun d2 lst = case lst of
  | [] -> ()
  | _::t -> let _ = d1 t in d2 t

fun d3 lst = case lst of
  | [] -> ()
  | _::t -> let _ = d2 t in d3 t
```

Figure 6.1: Degree-k cost functions up to degree 3

6.1 The Problem: Multiple Calls

It is very convenient that AARA is able to infer polynomial bounds, as polynomial costs describe some of the most common programs of interest. Indeed, typical code patterns of functional programs tend to encourage polynomial runtimes: functions that make one recursive call tend to be like those in Figure 6.1, where each function dk accrues cost in $\Theta(n^k)$ for inputs lists of length n. This tendency can actually be made more precise. For example, non-size-increasing types use functions making one recursive call to help characterize exactly the polynomial time computable functions [84].[1]

Of course, there are other common code patterns that do not yield polynomial time computations. Most notably, programs with *more than one* recursive call are completely natural to write in a functional language, and they tend to have exponential costs. Consider, for example, any existing solution to an NP-complete problem like subset sum (Figure 6.2[2]), which asks if there exists a subset of some multiset of numbers `nums` that sum to a given `target`. Such a solution takes $O(2^n)$ time for input lists of length n, and it naturally makes two recursive calls. More generically, consider Figure 6.3, where each program bk makes k recursive calls and accrues cost in $\Theta(k^n)$ for inputs lists of length n. This tendency can also be made more precise. For example, LFPL is a type system for functional languages making possibly-many recursive calls, and it precisely characterizes the exponential time computable functions [83].[3]

AARA with polynomial cost bounds simply cannot analyze such functions as in Figure 6.3 nor other exponential-cost functions, such as brute-force approaches to NP-complete problems like subset sum. This chapter focuses not only on expanding AARA to cover exponential costs, but also on making the approach general so that other kinds of cost bounds can be more easily

[1]Interestingly, this work uses a linear typing discipline and is related to the same line of research as AARA.

[2]For clarity, the code for subset sum given here makes use of the ”or” operator `||`, which can be implemented with the algebraic types provided in this thesis. For ease of providing examples, I continue to use similar shorthand for Boolean and numerical operators throughout this work.

[3]Interestingly, this work uses a linear typing discipline and is a *direct precursor* to AARA.

```
fun subSum (nums, target) = case nums of
  | [] -> target = 0
  | n::ns ->
    let with_n = subSum (ns, target - n) in
    let without_n = subSum (ns, target) in
    with_n || without_n
```

Figure 6.2: Code for subset sum

```
fun b0 lst = cast lst of
  | [] -> tick{1}
  | _::t -> tick{1}

fun b1 lst = case lst of
  | [] -> tick{1}
  | _::t -> let _ = tick{1} in
    b1 t

fun d2 lst = case lst of
  | [] -> tick{1}
  | _::t -> let _ = tick{1} in
    let _ = b2 t in
    b2 t

fun b3 lst = case lst of
  | [] -> tick{1}
  | _::t -> let _ = tick{1} in
  let _ = b3 t in
  let _ = b3 t in
  b3 t
```

Figure 6.3: Base-k cost functions up to base 3

incorporated into AARA as desired.

6.2 The Linear Ideas: Bases and Recurrences

To address the problem described in Section 6.1, I tease out and generalize the ideas that led to the success of AARA's polynomial resource functions[77]. These ideas boil down to using linear recurrences to represent new cost bounds of interest using linear combinations. To set up the background for working with with these objects, this section describes their key features in more detail.

6.2.1 Bases

The key way that AARA forms resource functions is through linear combinations of basis resource functions. I include this subsection only to ensure that certain terminology and concepts are clear concerning linear/conical bases and spans.

A *linear combination* of elements e_i is a sum of the form $\sum_{i=0}^{k} a_i \cdot e_i$ for some coefficients a_i. The *linear span* of some set e_i is the set of values expressible via the linear combinations of that set. An element e is *linearly independent* from a set of elements e_i if e is not in the linear span of e_i. A set of elements is called linearly independent if each of its elements is linearly independent from the rest of the set. A set of elements is a *linear basis* for a space if it is linearly independent and its linear span coincides with the space.

Each of the above linear concepts can be further refined into *conical* concepts. The only difference is that where linear combinations may be formed using any coefficients a_i, conical combinations require such coefficients to be nonnegative.

A *linear map* f is a function that distributes over sums and commutes with scalars. That is, $f(x + y) = f(x) + f(y)$ and $f(a \cdot x) = a \cdot f(x)$. Such a function behaves nicely with linear or conical combinations, which are made up of sums and scalar coefficients.

Linear maps and spans correspond nicely to matrices and vector spaces. Given a space that can be represented uniquely by the linear (or conical) combinations of some basis (as well as an ordering on the basis elements), elements of that space $\sum_{i=0}^{k} a_i \cdot e_i$ correspond exactly to vectors $\vec{b}$ where $\vec{b}_i = a_i$. Then any linear map on such a space can be represented as a matrix such that function application corresponds to matrix multiplication. Both this chapter and Chapter 8 make heavy use of this correspondence.

This vector construction always works for the spans of linear bases, but not conical ones. Whereas a collection of linearly independent elements spans a space uniquely, a collection of conically independent elements may not. For example, $\langle 1, 0\rangle$, $\langle -1, 0\rangle$, $\langle 0, 1\rangle$ and $\langle 0, -1\rangle$ are conically independent, but $\langle 0, 0\rangle$ can be obtained as either of conical combinations $\langle 1, 0\rangle + \langle -1, 0\rangle$ or $\langle 0, 1\rangle + \langle 0, -1\rangle$. Thus, when developing a conical space, this work tends to consider the conical span of a linear basis, so that the conical space is represented uniquely with conical combinations. Such uniqueness removes any need for conversion between equivalent representations.

Finally, I conclude this subsection with some additional properties concerning matrices with linearly independent rows: Such a matrix's *row echelon* form (or a prefix thereof) is an upper triangular matrix. The row echelon form of a matrix is obtained by adding linear combinations of matrix to rows to other matrix rows in order to cancel out leading nonzero entries. Formally, this process is known as *Gaussian elimination* and is a standard matrix manipulation from linear algebra. A row echelon form of a non-empty matrix with linearly dependent rows is never upper triangular. The row echelon form of linear bases plays a useful role in the construction of *maximally expressive* resource function bases in this chapter.

6.2.2 Recurrences

A linear recurrence is a numerical function defined recursively in terms of a linear combination of other functions calls. For example, the Fibonacci sequence follows a linear recurrence because the n^{th} Fibonacci number $F(n)$ is equal to $F(n-1)+F(n-2)$. This expression is a 2-element linear combination over the calls $F(n-1)$ and $F(n-2)$ where both coefficients are 1. More generally, a linear recurrence for a function f might look like $f(n) = \sum_{i=1}^{k} a_i \cdot f(n-i)$ for some set of coefficients a_i that do not depend on the function argument n.

For a linear recurrence to uniquely define a function, some initial values must be given. For the Fibonacci sequence, these values are $F(0) = F(1) = 1$. If the same recurrence is taken with the initial values $F(0) = 2$ and $F(1) = 1$, then the Lucas sequence would be defined instead.

One nice way to represent a linear recurrence is with a matrix equation. If $T(n)$ is defined by a linear recurrence over calls down to $T(n-j)$, let the j-element vector $\vec{b}$ be defined such that $\vec{b}_i = T(i)$ (0-indexed). One can then represent the recurrence $T(-)$ using an equation involving the following $j \times j$ matrix, where $\vec{0}$ is the column vector of zeros, I is the identity matrix, and $\vec{a}$ is the column vector of coefficients in the linear recurrence.

$$T(n) = \left(\left[\begin{array}{c|c} \vec{0} & I \\ \hline \multicolumn{2}{c}{\vec{a}^T} \end{array} \right]^n \vec{b} \right)_1$$

That is, the recurrent sequence $T(-)$ coincides with the first element of the vector given by the matrix multiplication. To see an example of this matrix equation, consider the Fibonacci recurrence, which assigns coefficients of 1 to its previous two values ($F(n) = 1 \cdot F(n-1) + 1 \cdot F(n-2)$) and also has initial values of 1. Then $\vec{a}$ and $\vec{b}$ are both vectors of ones, yielding the following identity:

$$F(n) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right)_1$$

These linear recurrences can also be given closed forms without matrices. While it is not so important to this work how such a closed form is derived, the form of such closed forms are useful to know. The general solution to a linear recurrence $T(n)$ takes the form of a linear combination of terms $n^{c_i} \cdot d_i^n$, for some set of constants c_i, d_i. Thus, these linear recurrences naturally generate polynomials (where $d_i = 1$), exponentials (where $c_i = 0$), and mixes between the two. This natural synergy between polynomials and exponentials is suggestive of the work in this chapter, which uses linear recurrences to create a hybrid polynomial and exponential cost bound system.

Despite having closed forms, the behaviour of functions following linear recurrences are not always easy to determine. It is not even known whether it is decidable if a sequence given by a linear recurrence ever contains a 0 (Skolem's problem) or a negative value (the positivity problem) [119]. The latter would pose a problem for AARA because energy must be nonnegative. The linear recurrences in this chapter avoid this problem by only considering cases where initial values and coefficients c_i are nonnegative, ensuring that the whole sequence is nonnegative.

So far, this section has only considered 1-dimensional linear recurrences. However, this thesis mostly works with multi-dimensional recurrences. The dimension of a recurrence refers to the nuber of arguments it takes. Pascal's identity $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$ is a 2-dimensional recurrence because the binomial coefficient function takes two arguments: n and k.

In this work, multidimensional recurrences take a form like $T(n,k) = \sum_{i=0}^{k} a_i \cdot T(n-1,i)$, where a_i does not depend on n. This independence of n lets this chapter's work use n to stand for the size of a data structure. Such sizes are not usually statically knowable, but the recurrence's independence of size allows it to be used regardless.

One important way to think of these multi-dimensional linear recurrences is as a collection of mutually defined 1-dimensional linear recurrences $T_k(n)$ indexed by k. This view allows one to represent a linear combination of such recurrences with a nice matrix equation, which will be used repeatedly throughout this chapter. Collect the values of $T_i(n)$ as a column vector $\vec{T}(n)$, and let column vector $\vec{b}_i$ give the coefficient of $T_i(n)$ in the linear combination. Then there exists a matrix A such that

$$\vec{b} \cdot \vec{T}(n) = (A\vec{b}) \cdot \vec{T}(n-1)$$

where the symbol "$\cdot$" is the dot product. In particular, column k (0-indexed) of the matrix A gives the coefficients a_i such that $T_k(n) = \sum_{i=0}^{k} a_i \cdot T_i(n-1)$. An example of such a matrix A for binomial coefficients is:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

I will make use of such matrix representations in both this chapter and Chapter 8 to parameterize shifting operations like $\lhd$ and $\blacktriangleleft$.

6.3 Setting Up

The goal of this chapter is to modularize the resource functions that AARA uses so that different classes of cost bounds can be derived without any fundamental changes to the type system. In particular, the only the change this chapter makes to AARA is to the shifting operators $\lhd$ and $\blacktriangleleft$.

To accomplish this goal, this section first identifies the key properties needed to represent polynomial cost bounds and explains their use. Then these properties are abstracted, and Stirling numbers of the second kind are introduced to form exponential resource functions. Section 6.4 then shows how to integrate such resource functions into AARA given the properties explained here. Later sections provide additional variations of this basic setup.

Polynomials

The key properties of binomial coefficients that AARA uses to represent polynomial resource functions are the following:

1. $\binom{n}{k} \geq 0$ for all $n, k \geq 0$
2. $\binom{n}{0} = 1$ for all $n \geq 0$
3. $\binom{0}{k} = 0$ for all $k \geq 1$
4. $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$
5. the set of $\binom{n}{k}$ form a linear basis for the space of polynomials over n

These properties come together in support of the following features:

- Because $\binom{n}{k}$ form a linear basis for the space of polynomials over n, AARA can represent polynomials using their linear combinations. Such linear combinations are represented in AARA via the annotations of indices of the form $\mathtt{d}_i$, $\mathtt{d}'_i$, or $\mathtt{c}$. These annotations represent coefficients in the linear combinations. These linear combinations are key to the remaining features.
- Because each $\binom{n}{k} \geq 0$ for all $n, k \geq 0$, one can ensure that the polynomial energy of a linear combination is nonnegative by constraining the coefficients to be nonnegative. Nonnegative energy is needed for reasoning about peak costs using the physicist's method.
- Because binomial coefficients follow a linear recurrence, their linear combinations can be converted between functions of n and $n-1$ while conserving energy. In particular, the fact that shifting conserves energy (Lemma 3.4.2) is a consequence of the equality $\vec{b} \cdot \vec{T}(n) = (A\vec{b}) \cdot \vec{T}(n-1)$ described in Section 6.2.2 where the shifting operator $\lhd$ takes the role of the linear map A and $\binom{n}{k}$ takes the role of $T_k(n)$. (This property is explained further later via Definition 6.3.1.)
- Because the coefficients in the linear recurrence of Pascal's identity are nonnegative, the matrix representing the shifting operation $\lhd$ is nonnegative. Thus, this matrix preserves conical combinations and can always be safely applied to obtain new annotations when destructing a data structure.
- Because $\binom{n}{0} = 1$ for all $n \geq 0$, the linear combinations naturally include constant or free energy tracked at index $\mathtt{c}$. This inclusion explains how constructing or destructing data structures interacts with the free energy of a context when applying the shift operator.
- Because $\binom{0}{k} = 0$ for all $k \geq 1$, empty data structures carry no energy. This property is important because it allows the empty data structures to have any annotation. If empty data structures had fixed annotations, so would every inductive data structure because every such data structure's annotations are determined from the annotations of the data structures it is built out of.
- All the matrix operations and constraints mentioned here are representable as linear constraints, and thus can be handled by linear programming.

In addition to the aforementioned five key properties of binomial coefficients, there are also two other properties of binomial coefficients worth mentioning. These additional properties allow AARA's system to not only work, but work well:

6. Not only do binomial coefficients form a linear basis for polynomials, but also their conical span forms a maximally large space of *nonnegative* polynomials. That is, no other linear basis for polynomials can express a strict superset of the nonnegative polynomials using only nonnegative linear combinations. The standard polynomial basis $\lambda n.n^k$ is not maximal in this way because it is strictly contained within the conical span of binomial coefficients. Thus, binomial coefficients are a maximally expressive basis for representing nonnegative polynomials.
7. Binomial coefficients can be given a natural combinatorial meaning with respect to lists. Specifically, $\binom{n}{k}$ counts the number of ordered k-tuples of list elements of a list of length n. Such combinatorial meaning can provide more natural approaches for reasoning about resource functions. For example, Grosen et al. have used the patterns of binomial coefficients to extend polynomial resource functions beyonds lists and trees to regular recursive types [65].

Before moving on, it is useful to make the connection between the shift operator $\lhd$ and linear maps more formal. To do this, Definition 6.3.1 is provided for a parameterized version of the shift operator $\overset{A}{\lhd}$. The parameterized shift only differs from the polynomial shift for indices like $\mathtt{d}_i$, $\mathtt{d}'_i$, or $\mathtt{c}$ (such indices are described in Section 3.2.2). On these indices, the parameterized shift applies a linear map A. If A is the matrix for Pascal's identity (like that given in Section 6.2.2), then in fact $\overset{A}{\lhd} = \lhd$ and Definition 3.2.2 is recovered. This redefinition exhibits how $\lhd$ behaves like a linear map over the annotations.

Likewise, a parameterized potential shifting operator $\overset{A}{\blacktriangleleft}$ can be defined as in Definition 6.3.2. Then similarly, if A is the matrix for Pascal's identity, $\overset{A}{\blacktriangleleft} = \blacktriangleleft$, recovering Definition 3.4.1. This potential shifting thus behaves like the linear map A for similar reasons as $\overset{A}{\lhd}$ does, so that over indices like $\mathtt{d}_i, \mathtt{d}'_i$, or $\mathtt{c}$, where $\mathtt{d}_i/\mathtt{d}'_i$ corresponds to i and $\mathtt{c}$ corresponds to 0 for indexing purposes, $\overset{A}{\blacktriangleleft}(\vec{a}) = A \cdot \vec{a}$.

Definition 6.3.1 (parameterized shifting)**.** *The parameterized shifting operator* $\overset{A}{\lhd}$ *transforms an annotation map for a list or tree according to a linear map* A*. Here, that means applying* A *to a vector of particular annotations* $\vec{b}$ *and copying the annotations for elements. The parameterized shifting operator is overloaded across both lists and trees, but to disambiguate whenever necessary, the following explicit definitions are provided alongside explicit selections of* $\vec{b}$*.*
$\overset{A}{\lhd}{}^{\ell}_{x,y}$ *acts on the annotation map* a *for a list* ℓ *where* $\ell = x :: y$*. Let* $\mathtt{c}$ *act as 0 and* $\ell.\mathtt{d}_i$ *act as* i *for vector indexing, and fix* $\vec{b}$ *such that* $\vec{b}_i = a(i)$*. Then, formally:*

$$\overset{A}{\lhd}{}^{\ell}_{x,y}(a) = \lambda i. \begin{cases} a(\ell.\mathtt{e}.j) & i = x.j \lor i = y.\mathtt{e}.j \\ (A \cdot \vec{b})_{\ell.\mathtt{d}_j} & i = y.\mathtt{d}_j \\ (A \cdot \vec{b})_{\mathtt{c}} & i = \mathtt{c} \\ a(i) & otherwise \end{cases}$$

$\overset{A}{\lhd}{}^{t}_{x,y,z}$ *acts on the annotation map* a *for a tree* t *where* $t = \mathtt{node}(x,\, y,\, z)$*. Let* $\mathtt{c}$ *act as 0 and* $t.\mathtt{d}'_i$ *act as* i *for vector indexing, and fix* $\vec{b}$ *such that* $\vec{b}_i = a(i)$*. Then, formally:*

$$\overset{A}{\lhd}{}^{t}_{x,y,z}(a) = \lambda i. \begin{cases} a(t.\mathtt{e}'.j) & i = x.\mathtt{e}'.j \lor i = y.j \lor i = z.\mathtt{e}'j \\ (A \cdot \vec{b})_{t.\mathtt{d}'_j} & i = x.\mathtt{d}'_j \lor i = z.\mathtt{d}'_j \\ (A \cdot \vec{b})_{\mathtt{c}} & i = \mathtt{c} \\ a(i) & otherwise \end{cases}$$

When unambiguous, one may simply write $\overset{A}{\lhd}$ *.*

Definition 6.3.2 (parameterized potential shifting)**.** *The parameterized potential shifting operator* $\overset{A}{\blacktriangleleft}$ *mirrors the critical action of the shifting operator* $\overset{A}{\lhd}$ *over indices like* $\mathtt{d}_i, \mathtt{d}'_i, \mathtt{c}, \mathtt{e}.i, \mathtt{e}'.i$*, but where no labels are present like* x *in* $x.\mathtt{d}_i$*.*
For lists annotated by a*, let* $b(\ell.i) = a(i)$*. Then formally:*

$$\overset{A}{\blacktriangleleft}(a) = \lambda i.\ \overset{A}{\lhd}{}^{\ell}_{x,y}(b)(y.i)$$

For trees annotated by a*, let* $b(t.i) = a(i)$*. Then formally:*

$$\overset{A}{\blacktriangleleft}(a) = \lambda i.\ \overset{A}{\lhd}{}^{t}_{x,y,z}(b)(x.i) = \lambda i.\ \overset{A}{\lhd}{}^{t}_{x,y,z}(b)(z.i)$$

Beyond Polynomials

To represent other kinds of cost bounds, I consider different families of basic resource functions that satisfy the same key properties as binomial coefficients. Let $R_k(n)$ be a family of resource functions indexed by k for data structures of size n. For AARA to best use the family $R_k(n)$ to represent resource functions, it would be ideal if the functions satisfied the all the following properties identified in the previous subsection:

1. $R_k(n) \geq 0$ for all $n, k \geq 0$
2. $R_0(n) = 1$ for all $n \geq 0$
3. $R_k(0) = 0$ for all $k > 0$
4. $R_{k+1}(n+1) = \sum_{i=0}^{k+1} a_i \cdot R_i(n)$ for constants $a_i \geq 0$
5. the set of $R_k(n)$ form a linear basis for the space of desired resource functions over n
6. optionally, the conical span of $R_k(n)$ forms a maximal conical space of nonnegative resource functions
7. optionally, $R_k(n)$ can be given a natural combinatorial meaning

Various functions satisfying some or all of these properties are given in Section 6.7.

The key family of basic resource functions that I consider in this chapter meets all of these properties and is given by $R_k(n) = \left\{ {n+1 \atop k+1} \right\}$, where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind [132]. The Stirling number $\left\{ {n \atop k} \right\}$ counts the number of ways to partition a set of n elements into k nonempty subsets, and the "offset" Stirling numbers of the second kind $\left\{ {n+1 \atop k+1} \right\}$ count the number of ways to pick k nonempty disjoint subsets out of n elements.

Conical combinations of offset Stirling numbers represent exponential resource functions of the form $\sum_{i=0}^{k} a_i \cdot (i+1)^n$. Such functions include $2^n, 3^n, 4^n$, etc. This work refers to linear combinations of these exponential functions as "exponentials".

Offset Stirling numbers have the closed form $\left\{ {n+1 \atop k+1} \right\} = \frac{1}{(k+1)!} \sum_{i=1}^{k+1} (-1)^{k+1-i} \cdot \binom{k+1}{i} \cdot i^{n+1}$ which shows $\left\{ {n+1 \atop k+1} \right\} \in O(k+1^n)$. For this reason, this work refers to the coefficient of $\left\{ {n+1 \atop k+1} \right\}$ as as the "base-(k+1)" annotation, similarly to how the annotation of $\binom{n}{k}$ is the degree-k annotation.

Stirling numbers of the second kind also satisfy a recurrence. This recurrence is given with initial values as follows:

$$\left\{ {n+1 \atop k+1} \right\} = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0, n = 0 \\ (k+1) \cdot \left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\} & \textit{otherwise} \end{cases}$$

This recurrence corresponds to the columns in a matrix like the following, which may be used to define a parameterized shift operator.

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

I establish the maximality of Stirling numbers in Section 6.5. That is, I establish that the conical combinations of Stirling numbers of the second kind form a maximally large space of nonnegative exponentials.

6.4 Parameterized System

Given the setup of Section 6.3, relatively little remains to adapt AARA to handle the modularization of resource functions via linear recurrences. Let such resource functions be given by the conical span of some basis $R_k(n)$ which satisfies all the non-optional properties described in Section 6.3. Let A be the pointwise nonnegative matrix corresponding to the recurrence for $R_k(n)$. That is, the matrix A satisfies

$$\vec{b} \cdot \vec{R}(n) = (A\vec{b}) \cdot \vec{R}(n-1)$$

where the functions R_k are collected as the vector $\vec{R}$, $\vec{b}$ is any vector of coefficients, and $n \geq 1$. Then all that remains is to replace the shifting operations $\lhd, \blacktriangleleft$ with parameterized versions $\overset{A}{\lhd}, \overset{A}{\blacktriangleleft}$. This section makes all those replacements explicit.

6.4.1 Typing Rules

To generalize the type system sufficiently to accept new resource functions given by $R_k(n)$, only one change needs to be made to the typing rules. That change is to replace all the polynomial shifts $\lhd$ in the typing rules with parameterized shifts $\overset{A}{\lhd}$. Such changes only apply to the rules *R-Cons, R-CaseL, R-Node* and *R-CaseT*. The replacement rules are provided in Figure 6.4. Otherwise, every typing rule remains as given in Figures 5.6 and 5.7.

Example 6.4.1. To see how these typing rules allow the use of, e.g., expontential resource functions, recall the code for subset sum in Figure 6.2. Suppose one is interested in bounding the number of arithmetic and Boolean operations performed when calling this function. It can be shown by induction that the number of such operations is $3 \cdot 2^n - 2 = 3 \cdot \left\{ {n+1 \atop 2} \right\} + 1$.

To derive a bound on the number of operations using AARA, the code is first put into let-normal form and ticks are added before each operation. By then using the recurrence for Stirling numbers of the second kind with this chapter's typing rules, the function `subSum` can be given the type $\langle L^3(\mathbb{Z}) \otimes \mathbb{Z};\, 1\rangle \to \langle \mathbb{B};\, 0\rangle \sim L^0(\mathbb{Z}) \otimes \mathbb{Z}$. This type corresponds exactly to the desired bound.

The typing derivation of `subSum` is witnessed by the energy comments given in Figure 6.5. As there is no remainder energy, I elide any comments for the remainder annotations. For notational brevity, I write "$a : x, y$" in a given line to indicate that there are y units of free energy available and list a has x units of base-2 energy at that line.

The key line to look at in Figure 6.5 is line 3 where the argument `nums` is pattern matched. In this line, energy is distributed over `ns`, the tail of `nums`, by releasing three units of energy and doubling the base-2 energy. This corresponds exactly to the recurrence $\left\{ {n+1 \atop 2} \right\} = 2 \cdot \left\{ {n \atop 2} \right\} + 1$ (scaled up by 3).

P-Cons

$$\frac{}{\Gamma, x : \tau, y : L(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\overset{A}{\lhd}{}^{\texttt{ret}}_{x',y'}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

P-CaseL

$$\frac{\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \qquad \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{c}, \vec{d})}{\Gamma, x : L(\sigma) \mid \vec{a}, \curlyvee_x^{x,x'}(\vec{b}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}, \curlyvee_x^{x,x'}(\vec{d})}$$

P-Node

$$\frac{}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}{}^{\texttt{ret}}_{x',y',z'}(\vec{a})))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

P-CaseT

$$\frac{\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'} \qquad \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{c}, \vec{d})}{\Gamma, t : T(\sigma) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}) \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d})}$$

Figure 6.4: New typing rules using parameterized shift

```
fun subSum (nums, target) = case nums of        (* nums: 3,1 *)
  | [] -> let _ = tick{1} in target = 0         (* []:   0,0 *)
  | n::ns ->                                    (* ns:   6,4 *)
    let _ = tick{1} in let tmp = target - n in  (* ns:   6,3 *)
    let with_n = subSum (ns, tmp) in            (* ns:   3,2 *)
    let without_n = subSum (ns, target) in      (* ns:   0,1 *)
    let _ = tick{1} in with_n || without_n      (* ns:   0,0 *)
```

Figure 6.5: Code for subset sum with energy comments

$$\Phi(v_1 :: v_2 : L(\tau) \mid \vec{a}) = \delta(A, \vec{a}) + \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}.i}) + \Phi(v_2 : L(\tau) \mid \overset{A}{\blacktriangleleft} (\vec{a}))$$

$$\Phi(\mathtt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \vec{a}) = \delta(A, \vec{a}) + \Phi(v_1 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\vec{a})) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}'.i}) + \Phi(v_3 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\vec{a}))$$

Figure 6.6: New potential energy definitions using parameterized shift

6.4.2 Potential Energy

Similarly to the typing rules, the definition of the potential function also only requires swapping polynomial shifts $\blacktriangleleft$ with parameterized ones $\overset{A}{\blacktriangleleft}$. The new definitions of potential energy for nonempty lists and trees are given in Figure 6.6. Otherwise, the definitions are unchanged from Figure 3.9. These definitions also make use of the "constant difference" operator δ defined in Definition 6.4.1 to pick out a particular value that arises during shifting.

Definition 6.4.1 (constant difference operator)**.** *The constant difference operator δ picks out the change in free energy when shifting $\vec{a}$ with the linear map A.*

$$\delta(A, \vec{a}) = \overset{A}{\blacktriangleleft} (\vec{a})_{\mathtt{c}} - \vec{a}_{\mathtt{c}}$$

By convention, if $\mathtt{c}$ *does not index $\vec{a}$, then it is treated as if $\vec{a}_{\mathtt{c}} = 0$.*

By construction, this definition of potential energy results in Lemmas 6.4.1 and 6.4.2. These lemmas are generalizations of Lemma 3.4.2 and Lemma 3.4.3, respectively.

Lemma 6.4.1 (parameterized shifting conserves energy)**.**

$$\Phi((x \mapsto v_1 :: v_2) : (x : L(\tau)) \mid \vec{a}) = \Phi((y \mapsto v_1, z \mapsto v_2) : (y : \tau, z : L(\tau)) \mid \overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))$$

$$\Phi((t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3)) : (t : T(\tau)) \mid \vec{a})$$
$$= \Phi((x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (x : T(\tau), y : \tau, z : T(\tau)) \mid \overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))$$

Proof. First, in case $\mathtt{c}$ does not already index $\vec{a}$, temporarily let $\vec{a}_{\mathtt{c}} = c$ for some c. The result will be independent of this mapping.

Both the case for lists and trees can be shown to hold directly just by following definitions.

lists For the list case, the following equations hold:

$$\begin{aligned}
&\Phi((y \mapsto v_1, z \mapsto v_2) : (y : \tau, z : L(\tau)) \mid \overset{A}{\lhd}{}^{x}_{y,z}(\vec{a})) \\
&= \overset{A}{\lhd}{}^{x}_{y,z}(\vec{a})_{\mathsf{c}} + \Phi(v_1 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{y.i}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \overset{A}{\lhd}{}^{x}_{y,z}(\vec{a})_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \Phi(v_1 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{y.i}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && \textit{algebra} \\
&= \vec{a}_{\mathsf{c}} + \overset{A}{\blacktriangleleft} (\lambda i.\, \vec{a}_{x.i})_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \Phi(v_1 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{y.i}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{x.i}) + \Phi(v_1 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{y.i}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{x.i}) + \Phi(v_1 : \tau \mid \lambda i.\, (\lambda(y.j).\, \vec{a}_{x.\mathsf{e}.j})_{y.i}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{x.i}) + \Phi(v_1 : \tau \mid \lambda j.\, (\lambda i.\, \vec{a}_{x.i})_{\mathsf{e}.j}) + \Phi(v_2 : L(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{x}_{y,z}(\vec{a}))_{z.i}) && =_{\beta} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{x.i}) + \Phi(v_1 : \tau \mid \lambda j.\, (\lambda i.\, \vec{a}_{x.i})_{\mathsf{e}.j}) + \Phi(v_2 : L(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda i.\, \vec{a}_{x.i})) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \Phi(v_1 :: v_2 : L(\tau) \mid \lambda i.\, \vec{a}_{x.i}) && \textit{def} \\
&= \Phi((x \mapsto v_1 :: v_2) : (x : L(\tau)) \mid \vec{a}) && \textit{def}
\end{aligned}$$

trees For the tree case, the following equations hold:

$$\begin{aligned}
&\Phi((x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (x : T(\tau), y : \tau, z : T(\tau)) \mid \overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a})) \\
&= \overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a})_{\mathsf{c}} + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{y.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a})_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{y.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && \textit{algebra} \\
&= \vec{a}_{\mathsf{c}} + \overset{A}{\blacktriangleleft} (\lambda i.\, \vec{a}_{t.i})_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{y.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{t.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{y.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{t.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda i.\, (\lambda(y.j).\, \vec{a}_{t.\mathsf{e}'.j})_{y.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{t.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{x.i}) + \Phi(v_2 : \tau \mid \lambda j.\, (\lambda i.\, \vec{a}_{t.i})_{\mathsf{e}'.j}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, (\overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a}))_{z.i}) && =_{\beta} \\
&= \vec{a}_{\mathsf{c}} + \delta(A, \lambda i.\, \vec{a}_{t.i}) + \Phi(v_1 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda i.\, \vec{a}_{t.i})) + \Phi(v_2 : \tau \mid \lambda j.\, (\lambda i.\, \vec{a}_{t.i})_{\mathsf{e}'.j}) + \Phi(v_3 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda i.\, \vec{a}_{t.i})) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \Phi(\mathtt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \lambda i.\, \vec{a}_{t.i}) && \textit{def} \\
&= \Phi((t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3)) : (x : T(\tau)) \mid \vec{a}) && \textit{def}
\end{aligned}$$

□

Lemma 6.4.2 (list potential energy using parameterized shift)**.** *The potential energy of a list v with annotation $\vec{a}$ is a function of v's length n plus the potential energy of v's elements.*

$$\Phi(v : L(\tau) \mid \vec{a}) = \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i})$$

Proof. The proof of this statement proceeds by induction over n, the length of v.

n=0 In this case, the list v is the empty list [ ] and the following equalities hold:

$$\begin{aligned}
\Phi([\,] : L(\tau) \mid \vec{a}) &= 0 && \textit{def} \\
&= \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(0) && R_i(0) = 0 \\
&= \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(0) + \sum_{v' \in v} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && v \ \textit{empty}
\end{aligned}$$

n+1 In this case, the list v takes the form $v_1 :: v_2$ and the following equalities hold. For these equalities, if c does not already index $\vec{a}$, temporarily let $\vec{a}_{\mathsf{c}} = c$ for some c. The result will be independent of this mapping.

$$\begin{aligned}
&\Phi(v_1 :: v_2 : L(\tau) \mid \vec{a}) \\
&= \delta(A, \vec{a}) + \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{i.\mathsf{e}}) + \Phi(v_2 : L(\tau) \mid \overset{A}{\blacktriangleleft} (\vec{a})) && \textit{def} \\
&= \delta(A, \vec{a}) + \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{i.\mathsf{e}}) + \sum_{i=1}^{D_{max}} \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v_2} \Phi(v' : \tau \mid \lambda i.\ \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{e}.i}) && \textit{IH} \\
&= \delta(A, \vec{a}) + \Phi(v_1 : \tau \mid \lambda i.\, \vec{a}_{i.\mathsf{e}}) + \sum_{i=1}^{D_{max}} \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{def} \\
&= \delta(A, \vec{a}) + \sum_{i=1}^{D_{max}} \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{algebra} \\
&= \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \sum_{i=1}^{D_{max}} \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{def} \\
&= \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{c}} \cdot R_0(n) - \vec{a}_{\mathsf{c}} + \sum_{i=1}^{D_{max}} \overset{A}{\blacktriangleleft} (\vec{a})_{\mathsf{d}_i} \cdot R_i(n) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && R_0(n) = 1 \\
&= -\vec{a}_{\mathsf{c}} + (A \cdot \vec{b}) \cdot \vec{R}(n) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{def} \\
&= -\vec{a}_{\mathsf{c}} + \vec{b} \cdot \vec{R}(n+1) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} \cdot R_0(n+1) - \vec{a}_{\mathsf{c}} + \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(n+1) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} - \vec{a}_{\mathsf{c}} + \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(n+1) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && R_0(n+1) = 1 \\
&= \sum_{i=1}^{D_{max}} \vec{a}_{\mathsf{d}_i} \cdot R_i(n+1) + \sum_{v' \in v_1 :: v_2} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathsf{e}.i}) && \textit{algebra}
\end{aligned}$$

□

Lemma 6.4.2 explains how the parameterization on A allows the AARA type system to represent resource functions that are linear combinations of $R_k(n)$. In particular, when letting A be the matrix for the Stirling numbers of the second kind, exponentials are representable.

While I do not include any explicit closed form for the potential energy of trees analagous to Lemma 3.4.4,[4] I do note that a tree's energy is related to a list's energy like before. In particular, they coincide when every node of the tree includes a leaf.

Finally, by design, the helpful principles given by Lemmas 3.4.5 and 3.4.6 still hold in this new setting with $R_k(n)$ resource functions. Lemma 3.4.6, the linearity of energy with respect to annotations, continues to hold because the potential function continues to be defined via linear combination. Lemma 3.4.5, the monotonicity of energy with respect to annotations, continues to hold because $R_k(n) \geq 0$. Further, the annotation of all zeros continues to assign zero energy because $R_k(0) = 0$; thus, like before, a direct consequence of Lemma 3.4.5 is that nonnegative annotations always assign nonnegative potential energy.

6.4.3 Soundness

The soundness of AARA parameterized on matrices like A is given as Theorem 6.4.3. This theorem is fundamentally no different than in previous chapters, and in fact is the same statement as Theorem 5.4.1. The only difference is that the soundness proof here must take into account the new presence of paramaterized shifts in rules related to the construction or destruction of lists or trees. Otherwise, it is still the case that the initial potential energy of the context bounds the peak cost of evalution, and the difference between initial and final energies bounds the net cost.

Theorem 6.4.3 (parameterized soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p$ *(diff. bounds net)*

Proof. The soundness proof proceeds by lexicographic induction over the derivation of the evaluation judgment followed by the typing judgment.

In all but four cases, the proof is identical to that for Theorem 5.4.1. Those four exceptions follow similarly to Theorem 5.4.1, but now are parameterized on a matrix A. By design, this parameterization can be covered entirely by using Lemma 6.4.1 instead of Lemma 3.4.2 in the proof cases. The four exception cases are given as follows:

[4]Even if such a closed form exists, I do not believe it would be particularly illuminating.

E-Cons Suppose the last rule applied for the evaluation judgment is *E-Cons*.

$$\textsc{E-Cons}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{P-Cons}\quad \frac{}{\Gamma, x : \tau, y : L(\tau) \mid \Upsilon_{x}^{x,x'}(\Upsilon_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

Because $v_1 :: v_2 : L(\tau)$ follows from *V-Cons* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 6.4.1), the net cost bound is also satisfied with the following equality:

$$\begin{aligned}&\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid \Upsilon_{x}^{x,x'}(\Upsilon_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(\vec{a}))))\\ =\ &\Phi((V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto v_1 :: v_2) : (\Gamma, x : \tau, y : L(\tau), \mathtt{ret} : L(\tau)) \mid \vec{a})\end{aligned}$$

E-CaseL-Cons Suppose the last rule applied for the evaluation judgment is *E-CaseL-Cons*.

$$\textsc{E-CaseL-Cons}\quad \frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p,q)}{V, x \mapsto v_1 :: v_2 \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{P-CaseL}\quad \frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid \vec{c}, \vec{d'}\\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{c}, \vec{d})\end{array}}{\Gamma, x : L(\sigma) \mid \vec{a}, \Upsilon_{x}^{x,x'}(\vec{b}) \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \vec{c}, \Upsilon_{x}^{x,x'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 : L(\sigma)$. Then further, the rule *V-Cons* can be inverted to learn both $v_1 : \sigma$ and $v_2 : L(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$$

Each of the following judgments has now been found:

- $V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p,q)$

- $(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$
- $\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{c}, \vec{d})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b})) \geq p$

(3) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b})) + q$
$\geq \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{c}, \vec{d})) + p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 6.4.1), both these cost bounds follow from (2) and (3) using the following equalities:

$$\begin{aligned}&\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{a}, \vec{b}))\\ &\quad= \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid \vec{a}, \curlyvee^{x,x'}_{x}(\vec{b}))\end{aligned}$$

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(\vec{c}, \vec{d}))$$

$$= \Phi((V, x \mapsto v_1 :: v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee^{x,x'}_{x}(\vec{d}))$$

E-Node Suppose the last rule applied for the evaluation judgment is *E-Node*.

E-Node
$$\frac{}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\ y,\ z) \Downarrow \texttt{node}(v_1,\ v_2,\ v_3) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

P-Node
$$\frac{}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee^{x,x'}_{x}(\curlyvee^{y,y'}_{y}(\curlyvee^{z,z'}_{z}(\overset{A}{\lhd}{}^{\texttt{ret}}_{x',y',z'}(\vec{a})))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

Because $\texttt{node}(v_1,\ v_2,\ v_3) : T(\tau)$ follows from *V-Node* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because sharing conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 6.4.1), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \curlyvee^{x,x'}_{x}(\curlyvee^{y,y'}_{y}(\curlyvee^{z,z'}_{z}(\overset{A}{\lhd}{}^{\texttt{ret}}_{x',y',z'}(\vec{a})))))$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau), \texttt{ret} : T(\tau)) \mid \vec{a})$$

E-CastT-Node Suppose the last rule applied for the evaluation judgment is *E-CaseT-Node*.

E-CaseT-Node
$$\frac{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

P-CaseT
$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid \vec{a}, \vec{b'} \vdash e_1 : \tau \mid\ \vec{c}, \vec{d'} \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{c}, \vec{d})\end{array}}{\Gamma, t : T(\sigma) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ \vec{c}, \curlyvee_t^{t,t'}(\vec{d})}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 v_3 : T(\sigma)$. Then further, the rule *V-Node* can be inverted to learn all of $v_1 : T(\sigma)$, $v_2 : \sigma$, and $v_3 : T(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$$

Each of the following judgments has now been found:

- $V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$
- $\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b}) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{c}, \vec{d})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b}))$
$\geq p$

(3) $\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b})) + q$
$\geq \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{c}, \vec{d}))$
$+ p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 6.4.1), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{a}, \vec{b}))$$

$$= \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma)) \mid \vec{a}, \curlyvee_t^{t,t'}(\vec{b}))$$

$$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(\vec{c}, \vec{d}))$$

$$= \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{c}, \curlyvee_t^{t,t'}(\vec{d}))$$

□

6.4.4 Automation

By design, the resource function abstraction described in this chapter does not significantly impact how AARA is automated. Thus, type inference still follows the following two steps:

1. basic type inference
2. collect and solve linear contraints

The collected constraints are still largely the same as from previous chapters. These constraints are just those dictated by the typing rules, with the addition that all annotations are nonnegative. By design, ensuring all annotations are nonnegative is sufficient to ensure that potential energy is nonnegative. This nonnegativity is ensured because the values scaled by the annotations, specifically the resource functions $R_k(n)$, are each nonnegative.

The only difference between the automation of this chapter and previous chapters is that some of the linear constraints associated with shifting are changed. Rather than shift according to the matrix for Pascal's identity, this chapter's version of AARA shifts according to a provided matrix parameter. The number of collected constraints is only dependent upon the parameter D_{max} giving the dimension of the matrix, not on the matrix contents.

In total then, type inference in this system is efficient. After basic type inference, the rest of type inference can be reduced to collecting and solving linear constraints, which only takes polynomial time in the size of the source code.

6.5 Maximality

There are many families of functions $R_k(n)$ that satisfy the non-optional properties of Section 6.3, and many such families even form bases for the same space of resource functions. For example, Both $\lambda n.\, \binom{n}{k}$ and $\lambda n.n^k$ are linear bases for polynomials, and both $\lambda n.\, \left\{ {n+1 \atop k+1} \right\}$ and $\lambda n.\, (k+1)^n$ are linear bases for exponentials. However, there are good reasons to favor using the bases $\lambda n.\, \binom{n}{k}$ and $\lambda n.\, \left\{ {n+1 \atop k+1} \right\}$ over other options. One reason is non-technical: both have nice combinatorial interpretations. But the main technical reason to favor them is their *maximal expressivity*. In this section, I provide both sufficient conditions for maximal expressivity and a simple algorithm that can transform linear bases into maximally expressive ones.

This chapter uses a given collection of functions $R_k(n)$ to form a *linear* basis for a space of resource functions, but AARA only expresses these resource functions using *conical* combinations of $R_k(n)$. This mismatch between linearity and conicity is important—a linear basis gives a unique representation, and a conical combination ensures nonnegativity. However, this mismatch also fundamentally means that the resource functions expressed via conical combinations only cover a portion of the whole linear space of resource functions. For a family of resource functions $R_k(n)$ to be *maximally expressive*, this portion should be maximally large so that the most resource functions possible can be expressed with conical combinations of $R_k(n)$. In other words, no alternative linear basis for the resource function space should have a strictly larger conical span. Both binomial coefficients and offset Stirling numbers of the second kind are maximally expressive in this sense.

It might not be easy to determine if a given nonnegative basis $R_k(n)$ is maximally expressive. To aid in this determination, I provide a simple characterization of $R_k(n)$ which ensures maximal expressivity. This characterization applies to both binomial coefficients and offset Stirling numbers of the second kind, explaining generally why they are good choices of basis. Intuitively, the characterization is that each function R_i has a distinct asymptotic growth[5] and is 0 on its first i inputs. I formally state this characterization as Theorem 6.5.1.

Theorem 6.5.1 (maximally expressive characterization)**.** *Let the finite function families R_k and S_k both be nonnegative linear bases for the same space of functions $\mathcal{F}$ over $\mathbb{N}$. Let their conical spans be $\mathcal{C}$ and $\mathcal{D}$, respectively.*
If $R_k \in o(R_{k+1})$ and $R_k(n) = 0$ until $n \geq k$, then it cannot be that $\mathcal{C} \subsetneq \mathcal{D}$.

Proof. Let the highest index of the family R_k be p, and suppose for the sake of contradiction that $\mathcal{C} \subsetneq \mathcal{D}$.

To begin, observe the following 1:1 correspondence between R_k and S_k: For each i, there is a unique j such that $S_j \in O(R_i) - O(R_{i-1})$ if $i > 0$, or simply $S_j \in O(R_i)$ if $i = 0$. This correspondence can be shown to hold by comparing the growth rate of each R_i to the conical combination of S_k assumed to exist that equals R_i. Such a conical combination cannot be made with any functions outside of $O(R_i)$, else the conical combination would be outside of $O(R_i)$, but $R_i \in O(R_i)$. Further, for $i > 0$, such a conical combination cannot be made solely out of functions in $O(R_{i-1})$, else the conical combination would be in $O(R_{i-1})$, but R_i is in the disjoint set $\omega(R_{i-1})$. Thus there must be at least one such function S_j where $S_j \in O(R_i) - O(R_{i-1})$ if $i > 0$, or $S_j \in O(R_i)$ if $i = 0$. Finally, because both S_k and R_k are linear bases for $\mathcal{F}$, they must have the same finite cardinality $p + 1$, rendering the correspondence 1:1.

For simplicity, index S_k from now on to match R_k's indexing, so that $S_i \in O(R_i) - O(R_{i-1})$ for $i > 0$. This growth rate correspondence ensures that any representation of R_i as a conical combination of S_k only requires indices of S_k up to i.

Now recall the supposition that $\mathcal{C} \subsetneq \mathcal{D}$. In other words, there is some function $f \in \mathcal{D}$ such that $f \notin \mathcal{C}$. Without loss of generality, this f can be chosen as one of the basis functions S_k; pick such a basis function S_ℓ that mimimizes ℓ. The following implications then hold for some coefficient vector $\vec{a}$ and index i:

$$\begin{aligned} \mathcal{C} \subsetneq \mathcal{D} &\implies S_\ell \in \mathcal{D} \wedge S_\ell \notin \mathcal{C} && \textit{def} \\ &\implies S_\ell \in \mathcal{F} \wedge S_\ell \notin \mathcal{C} && \mathcal{D} \subseteq \mathcal{F} \\ &\implies S_\ell = \vec{a} \cdot \vec{R} \wedge \vec{a}_i < 0 && R_k \textit{ linear basis of } \mathcal{F},\ \textit{conical basis of } \mathcal{C} \end{aligned}$$

Let i be chosen as the lowest such index where $\vec{a}_i < 0$.

It must be that $\ell \geq i$. If instead $\ell < i$, the nonnegative weight $\vec{a}_i$ of R_i would cause S_ℓ to be in $\Omega(R_i)$, while also $S_\ell \in O(R_\ell) \subseteq o(R_i)$ from known growth rate correspondences and orderings. Because $\Omega(R_i)$ and $o(R_i)$ are disjoint, it must therefore be that $\ell \geq i$.

[5]Likely this part of the condition could be replaced with some other way of forcing alternative bases to correspond nicely with R_i. However, having distinct asymptotic growths is a sufficient condition for polynomials and exponentials.

Now consider evaluating S_ℓ on i. This reveals some more information about representing S_ℓ as a linear combination of the functions R_k.

$$\begin{aligned}
& S_\ell(i) \geq 0 && S_k \geq 0 \\
\implies & \sum_{j=0}^{p} \vec{a}_j \cdot R_j(i) \geq 0 && S_\ell = \vec{a} \cdot \vec{R} \\
\implies & \sum_{j=0}^{i} \vec{a}_j \cdot R_j(i) \geq 0 && j > i \implies R_j(i) = 0 \\
\implies & \vec{a}_i \cdot R_i(i) + \sum_{j=0}^{i-1} \vec{a}_j \cdot R_j(i) \geq 0 && \mathit{algebra} \\
\implies & \sum_{j=0}^{i-1} \vec{a}_j \cdot R_j(i) > 0 && \vec{a}_i < 0 \wedge R_i(i) > 0 \\
\implies & \exists j < i.\, \vec{a}_j > 0 && R_k \geq 0
\end{aligned}$$

Let m be the minimal witnessing index j of the last implication. Because m is the index of the first positive entry and i is the index of the first negative entry, where $m < i$, the vector of coefficients $\vec{a}$ is known to have the following form: only 0 as entries until index m, where $\vec{a}_m > 0$, then only nonnegative entries up through index i, where $\vec{a}_i < 0$.

Next consider representing R_ℓ as a linear combination of S_k, which must be possible because $R_\ell \in \mathcal{F}$ and S_k is a linear basis for $\mathcal{F}$. Then the following inequalities hold for some vectors $\vec{b}, \vec{c}$.

$$\begin{aligned}
R_\ell &= \sum_{j=0}^{p} \vec{b}_j \cdot S_j \quad (\vec{b} \geq 0) && R_\ell \in \mathcal{C} \subsetneq \mathcal{D} \\
&= \sum_{j=0}^{\ell} \vec{b}_j \cdot S_j \quad (\vec{b}_\ell > 0) && \mathit{growth\ rate} \\
&= \vec{b}_\ell \cdot S_\ell + \sum_{j=0}^{\ell-1} \vec{b}_j \cdot S_j && \mathit{algebra} \\
&= \vec{b}_\ell \cdot (\sum_{j=0}^{p} \vec{a}_j \cdot R_j) + \sum_{j=0}^{\ell-1} \vec{b}_j \cdot S_j && S_\ell = \vec{a} \cdot \vec{R} \\
&= \vec{b}_\ell \cdot (\sum_{j=m}^{p} \vec{a}_j \cdot R_j) + \sum_{j=0}^{\ell-1} \vec{b}_j \cdot S_j && j < m \implies \vec{a}_j = 0 \\
&= \vec{b}_\ell \cdot \vec{a}_m \cdot R_m + \vec{b}_\ell \cdot (\sum_{j=m+1}^{p} \vec{a}_j \cdot R_j) + \sum_{j=0}^{\ell-1} \vec{b}_j \cdot S_j && \mathit{algebra} \\
&= \vec{b}_\ell \cdot \vec{a}_m \cdot R_m + \vec{b}_\ell \cdot (\sum_{j=m+1}^{n} \vec{a}_j \cdot R_j) + \sum_{j=0}^{\ell-1} \vec{c}_j \cdot R_j \quad (\vec{c} \geq 0) && j < \ell \implies S_j \in \mathcal{C}
\end{aligned}$$

Finally, examine the last of these equalities. The righthand side expression is a linear combination of R_k where the coefficient of R_m, $\vec{b}_\ell \cdot \vec{a}_m + \vec{c}_m$, is strictly positive. However, the lefthand side expression R_ℓ is also a linear combination of R_k (a trivial singleton combination) where the coefficient of R_m is 0, as $m < i \leq \ell$. Because the functions R_k are linearly independent, such an equality is impossible. Thus the original assumption is false and it is not the case that $\mathcal{C} \subsetneq \mathcal{D}$. □

The characterization of Theorem 6.5.1 is useful not only because it applies to binomial coefficients and Stirling numbers of the second kind, but also because it is relatively straightforward to transform a linear basis of functions to have the second property. Such a transformation yielding $R_k(n) = 0$ until $n \geq k$ is given by the well-studied algorithm of Gaussian elimination, putting the functions into a row echelon form.

To see an example of this transformation, consider the standard basis for polynomials, the functions $\lambda n.\, n^k$. These can be put into a matrix as follows, where index i, j contains the value j^i (letting $0^0 = 1$).

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots \\ 0 & 1 & 2 & 3 & \dots \\ 0 & 1 & 4 & 9 & \dots \\ 0 & 1 & 8 & 27 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Putting this matrix into row echelon form yields:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots \\ 0 & 1 & 2 & 3 & \dots \\ 0 & 0 & 2 & 6 & \dots \\ 0 & 0 & 0 & 3 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

This new matrix's rows corresponds to the functions $\lambda n.\, 1$, $\lambda n.\, n$, $\lambda n.\, n^2 - n$, $\lambda n.\, n^3 - 3n^2 + 2n$, etc. Notably, all these functions are constant scalars of binomial coefficients. Removing these scalars to get the binomial coefficients exactly corresponds to normalizing the leading coefficients of each row to 1. This natural matrix transformation emphasizes just how natural binomial coefficients are for representing polynomial resource functions.

If one similarly puts functions of the form $\lambda n.\, (k+1)^n$ into row echelon form with leading coefficients of 1, one performs the following matrix transformation:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots \\ 1 & 2 & 4 & 8 & \dots \\ 1 & 3 & 9 & 27 & \dots \\ 1 & 4 & 16 & 64 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 1 & 1 & 1 & \dots \\ 0 & 1 & 3 & 7 & \dots \\ 0 & 0 & 1 & 6 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

At index i, j, the latter matrix contains the value $\left\{ {j+1 \atop i+1} \right\}$, the offset Stirling numbers of the second kind. Thus, these offset Stirling numbers are the natural choice for representing exponential resource functions.

6.6 Mixing

Now that this chapter has established the basics of modularizing resource functions using linear recurrences, an additional recipe can be laid out to combine such resource functions. In particular, if $R_k(n)$ and $S_\ell(n)$ are two families of resource functions, this recipe allows multiplicatively mixing the two families to obtain a resource basis of the form $R_k(n)S_\ell(n)$. This section includes both this mixing recipe as well as a deeper dive on the specific mix between polynomial and exponential resource functions.

6.6.1 General Mixing

The key to generally mixing resource functions given by linear recurrences is to simply apply each recurrence separately.[6] To exemplify this process, consider the following identity for the mix between polynomials and exponentials:

$$\begin{aligned}\binom{n+1}{k+1}\left\{{n+2 \atop j+2}\right\} &= (\binom{n}{k+1} + \binom{n}{k})((j+2)\left\{{n+1 \atop j+2}\right\} + \left\{{n+1 \atop j+1}\right\}) \\ &= (j+2)\binom{n}{k+1}\left\{{n+1 \atop j+2}\right\} + \binom{n}{k+1}\left\{{n+1 \atop j+1}\right\} + (j+2)\binom{n}{k}\left\{{n+1 \atop j+2}\right\} + \binom{n}{k}\left\{{n+1 \atop j+1}\right\}\end{aligned}$$

This identity simply applies the recurrence for binomials and exponentials separately, then distributes. The resulting expression is just a linear combination of mixed polynomial and exponential terms with no dependence on the size n. As a result, the identity expresses another linear recurrence, and can in principle be expressed with a matrix equation just as described in Section 6.7.

More generally, the product between any two valid resource function bases meeting the properties of Section 6.3 yields another valid resource function basis meeting the properties of Section 6.3. This property can be formally stated as Lemma 6.6.1.

Lemma 6.6.1 (mixed resource functions)**.** *If*

1. $R_k(n) \geq 0$ *and* $S_\ell(n) \geq 0$ *for all* $n, k, \ell \geq 0$
2. $R_0(n) = S_0(n) = 1$ *for all* $n \geq 0$
3. $R_k(0) = S_\ell(0) = 0$ *for all* $k, \ell > 0$
4. $R_{k+1}(n+1) = \sum_{i=0}^{k+1} a_i \cdot R_i(n)$ *and* $S_{\ell+1}(n+1) = \sum_{i=0}^{\ell+1} b_i \cdot S_i(n)$ *for constants* $a_i \geq 0$ *and* $b_i \geq 0$

then

5. $R_k(n)S_\ell(n) \geq 0$ *for all* $n \geq 0$ *and* $\langle k, \ell\rangle \geq \langle 0, 0\rangle$ *(here* $\geq$ *is pointwise)*
6. $R_0(n)S_0(n) = 1$ *for all* $n \geq 0$
7. $R_k(0)S_\ell(0) = 0$ *for all* $\langle k, \ell\rangle > \langle 0, 0\rangle$ *(here* $>$ *means* $\geq$ *and* $\neq$*)*
8. $R_{k+1}(n+1)S_{\ell+1}(n+1) = \sum_{\langle i,j\rangle=\langle 0,0\rangle}^{\langle k+1,\ell+1\rangle} c_{i,j} \cdot R_i(n)S_j(n)$ *for constants* $c_{i,j} \geq 0$

Proof. This lemma is proved by cases.

[6] To the more linear-algebra-minded reader, this process is a sort of tensor construction.

Property 5 Property 5 follows from property 1 because the product of nonnegative numbers is nonnegative.

Property 6 Property 6 follows from property 2 because $1 \cdot 1 = 1$.

Property 7 Property 7 follows from property 3 because $0 \cdot 0 = 0$.

Property 8 Property 8 is the only interesting case. It follows from the following equations:

$$\begin{aligned} R_{k+1}(n+1)S_{\ell+1}(n+1) &= (\sum_{i=0}^{k+1} a_i \cdot R_i(n))(\sum_{j=0}^{\ell+1} b_i \cdot S_i(n)) && \textit{property } 4 \\ &= \sum_{\langle i,j\rangle=\langle 0,0\rangle}^{\langle k+1,\ell+1\rangle} a_i \cdot b_j \cdot R_i(n)S_j(n) && \textit{algebra} \end{aligned}$$

Thus $c_{i,j} = a_i \cdot b_j$ is sufficient, where $c_{i,j} \geq 0$ because each of a_i and b_j are nonnegative.

□

To successfully use resource functions derived from Lemma 6.6.1 in the parameterized system of Section 6.4 and Section 6.5, there is only one wrinkle: Whereas the original functions were indexed in a total order, the mixed functions are only indexed by a lattice. However, this wrinkle turns out to not be an issue because no property developed in this chapter really depends on total ordering over lattice ordering. Simply identify 0 with the bottom element $\langle 0, 0\rangle$, D_{max} with the top element, interpret inequalities between lattice elements as described in Lemma 6.6.1, etc., and then everything developed in this chapter applies just as well to these new mixed resource functions. Or one can extend every lattice into a total order via topological sorting—this treatment also allows 1-dimensional indices like $\mathtt{d}_i$ to enumerate multidimensional lattice indices. Section 6.6.2 takes this latter approach, allowing annotation indices to appear as pairs of numbers for notational convenience.

This mixing recipe can also be repeated to obtain mixes between more than two resource functions. While the appropriate matrix representing such recurrences might grow complicated, they can always be recovered from the coefficients $c_{i,j}$ constructed in Lemma 6.6.1.

6.6.2 Mixing Polynomials and Exponentials

Now I turn the focus to mixes between polynomial and exponential functions specifically. A basis for such functions takes the form of $\lambda n. \binom{n}{k}\left\{ {n+1 \atop j+1} \right\}$. These functions can represent linear combinations of functions of the form $\lambda n.\, n^k(j+1)^n$, which include polynomials when $j = 0$ and exponentials when $k = 0$.

When mixing polynomial and exponential resource functions, there is one particular optimization that can be introduced. This optimization makes use of the following identity:

$$\left\{ {n+1 \atop 2} \right\} = 2^n - 1 = \sum_{k=1}^{\infty} \binom{n}{k}$$

$$Ind(L(\tau)) = \{\mathtt{d}_{m,n} \mid 0 \leq m \leq D_{max} \wedge 1 \leq n \leq B_{max} \wedge m = 0 \rightarrow n \neq 1\} \cup \mathtt{e}.Ind(\tau)$$

$$Ind(T(\tau)) = \{\mathtt{d}'_{m,n} \mid 0 \leq m \leq D_{max} \wedge 1 \leq n \leq B_{max} \wedge m = 0 \rightarrow n \neq 1\} \cup \mathtt{e}'.Ind(\tau)$$

Figure 6.7: Mixed annotation indices

As a result, one unit of base 2 potential energy can be converted to one unit of *every* kind of base-k potential energy. In the rest of this subsection, I make this notion formal through a special structural rule for "demotion", alongside additional index support in the type system.

Annotation Indices

To better organize the naturally 2-dimensional space of mixed polynomial/exponential resource functions, I refine the annotation index system here to explicitly give two subscripts to indices like $\mathtt{d}_{\_}$. These subscripts are explicitly exhibited in Figure 6.7, where only the new indices for lists and trees are included; all other indices remain the same. Note that these "new" indices are just a notational convenience; the system presented so far in this chapter already can accept such 2-dimensional lattice annotation indices.[7]

With these subscripts, $\mathtt{d}_{i,j}$ can be used to represent the basis resource function $\lambda n. \binom{n}{i} \left\{ {n+1 \atop j} \right\}$. Note that j here is not incremented; I include this minor reindexing for notational ease so that, e.g., $\mathtt{d}_{0,3}$ represents base-3 potential energy without off-by-one confusion. This figure also uses a new designated maximum B_{max} for the largest base of exponential potential energy considered, in addition to D_{max} giving the largest degree of polynomial potential considered. As a formal lattice, the maximum element is therefore the pair $\langle D_{max}, B_{max} \rangle$.

Finally, there is one new condition included in the indices: $m = 0 \rightarrow n \neq 1$. This condition just ensures that not both m and n are their minimal values, as such an index $\mathtt{d}_{0,1}$ already exists as the free energy index $\mathtt{c}$. This avoidance of duplication is not really new, but previously indices like $\mathtt{d}_0$ were avoided simply by restricting the 1-dimensional range.

Typing Rules

The system of mixed polynomials and exponentials uses all the typing rules of Section 6.4.1 with parameterized shifts instanced at the appropriate matrix for the mixed polynomial/exponential recurrence. In addition, this system introduces one additional structural rule given in Figure 6.8. This new rule is for "demoting" p units of base-2 exponential potential energy into p units of *each* degree-k potential energy, and takes the form of a subtyping rule with a more complicated relation than pointwise comparison. This subtyping is given by the relation $<:$, which is also

[7]More explicitly, these pairs of indices can be given the lexicographic linear ordering of the size of the second element followed by the size of the first. An enumeration according to this order not only can be fit into a 1-dimensional index system, but also respects asymptotic growth ordering for the applicability of Theorem 6.5.1. As Section 6.6.1 has shown, this setup still follows a linear recurrence, and so shifting, potential energy, etc. are all well-defined as given earlier in this chapter.

$$\textsc{D-Demote}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a'} <: \vec{a} \qquad \vec{b} <: \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

$$\textsc{D-List}\quad \frac{\begin{array}{c}\mathtt{dom}(\vec{a}) = \mathtt{dom}(\vec{b}) = \{i.\mathtt{d}_{\mathit{0,2}}\} \cup \{i.\mathtt{d}_{k,\mathit{1}} \mid 1 \leq k \leq D_{max}\} \\ \vec{a}_{i.\mathtt{d}_{\mathit{0,2}}} - \vec{b}_{i.\mathtt{d}_{\mathit{0,2}}} = p \qquad \forall k.\, \vec{b}_{i.\mathtt{d}_{k,\mathit{1}}} - \vec{a}_{i.\mathtt{d}_{k,\mathit{1}}} = p \qquad p \geq 0\end{array}}{\vec{a}, \vec{c} <: \vec{b}, \vec{c}}$$

$$\textsc{D-Tree}\quad \frac{\begin{array}{c}\mathtt{dom}(\vec{a}) = \mathtt{dom}(\vec{b}) = \{i.\mathtt{d}'_{\mathit{0,2}}\} \cup \{i.\mathtt{d}'_{k,\mathit{1}} \mid 1 \leq k \leq D_{max}\} \\ \vec{a}_{i.\mathtt{d}'_{\mathit{0,2}}} - \vec{b}_{i.\mathtt{d}'_{\mathit{0,2}}} = p \qquad \forall k.\, \vec{b}_{i.\mathtt{d}'_{k,\mathit{1}}} - \vec{a}_{i.\mathtt{d}'_{k,\mathit{1}}} = p \qquad p \geq 0\end{array}}{\vec{a}, \vec{c} <: \vec{b}, \vec{c}}$$

Figure 6.8: Demotion rules

formalized in Figure 6.8. (While I do not formulate demotion to be idempotent here to keep the rules simple, it is easy to create an idempotent version for actual implementations.)

Intuitively, due to the aforementioned relation between $\left\{ {n+1 \atop 2} \right\}$ and binomial coefficients, demotion can lose energy, but never can gain energy; it is therefore a safe manipulation of annotations. This safety will be made formal in Lemma 6.6.2.

This demotion rule also allows the system to support *negative* annotations for the first time. Previously the typing judgment $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ was defined so that $\vec{a} \geq 0$ and $\vec{b} \geq 0$, but this condition can now be relaxed to allow more flexible AARA typing. Because demotion can only lose energy, if all annotations are nonnegative after demotion, the annotations before demotion must have expressed nonnegative total energy. Thus, rather than require that each annotation like $\mathtt{d}_{k,\mathit{1}}$ is individually nonnegative in $\vec{a}$, it instead suffices to require that $\vec{a}_{i.\mathtt{d}_{\mathit{0,2}}} + \vec{a}_{i.\mathtt{d}_{k,\mathit{1}}} \geq 0$ for each k. The index $\vec{a}_{i.\mathtt{d}_{\mathit{0,2}}}$ must still be nonnegative on its own, but this relaxed condition now allows indices like $\vec{a}_{i.\mathtt{d}_{k,\mathit{1}}}$ to be negative. Such annotations could demote into nonnegative annotations, and therefore express a nonnegative total amount of energy, as is necessary for physicist's method peak-cost reasoning. (This same relaxation applies to $\vec{b}$ and $\mathtt{d}'\_$ in place of $\mathtt{d}\_$ as well.) The benefits of the additional flexibility afforded by demotion are shown later in Example 6.6.1.

Potential Energy

The potential function is not changed, but rather specialized to the mix of polynomials and exponentials. As a result, the same properties hold as before. For example, the potential of a list can be given by the following due to Lemma 6.4.2:

$$\Phi(v : L(\tau) \mid \vec{a}) = \sum_{i=1}^{D_{max}} \sum_{j=2}^{B_{max}} \vec{a}_{\mathtt{d}_{i,j}} \cdot \binom{|v|}{i} \left\{ {|v|+1 \atop j} \right\} + \sum_{v' \in v} \Phi(v' : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}.i})$$

However, there is one new useful comment to make about potential energy: Demotion does not gain energy. This property is formalized as Lemma 6.6.2.

Lemma 6.6.2 (demotion energy)**.** *Demotion does not gain energy.*

$$\vec{a} <: \vec{b} \implies \Phi(V : \Gamma \mid \vec{a}) \geq \Phi(V : \Gamma \mid \vec{b})$$

Proof. This statement is proven by cases over which rule is used to obtain the relation $<:$ between $\vec{a}$ and $\vec{b}$. Becauses lists have a nice closed form for potential energy given by Lemma 6.4.2, the list case is straightforward. The tree case, however, is more involved.

D-List Suppose this rule was used to infer the subtyping relation:

$$\textsc{D-List}\quad \frac{\begin{array}{c}\texttt{dom}(\vec{a}) = \texttt{dom}(\vec{b}) = \{i.\texttt{d}_{\mathit{0},\mathit{2}}\} \cup \{i.\texttt{d}_{k,\mathit{1}} \mid 1 \leq k \leq D_{max}\} \\ \vec{a}_{i.\texttt{d}_{\mathit{0},\mathit{2}}} - \vec{b}_{i.\texttt{d}_{\mathit{0},\mathit{2}}} = p \qquad \forall k.\, \vec{b}_{i.\texttt{d}_{k,\mathit{1}}} - \vec{a}_{i.\texttt{d}_{k,\mathit{1}}} = p \qquad p \geq 0\end{array}}{\vec{a}, \vec{c} <: \vec{b}, \vec{c}}$$

Then its premisses hold by inversion. Further there is some list in V, typed by Γ, at index i, and annotated by $\vec{a}$ and $\vec{b}$. Let that list be given by $v : L(\tau)$. Because the potential energy is linear with respect to the annotation (Lemma 3.4.6), it suffices to consider only $\vec{a}$ and $\vec{b}$, as $\vec{c}$ remains constant. That is, it suffices to show the following, where $\vec{a'}, \vec{b'}$ extend $\vec{a}, \vec{b}$ respectively with zeros for indices in the domain of $\vec{c}$.

$$\Phi(v : L(\tau) \mid \lambda j.\, \vec{a'}_{i.j}) \geq \Phi(v : L(\tau) \mid \lambda j.\, \vec{b'}_{i.j})$$

This inequality can be shown as follows:

$$\begin{aligned}
&\Phi(v : L(\tau) \mid \lambda j.\, \vec{a'}_{i.j}) \\
&= \vec{a}_{i.\mathtt{d}_{0,2}} \cdot \left\{ {|v| \atop 2} \right\} + \sum_{k=1}^{D_{max}} \vec{a}_{i.\mathtt{d}_{k,1}} \cdot \binom{|v|}{k} && \textit{Lemma 6.4.2, zeroing} \\
&= (\vec{b}_{i.\mathtt{d}_{0,2}} + p) \cdot \left\{ {|v| \atop 2} \right\} + \sum_{k=1}^{D_{max}} (\vec{b}_{i.\mathtt{d}_{k,1}} - p) \cdot \binom{|v|}{k} && \textit{D-List premisses} \\
&= \vec{b}_{i.\mathtt{d}_{0,2}} \cdot \left\{ {|v| \atop 2} \right\} + p \cdot \left\{ {|v| \atop 2} \right\} + \sum_{k=1}^{D_{max}} (\vec{b}_{i.\mathtt{d}_{k,1}} - p) \cdot \binom{|v|}{k} && \textit{algebra} \\
&= \vec{b}_{i.\mathtt{d}_{0,2}} \cdot \left\{ {|v| \atop 2} \right\} + p \cdot \sum_{k=1}^{\infty} \binom{|v|}{k} + \sum_{k=1}^{D_{max}} (\vec{b}_{i.\mathtt{d}_{k,1}} - p) \cdot \binom{|v|}{k} && \textit{identity} \\
&\geq \vec{b}_{i.\mathtt{d}_{0,2}} \cdot \left\{ {|v| \atop 2} \right\} + p \cdot \sum_{k=1}^{D_{max}} \binom{|v|}{k} + \sum_{k=1}^{D_{max}} (\vec{b}_{i.\mathtt{d}_{k,1}} - p) \cdot \binom{|v|}{k} && \binom{|v|}{k} \geq 0 \\
&= \vec{b}_{i.\mathtt{d}_{0,2}} \cdot \left\{ {|v| \atop 2} \right\} + \sum_{k=1}^{D_{max}} \vec{b}_{i.\mathtt{d}_{k,1}} \cdot \binom{|v|}{k} && \textit{algebra} \\
&= \Phi(v : L(\tau) \mid \lambda j.\, \vec{b'}_{i.j}) && \textit{Lemma 6.4.2, zeroing}
\end{aligned}$$

D-Tree Suppose this rule was used to infer the subtyping relation:

$$\frac{\begin{array}{c}\textsc{D-Tree} \\ \mathtt{dom}(\vec{a}) = \mathtt{dom}(\vec{b}) = \{i.\mathtt{d}'_{0,2}\} \cup \{i.\mathtt{d}'_{k,1} \mid 1 \leq k \leq D_{max}\} \\ \vec{a}_{i.\mathtt{d}'_{0,2}} - \vec{b}_{i.\mathtt{d}'_{0,2}} = p \qquad \forall k.\, \vec{b}_{i.\mathtt{d}'_{k,1}} - \vec{a}_{i.\mathtt{d}'_{k,1}} = p \qquad p \geq 0\end{array}}{\vec{a}, \vec{c} <: \vec{b}, \vec{c}}$$

Then its premisses hold by inversion. Further there is some tree in V, typed by Γ, at index i, and annotated by $\vec{a}$ and $\vec{b}$. Let that tree be given by $v : T(\tau)$. Because the potential energy is linear with respect to the annotation (Lemma 3.4.6), it suffices to consider only $\vec{a}$ and $\vec{b}$, as $\vec{c}$ remains constant. That is, it suffices to show the following, where $\vec{a'}, \vec{b'}$ extend $\vec{a}, \vec{b}$ respectively with zeros for indices in the domain of $\vec{c}$.

$$\Phi(v : T(\tau) \mid \lambda j.\, \vec{a'}_{i.j}) \geq \Phi(v : T(\tau) \mid \lambda j.\, \vec{b'}_{i.j})$$

To show this inequality,first note the following identity:

$$\begin{aligned}
\delta(A, \lambda j.\, \vec{a'}_{i.j}) &= \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})_{\mathtt{c}} - \lambda j.\, \vec{a'}_{i.j}(\mathtt{c}) && \textit{def} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})_{\mathtt{c}} && \textit{zeroing} \\
&= \vec{a}_{i.\mathtt{d}_{0,2}} + \vec{a}_{i.\mathtt{d}_{1,1}} && \textit{def} \\
&= (\vec{a}_{i.\mathtt{d}_{0,2}} - p) + (\vec{a}_{i.\mathtt{d}_{1,1}} + p) && \textit{algebra} \\
&= \vec{b}_{i.\mathtt{d}_{0,2}} + \vec{b}_{i.\mathtt{d}_{1,1}} && D - \textit{Tree premisses} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{b'}_{i.j})_{\mathtt{c}} && \textit{def} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{b'}_{i.j})_{\mathtt{c}} + \lambda j.\, \vec{b'}_{i.j}(\mathtt{c}) && \textit{zeroing} \\
&= \delta(A, \lambda j.\, \vec{b'}_{i.j}) && \textit{def}
\end{aligned}$$

Then note the following relation between $\overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})$ and $\overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})$. Specifically, let $\vec{d}$ match $\overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})$ except at index $\mathtt{d}_{D_{max},1}$, where $\vec{d}_{\mathtt{d}_{D_{max},1}} = \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{d}_{D_{max},1}} + p$. Then $\overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j}) \leq \vec{d} <: \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})$. The demotion part of this relation can be checked by considering the values of these annotation maps at each kind of index where $2 \cdot p$ units of base-2 energy get demoted.

At index $\mathtt{d}_{\mathtt{c}}$:

$$\begin{aligned}
\vec{d}_{\mathtt{c}} &= \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{c}} && \textit{def} \\
&= \vec{b}_{i.\mathtt{d}_{0,2}} + \vec{b}_{i.\mathtt{d}_{1,1}} && \textit{def} \\
&= (\vec{a}_{i.\mathtt{d}_{0,2}} - p) + (\vec{a}_{i.\mathtt{d}_{1,1}} + p) && D - \textit{Tree premisses} \\
&= \vec{a}_{i.\mathtt{d}_{0,2}} + \vec{a}_{i.\mathtt{d}_{1,1}} && \textit{algebra} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})_{\mathtt{c}} && \textit{def}
\end{aligned}$$

At index $\mathtt{d}_{0,2}$:

$$\begin{aligned}
\vec{d}_{\mathtt{d}_{0,2}} &= \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{d}_{0,2}} && \textit{def} \\
&= 2 \cdot \vec{b}_{i.\mathtt{d}_{0,2}} && \textit{def} \\
&= 2 \cdot (\vec{a}_{i.\mathtt{d}_{0,2}} - p) && D - \textit{Tree premiss} \\
&= 2 \cdot \vec{a}_{i.\mathtt{d}_{0,2}} - 2 \cdot p && \textit{algebra} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})_{\mathtt{d}_{0,2}} - 2 \cdot p && \textit{def}
\end{aligned}$$

At index $\mathtt{d}_{D_{max},1}$:

$$\begin{aligned}
\vec{d}_{\mathtt{d}_{D_{max},1}} &= \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{d}_{D_{max},1}} + p && \textit{def} \\
&= \vec{b}_{\mathtt{d}_{D_{max},1}} + p && \textit{def} \\
&= (\vec{a}_{\mathtt{d}_{D_{max},1}} + p) + p && D - \textit{Tree premiss} \\
&= \vec{a}_{\mathtt{d}_{D_{max},1}} + 2 \cdot p && \textit{algebra} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})_{\mathtt{d}_{D_{max},1}} && \textit{def}
\end{aligned}$$

At index $\mathtt{d}_{k,1}$ for $k < D_{max}$:

$$\begin{aligned}
\vec{d}_{\mathtt{d}_{k,1}} &= \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{d}_{k,1}} + \overset{A}{\blacktriangleleft} (\lambda j.\vec{b'}_{i.j})_{\mathtt{d}_{k+1,1}} && \textit{def} \\
&= \vec{b}_{i.\mathtt{d}_{k,1}} + \vec{b}_{i\mathtt{d}_{k+1,1}} && \textit{def} \\
&= (\vec{a}_{i.\mathtt{d}_{k,1}} + p) + (\vec{a}_{i\mathtt{d}_{k+1,1}} + p) && D - \textit{Tree premiss} \\
&= \vec{a}_{i.\mathtt{d}_{k,1}} + \vec{a}_{i\mathtt{d}_{k+1,1}} + 2 \cdot p && \textit{algebra} \\
&= \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})_{\mathtt{d}_{D_{max},1}} + 2 \cdot p && \textit{def}
\end{aligned}$$

And at other indices, each map is just 0 by definition. Thus, *D-Tree* applies, deriving that $\vec{d} <: \overset{A}{\blacktriangleleft} (\lambda j.\vec{a'}_{i.j})$.

With these relations in mind, this case can finally be finished as follows:

$$\begin{aligned}
&\Phi(\mathtt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \lambda j.\, \vec{a'}_{i.j}) \\
&= \delta(A, \lambda j.\, \vec{a'}_{i.j}) + \Phi(v_1 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})) + \Phi(v_2 : \tau \mid \lambda i.\, 0) + \Phi(v_3 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})) && \textit{def} \\
&= \delta(A, \lambda j.\, \vec{b'}_{i.j}) + \Phi(v_1 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})) + \Phi(v_2 : \tau \mid \lambda i.\, 0) + \Phi(v_3 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{a'}_{i.j})) && \textit{identity} \\
&\geq \delta(A, \lambda j.\, \vec{b'}_{i.j}) + \Phi(v_1 : T(\tau) \mid \vec{d}) + \Phi(v_2 : \tau \mid \lambda i.\, 0) + \Phi(v_3 : T(\tau) \mid \vec{d}) && \textit{IH} \\
&\geq \delta(A, \lambda j.\, \vec{b'}_{i.j}) + \Phi(v_1 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{b'}_{i.j})) + \Phi(v_2 : \tau \mid \lambda i.\, 0) + \Phi(v_3 : T(\tau) \mid \overset{A}{\blacktriangleleft} (\lambda j.\, \vec{b'}_{i.j})) && \textit{Lemma } 3.4.5 \\
&= \Phi(\mathtt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \lambda j.\, \vec{b'}_{i.j}) && \textit{def}
\end{aligned}$$

□

Soundness

The soundness of mixed polynomial/exponential AARA is given as Theorem 6.6.3. This theorem only needs to take into account the new demotion rule, and otherwise proceeds exactly as in Theorem 6.4.3, just specialized to the particular recurrence for mixed polynomials and exponentials. Otherwise, it continues to be the case that the initial potential energy of the context bounds the peak cost of evalution, and the difference between initial and final energies bounds the net cost.

Theorem 6.6.3 (demotion soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p$ *(diff. bounds net)*

Proof. This proof proceeds by induction exactly as *Theorem* 6.4.3, but with one new structural typing rule to consider for demotion. I only show that new rule here.

Demotion Suppose the last rule applied for the typing judgment is *D-Demote*.

$$\frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a'} <: \vec{a} \qquad \vec{b} <: \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}\ \textsc{D-Demote}$$

Then the premiss of this rule holds by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'}$ to learn:

(1) $v : \tau$
(2) $\Phi(V : \Gamma \mid \vec{a'}) \geq p$
(3) $\Phi(V : \Gamma \mid \vec{a'}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from from inequalities (2) and (3) by applying the demotion energy lemma Lemma 6.6.2 alongside the annotation subtyping relations $\vec{a'} <: \vec{a}$ and $\vec{b} <: \vec{b'}$.

□

Example 6.6.1. To demonstrate the utility of mixed polynomial exponential resource functions, consider the function `subSum'` in Figure 6.9. This function solves the NP-complete problem subset sum by brute force for sets represented by lists with repetition. Subset sum asks if a subset of a given finite set[8] of numbers sums to some target. The code for `subSum'` assumes this set is represented by a list with possible repetitions, so to consider each number in a set only once, it filters these repititions out dynamically. This filtering occurs by iterating over the list using the function `remove`.

Now consider the cost of running `subSum'` on a list of length n. If each Boolean or arithmetic operation accrues one unit of cost, then its brute force approach has at least exponential cost. However, the use of the function `remove` makes the cost more complicated. Letting

[8]Often subset sum is formulated using a multiset instead of a set. However, both definitions are NP-complete.

```
  fun subSum' (nums, target) = case nums of
    | [] -> target = 0
    | n::ns ->
      let ns' = remove n ns in
      let with_n = subSum' (ns', target - n) in
      let without_n = subSum' (ns', target) in
      with_n || without_n
```

Figure 6.9: Code for subset sum not counting repetitions

```
fun subSum' (nums, target) =                     (* nums: 1,2,0,1 *)
  | [] ->                                        (* []:   0,0,0,1 *)
    let _ = tick{1} in target = 0                (* []:   0,0,0,0 *)
  | n::ns ->                                     (* ns:   2,6,1,4 *)
    let ns' = remove n ns                        (* ns':  2,6,0,4 *)
    let _ = tick{1} in let t = target - n        (* ns':  2,6,0,3 *)
    let with_n = subSum' (ns', t) in             (* ns':  1,4,0,2 *)
    let without_n = subSum' (ns', target) in     (* ns':  0,2,0,1 *)
    let _ = tick{1} in with_n || without_n       (* ns':  0,2,0,0 *)
```

Figure 6.10: Let-normal code for `subSum'` with energy comments and no demotion

`remove` consume cost equal to the length of its input list, one can prove by induction that running the total cost of running `subSum'` is at worst $4 \cdot 2^n - n - 3$.

Because the exact cost bound involves both exponential and polynomial terms, itcannot be tightly bounded by an AARA system without exponentials and polynomials. This subsection now proceeds to type this function using mixed exponential and polynomial AARA both with and without demotion, demonstrating how demotion is necessary for achieving a tight cost analysis.

First consider typing `subSum'` *without* demotion. This AARA system finds a net cost bound of $n \cdot 2^n + 2 \cdot 2^n - n - 1 = \binom{n}{1}\left\{{n+1 \atop 2}\right\} + 2 \cdot \left\{{n+1 \atop 2}\right\} + 1$. Such a cost corresponds to `subSum'` having an AARA type like $\langle L^{1,2,0}(\mathbb{Z}) \otimes \mathbb{Z};\, 1\rangle \to \langle \mathbb{B};\, 0\rangle \sim L^{0,0,0}(\mathbb{Z}) \otimes \mathbb{Z}$, where the $1, 2, 0$ refers to the units of linear-base-2 energy, base-2 energy, and linear energy, respectively. The annotation bookkeeping in the comments of of Figure 6.10 shows that (the let-normal version of) `subSum'` can indeed be assigned this type, where the the comment $x : a, b, c, d$ represents the context at that line of the program having d units of free energy and a list x with a units of linear-base-2 energy, b units of base-2 energy, and c units of linear energy. The bound found here is not tight, which can be seen because the last line leaves two units of base-2 energy which must be weakened away and wasted. (This energy cannot be recovered because unshifting the annotation map for `ns'` would result in negative entries.)

Now consider typing `subSum'` *with* demotion. This AARA system finds a net cost bound of $4 \cdot \left\{{n+1 \atop 2}\right\} - \binom{n}{1} + 1 = 4 \cdot 2^n - n - 3$, which is the exact bound of interest. To do so, this system makes use of the fact that it can use negative annotations for polynomial resource functions so long as they do not exceed the base-2 annotation in magnitude. Indeed, $4 \geq |-1|$, so this system

```
fun subSum (nums, target) =                          (* nums: 4,-1,1 *)
  | [] ->                                            (* []:   0,0,1 *)
    let _ = tick{1} in target = 0                    (* []:   0,0,0 *)
  | n::ns ->                                         (* ns:   8,-1,4 *)
    let ns' = remove n ns                            (* ns':  8,-2,4 *)
    let _ = tick{1} in let t = target - n            (* ns':  8,-2,3 *)
    let with_n = subSum (ns', t) in                  (* ns':  4,-1,2 *)
    let without_n = subSum (ns', target) in          (* ns':  0,0,1 *)
    let _ = tick{1} in with_n || without_n           (* ns':  0,0,0 *)
```

Figure 6.11: Let-normal code for `subSum'` with energy comments and demotion

may consider a type of $\langle L^{4,-1}(\mathbb{Z}) \otimes \mathbb{Z}; 1\rangle \to \langle \mathbb{B}; 0\rangle \sim L^{0,0}(\mathbb{Z}) \otimes \mathbb{Z}$, where the $4, -1$ refers to having four units of base-2 energy and -1 units of linear energy. This type can be justified by the energy accounting of Figure 6.11, which uses a similar comment notation to the previous example. This bound can also be seen to be tight because the energy accounting wastes no energy through weakening, unlike the previous example. Thus, demotion is a useful optimization to the mixed polynomial/exponential system of AARA.

6.7 Additional Recurrences

I now show three additional examples of potentially useful linear recurrences beyond those for binomial coefficients and Stirling numbers of second kind. Some of these recurrences are not directly integrable into AARA without additional work but are still provided to show that this chapter's techniques show promise for further development of AARA's resource functions.

Harmonic Numbers

It is possible to get a linear recurrence for harmonic numbers H_n by mixing them with polynomials, letting $R_k(n) = \binom{n}{k} \cdot H_n$. The recurrence for such resource functions is given as follows:

$$R_k(n) = \begin{cases} H_n & k = 0 \\ 0 & n = 0 \\ \frac{1}{k}\binom{n-1}{k-1} + R_k(n-1) + R_{k-1}(n-1) & \textit{otherwise} \end{cases}$$

However, note this recurrence has a dependence on both the binomial coefficient $\binom{n-1}{k-1}$ and the harmonic number H_n, which does not fit the recurrence pattern laid out in this chapter. While binomial coefficients can be handled using the recurrence of Pascal's identity, Harmonic numbers are harder to pin down. Nonetheless, $H_n \in \Theta(log(n))$ and there has been work on logarithmic resource functions [89]. This recurrence therefore shows a plausible route to representing quasipolynomial cost bounds like $n \cdot log(n)$, which currently do not exist in the AARA literature.

Factorials

It is possible to get a linear recurrence for factorials $n!$ by mixing them with polynomials, letting $R_k(n) = \binom{n}{k}n!$. The recurrence for such resource functions is given as follows:

$$R_k(n) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \wedge n = 0 \\ (k+1) \cdot R_{k+1}(n-1) + (2k+1) \cdot R_k(n-1) + k \cdot R_{k-1}(n-1) & \textit{otherwise} \end{cases}$$

However, note that this recurrence for $R_k(n)$ depends on the higher-index term $R_{k+1}(n-1)$. The dependence on increasing indices makes it difficult to finitely bound how many such indices should be considered by the AARA index system. Perhaps future work can find a way around this obstacle.

Step Functions

It is possible to get a linear recurrence giving a step function, providing energy only for data structures exceeding a certain size. By mixing such step functions with other resource functions, costs can be found that depend only on the sizes of larger data structures. To develop such a recurrence for a size cutoff k, let $R_k(n)$ be defined as follows:

$$R_k(n) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \wedge n = 0 \\ R_{k-1}(n-1) & \textit{otherwise} \end{cases}$$

6.8 Related Work

While this work seems to be the first to use Stirling numbers as a basis for exponential functions, various work has previously considered exponential functions.

In AARA

Aside from logarithms [89], there has not really been other work in AARA concerned with deriving non-polynomial cost bounds.[9]

The one place that exponential resource functions were previously mentioned in the context of AARA was the in the work of Hofmann and Moser [88]. This work adapts AARA to term rewriting and uses tree automata to generate multivariate resource functions (which I discuss more in Chapter 7). They briefly point out that exponential cost bounds can be generated by such tree automata, but this idea is explored no further. It is not clear if their resource functions actually subsume the range of resource functions generated by Stirling numbers of the second kind (which are proven to be maximally expressive in Section 6.5).

[9]I speculate that this lack of development could be because there are not many other common classes of cost functions outside of (mixes of) polynomials, exponentials, and logarithms. Perhaps the next most likely cost bounds of interest would be factorials, roots, or—in a multivariate setting—maxima.

Outside of AARA

Outside of AARA, many other automatic cost analysis systems have harnessed the power of linear recurrences for exponential cost analyses. This ubiquity is because linear recurrences are easily solvable, and much work in automatic cost analysis is based on solving recurrences, as discussed in Chapter 4. Other approaches also come upon exponential cost bounds from other angles, like ranking functions [31].

Chapter 7

Multivariate Exponentials

This chapter is a direct continuation of Chapter 6, extending the support of exponential cost bounds to the multivariate setting. So far, this thesis has only dealt with so-called *univariate* AARA, where resource functions are linear combinations of single-variable terms like $x^2 + 5 \cdot y$. However, this chapter (and this chapter alone) now makes the foray into *multivariate* resource functions [65, 80], which are linear combinations of *products of* single-variable terms like $x \cdot y^2 + x + 5 \cdot z \cdot y$. Such multivariate resource functions are not only more expressive, but also are critical for reasoning about key invariants between multiple data structures, such as between an argument and accumulator in a tail-recursive function.

Some observations of Chapter 6 are key to making this chapter's multivariate system work. This chapter relies intimately on the properties of offset Stirling numbers of the second kind. In particular, this work makes indispensable use of offset Stirling numbers' combinatorial interpretation.

This chapter is terminal work in this thesis. That is, no other chapter builds off of this chapter's system.

7.1 The Problem: Tail Recursion and Accumulators

While multivariate cost bounds might seem rather exotic, they can turn out to be critical for reasoning about how some simple code transforms potential energy. This importance is especially apparent when observing the analysis behaviour on common tail-recursive code patterns making use of accumulators. For an example, consider the reversing function `rev` in Figure 7.1.

```
fun revApp (a,b) =
  case a of
  | [] -> b
  | x::xs -> revApp (xs, x::b)

fun rev lst = revApp (lst, [])
```

Figure 7.1: Code for `rev` and `revApp`

This code is a standard[1] tail-recursive implementation of such a function and uses a helper function `revApp` with a list accumulator.

Suppose that `rev` incurs no cost, and thus only transforms potential energy. If one calls `rev` on a list of length n carrying n^2 units of energy, then one should expect the output list to also carry n^2 units of energy, as the output has the same length. However, this expectation is trickier than it seems. To assign n^2 energy to the output list, it would be necessary for intermediate program states in `revApp` to represent this quantity in terms of the sizes of the lists `a` and `b`. Since $|\mathtt{a}|+|\mathtt{b}| = n$, this representation takes the form of $n^2 = (|\mathtt{a}|+|\mathtt{b}|)^2 = |\mathtt{a}|^2+2\cdot|\mathtt{a}|\cdot|\mathtt{b}|+|\mathtt{b}|^2$, which includes the multivariate term $2\cdot|\mathtt{a}|\cdot|\mathtt{b}|$. Thus, to analyze such functions, Hoffmann et al. have introduced multivariate polynomial resource functions [80]. This work has since become the base of the most developed implementation of an AARA typechecker at the time of writing, *Resource Aware ML*.

The same problem arises using exponential energy. Suppose `rev` is called on a list of length n carrying 2^n units of energy. Then intermediate lists `a` and `b` need to represent $2^n = 2^{|a|+|b|} = 2^{|a|}\cdot 2^{|b|}$ units of energy, which also is multivariate. Thus, support of multivariate exponential energy is needed for exponential AARA to analyze code like `rev`.

Much of the development of multivariate cost bounds can follow exactly as in the work of Hoffmann, Aehlig, and Hofmann [80]. However, there are two key wrinkles to using their work approach: in their work, both sharing and the definition of potential energy are intimately intertwined with properties of binomial coefficients. These are the two main problems that my work must overcome to adapt the approach to multivariate exponentials.

Sharing Sharing is tricky to define in a multivariate setting. For example, if some data structure x with quadratic energy is split between x's identical copies y and z, then the energy quantity $|x|^2$, could be represented by either one of the univariate polynomials $|y|^2$ or $|z|^2$, or by the multivariate polynomial $|y|\cdot|z|$. The multivariate option depends heavily on the fact that the product of linear functions is a quadratic function. For the best results, it is important that AARA can make use of all such relations between polynomials of any degree.

It turns out that the the best way to accomplish this goal is to show that the conical space of resource functions is closed under the introduction of products. For polynomials, this property boils down to showing that $\binom{n}{i}\cdot\binom{n}{j}$ is always equal to the conical combination $\sum_k c_k.\binom{n}{k}$ for some coefficients $c_k \geq 0$ independent of n. This choice of c_k is actually unique because binomial coefficients are linearly independent functions. As a result, conical combination of resource functions of the form $\binom{n}{i}\cdot\binom{n}{j}$ can be mapped in one and only one way to equivalent conical combinations of resource functions of the form $\binom{n}{k}$. This mapping yields the annotation map for sharing $\curlyvee_x^{y,z}$, where $n = |x| = |y| = |z|$. The trick in this chapter will be to show the that conical space of exponential resource functions is likewise closed under the introduction of products.

Potential Energy Definition The only preexisting multivariate versions of AARA [65, 80] assign potential energy via an intricate pattern-matching annotation index system. For example, the quadratic potential energy of a unit list x is represented with an annotation index of a matching type, in this case a list of two units, $[\langle\rangle, \langle\rangle]$. This annotation index represents energy counted by

[1]The standard library of, e.g., OCaml implements `rev` in exactly this way.

the number of ways the index can pattern match the list using a special notion of matching. In this case, this index matches any ordered sequence of two (possibly-non-adjacent) unit values in the list. That is, it counts the number of ways to select two units in the list x, which is $\binom{|x|}{2}$ as desired. In general, a list of k units gives the annotation index for $\binom{|x|}{k}$.

While the previous pattern-match counting is not too bad, it can quickly get complicated. For example, $[[\langle\rangle, \langle\rangle], [\langle\rangle]]$ counts a rather exotic property of unit list lists depending upon both outer and inner list lengths. More specifically, for a list x, it counts $\sum_{0 \le i < j < |x|} \binom{|x_i|}{2} \cdot \binom{|x_j|}{1}$, where x_i is the i^{th} (0-indexed) list in x. This kind of complex dependence is important to support because it arises naturally when breaking up nested lists into a head and tail—there should be multivariate annotations concerning, e.g., the product of the sizes of the head list and tail list.

Unfortunately, these systems rely heavily on combinatorial properties of binomial coefficients. Thus they are unsuited for the Stirling numbers that represent exponentials. A new annotation index system must be developed based combinatorial properties of Stirling numbers to properly support multivariate exponentials.

7.2 The Linear Idea: More Recurrences

In this section I provide the key ideas to solve the problems identified in Section 7.1. These ideas both work by leveraging the recurrence for Stirling numbers. Later sections of this chapter use these ideas to define the desired multivariate exponential system.

7.2.1 Conical Closure

To begin to solve the problems around sharing, I provide Lemma 7.2.1. This lemma gives the necessary relation between conical combinations of products of offset Stirling numbers and conical combinations of Stirling numbers simpliciter. The provided proof relies on the linear recurrence for Stirling numbers given in Chapter 6. More specifically, it relies on the recurrence for mixing Stirling numbers with themselves.

Lemma 7.2.1 (offset Stirling number conical products)**.** *For all finite sets of coefficients $a_{i,j} \ge 0$ there exists a unique finite set of coefficients $b_k \ge 0$ such that*

$$\sum_{i,j \ge 0} a_{i,j} \cdot \left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\} = \sum_{k \ge 0} b_k \cdot \left\{ {n+1 \atop k+1} \right\}$$

Proof. To prove this statement, it suffices to show that it holds for the singleton conical combination of one basis resource function $\left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\}$. This simplified statement suffices because of the following:

$$\begin{aligned}
\sum_{i,j\geq 0} a_{i,j} \cdot \left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\} &= \sum_{i,j\geq 0} a_{i,j} \cdot (\sum_{k\geq 0} c_{i,j,k} \left\{ {n+1 \atop k+1} \right\}) && \textit{singleton case} \\
&= \sum_{k\geq 0} (\sum_{i,j\geq 0} a_{i,j} \cdot c_{i,j,k}) \cdot \left\{ {n+1 \atop k+1} \right\} && \textit{algebra}
\end{aligned}$$

The uniqueness of this representation follows because offset Stirling numbers are linearly independent. Thus, $b_k = \sum_{i,j\geq 0} a_{i,j} \cdot c_{i,j,k}$ suffices.

It now remains to be shown that for all i, j, there exists a finite set of coefficients $c_{i,j,k} \geq 0$ such that

$$\left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\} = \sum_{k\geq 0} c_{i,j,k} \cdot \left\{ {n+1 \atop k+1} \right\}$$

To show that this property holds, define $c_{i,j,k}$ inductively as follows. It can then be checked that this definition suffices, where higher lines take priority if multiple cases match.

$$c_{i,j,k} = \begin{cases} 1 & i = 0 \wedge j = k \\ 1 & j = 0 \wedge i = k \\ 0 & i = 0 \vee j = 0 \vee k = 0 \\ 0 & (i+1)\cdot(j+1) \leq k \\ ((i+1)\cdot(j+1) - k - 1)\cdot c_{i,j,k-1} & \\ \quad +(i+1)\cdot c_{i-1,j,k-1} + (j+1)\cdot c_{i,j-1,k-1} + c_{i-1,j,k-1} & \textit{otherwise} \end{cases}$$

To check this definition's sufficiency, it must be shown that $c_{i,j,k} \geq 0$ for all $i, j, k \geq 0$ and that these coefficients induce the equality $\left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\} = \sum_{k\geq 0} c_{i,j,k} \cdot \left\{ {n+1 \atop k+1} \right\}$

To show nonnegativity, one can induct on the definition of $c_{i,j,k}$. The only nontrival case is the final one where $(i+1)\cdot(j+1) \geq k+1$, which leaves every coefficient nonnegative in the linear combination of subcases. By induction, the subcases are also nonnegative, so the whole linear combination is also nonnegative. Thus the only nontrivial case of the induction holds, completing the nonnegativity proof.

To continue, observe that i and j are treated symmetrically, so it suffices to consider only the cases where $i \leq j$. Finally, this definition's sufficiency can be confirmed by inducting on n and i to show that the needed equality holds.

n≥0,i=0

$$\begin{aligned}
\left\{ n+1 \atop i+1 \right\} \cdot \left\{ n+1 \atop j+1 \right\} &= \left\{ n+1 \atop 1 \right\} \cdot \left\{ n+1 \atop j+1 \right\} && i = 0 \\
&= \left\{ n+1 \atop j+1 \right\} && \left\{ n+1 \atop 1 \right\} = 1 \\
&= c_{0,j,j} \cdot \left\{ n+1 \atop j+1 \right\} && c_{0,j,j} = 1 \\
&= \sum_{k \geq 0} c_{0,j,k} \cdot \left\{ n+1 \atop k+1 \right\} && j \neq k \implies c_{0,j,k} = 0
\end{aligned}$$

n=0,i>0 This case works as follows, recalling that $j > 0$ since $j \geq i$.

$$\begin{aligned}
&\left\{ n+1 \atop i+1 \right\} \cdot \left\{ n+1 \atop j+1 \right\} \\
&= \left\{ 1 \atop i+1 \right\} \cdot \left\{ n+1 \atop j+1 \right\} && n = 0 \\
&= 0 && i > 0 \implies \left\{ 1 \atop i+1 \right\} = 0 \\
&= \sum_{k > 0} c_{i,j,k} \left\{ 1 \atop k+1 \right\} && k > 0 \implies \left\{ 1 \atop k+1 \right\} = 0 \\
&= \sum_{k > 0} c_{i,j,k} \left\{ n+1 \atop k+1 \right\} && n = 0 \\
&= \sum_{k \geq 0} c_{i,j,k} \left\{ n+1 \atop k+1 \right\} && i, j > 0 \implies c_{i,j,0} = 0
\end{aligned}$$

n>0,i>0 This case works as follows, recalling that $j > 0$ since $j \geq i$.

$$
\begin{aligned}
&\left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {n+1 \atop j+1} \right\} \\
&= (i+1)\cdot(j+1)\cdot \left\{ {n \atop i+1} \right\} \cdot \left\{ {n \atop j+1} \right\} + (i+1)\cdot \left\{ {n \atop i} \right\} \cdot \left\{ {n \atop j+1} \right\} \\
&\quad + (j+1)\cdot \left\{ {n \atop i+1} \right\} \cdot \left\{ {n \atop j} \right\} + \left\{ {n \atop i} \right\} \cdot \left\{ {n \atop j} \right\} && \textit{recurrence} \\
&= (i+1)\cdot(j+1)\cdot \sum_{k\geq 0} c_{i,j,k} \cdot \left\{ {n \atop k+1} \right\} + (i+1)\cdot \sum_{k\geq 0} c_{i-1,j,k} \cdot \left\{ {n \atop k+1} \right\} \\
&\quad + (j+1)\cdot \sum_{k\geq 0} c_{i,j-1,k} \cdot \left\{ {n \atop k+1} \right\} + \sum_{k\geq 0} c_{i-1,j-1,k} \cdot \left\{ {n \atop k+1} \right\} && IH \\
&= \sum_{k\geq 0} (((i+1)\cdot(j+1) - k - 1)\cdot c_{i,j,k} + (i+1)\cdot c_{i-1,j,k} + (j+1)\cdot c_{i,j-1,k} + c_{i-1,j-1,k}) \cdot \left\{ {n \atop k+1} \right\} \\
&\quad + \sum_{k\geq 0} c_{i,j,k} \cdot (k+1) \cdot \left\{ {n \atop k+1} \right\} && \textit{algebra} \\
&= \sum_{k\geq 0} c_{i,j,k+1} \cdot \left\{ {n \atop k+1} \right\} + \sum_{k\geq 0} c_{i,j,k} \cdot (k+1) \cdot \left\{ {n \atop k+1} \right\} && \textit{def} \\
&= \sum_{k\geq 1} c_{i,j,k} \cdot \left\{ {n \atop k} \right\} + \sum_{k\geq 0} c_{i,j,k} \cdot (k+1) \cdot \left\{ {n \atop k+1} \right\} && \textit{reindexing} \\
&= \sum_{k\geq 0} c_{i,j,k} \cdot \left\{ {n \atop k} \right\} + \sum_{k\geq 0} c_{i,j,k} \cdot (k+1) \cdot \left\{ {n \atop k+1} \right\} && i,j>0 \implies c_{i,j,0}=0 \\
&= \sum_{k\geq 0} c_{i,j,k} \cdot ((k+1)\cdot \left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\}) && \textit{algebra} \\
&= \sum_{k\geq 0} c_{i,j,k} \cdot \left\{ {n+1 \atop k+1} \right\} && \textit{recurrence}
\end{aligned}
$$

□

Using the construction for coefficients defined in Lemma 7.2.1, one can find that, e.g., $\left\{ {n+1 \atop 2} \right\}^2$ is equivalent to $6 \cdot \left\{ {n+1 \atop 4} \right\} + 6 \cdot \left\{ {n+1 \atop 3} \right\} + \left\{ {n+1 \atop 2} \right\}$. Moreover, as guaranteed by the lemma, every coefficient in this linear combination of Stirling numbers is nonnegative, so it lies in the conical space of offset Stirling numbers.

Because the full space of resource functions in this multivariate system includes at least the offset Stirling numbers, Lemma 7.2.1 suggests that sharing could be possible to define using only conical combinations. However, the full multivariate system includes complicated variations of resource functions that do not clearly reduce to the case handled by this lemma. Thus, the general case must be proven in Section 7.6.

7.2.2 Combinatorial Counting

To find a replacement for the pattern-match counting of previous multivariate work, it is necessary to take a deeper look at the combinatorial interpretation afforded by offset Stirling numbers. Only then can one properly extend the Stirling-number recurrence to the multivariate setting. I give the high-level overview of my solution here, and the formal development in Section 7.5.

The development of the multivariate polynomial pattern-match counting is motivated by the combinatorial interpretation of binomial coefficients and their recurrence $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$.

Because $\binom{n}{k}$ counts the number of ways to select k elements out of n, the ideas of previous work is to develop a mechanism that naturally picks out this selection. The trick is then to extend this mechanism in a principled manner so it can be recursively applied over all the various values that might arise in the system. One can develop a natural mechanism for this purpose by studying the binomial coefficient recurrence. Indeed, generalizing this recurrence has been key to extending multivariate AARA to regular recursive types [65].

In the end, the right mechanism turns out to be a flavor of pattern-matching. List matching in particular just selects list elements, and these list elements can then be recursively matched at their types. The resulting counting method closely matches how Pascal's identity for $\binom{n+1}{k+1}$ can be derived:

- skip the first element and select all $k+1$ elements from the remaining n (counting $\binom{n}{k+1}$)
- select the first element and then select k elements from the remaining n (counting $\binom{n}{k}$)

When an element is selected, the pattern-matching can then naturally recurse on that element, extending to nested lists in a principled manner.

For multivariate exponentials, a similar mechanism is needed to count offset Stirling numbers. Using the combinatorial understanding of Stirling numbers, the offset Stirling number $\left\{ {n+1 \atop k+1} \right\}$ counts the number of ways to partition all $n+1$ elements into $k+1$ disjoint nonempty subsets. However, this interpretation does not map well onto the task at hand because $\left\{ {n+1 \atop k+1} \right\}$ is the resource function for a list of length n—there is an off-by-one issue, and thus this interpretation cannot naturally select elements of the list to recurse on.

A better interpretation to use is the combinatorial interpretation specific to offset Stirling numbers: $\left\{ {n+1 \atop k+1} \right\}$ counts the number of ways to select k nonempty disjoint subsets of n elements.[2] Then the idea should be to recursively match list elements according to which of these selected subsets they belong to (if any). For ease of reference, consider these subsets in the order of their highest-index list elements. This ordering means that the "first" subset is the first to be exhausted when consuming list elements in their natural order. This interpretation yields the following way of deriving the recurrence $\left\{ {n+1 \atop k+1} \right\} = (k+1) \cdot \left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\}$:

- do not put the first element in any subset, so that all k nonempty subsets come from the remaining n elements (counting $\left\{ {n \atop k+1} \right\}$, giving the 1 in $(k+1) \cdot \left\{ {n \atop k+1} \right\}$)
- pick the first element as an element of one of the k subsets, but not as the last element, so that the remaining $n-1$ elements must still be distributed among k nonempty subsets, (counting $\left\{ {n \atop k+1} \right\}$ a total of k times, giving the k in $(k+1) \cdot \left\{ {n \atop k+1} \right\}$)
- pick the first element as the last element of some subset—this subset must be the first in subset order—and then the remaining $n-1$ elements must be distributed among $k-1$ nonempty subsets (counting $\left\{ {n \atop k} \right\}$)

7.3 Annotation Indices

The multivariate annotation indices of a type τ are given by the set $MInd(\tau)$ defined in Figure 7.2. To simplifiy presentation, this definition uses the same convention from Figure 3.5

[2]This interpretation can be derived from the previous by discarding the partition that contains the $n+1^{st}$ element.

$$MInd(\mathbb{1}) = \{\langle\rangle\} \qquad MInd(\tau \xrightarrow{\vec{a}|\vec{b}} \sigma) = \{\langle\rangle\} \qquad MInd(\tau \otimes \sigma) = \langle MInd(\tau),\, MInd(\sigma)\rangle$$

$$MInd(\tau \oplus \sigma) = \mathtt{l}(MInd(\tau)) \cup \mathtt{r}(MInd(\sigma)) \qquad MInd(L(\tau)) = \{[\,]\} \cup MInd(\tau) :: MInd(L(\tau))$$

$$MInd(T(\tau)) = \{[\,]\} \cup MInd(\tau) :: MInd(T(\tau))$$

$$MInd(\Gamma) = \prod_{x \in \mathtt{dom}(\Gamma)} \{x \mapsto i \mid i \in MInd(\Gamma(x))\}$$

Figure 7.2: Multivariate annotation indices

that operations distribute over sets. The annotation indices for a type τ take the form of values of the type τ (with a few exceptions), which differs from the path-based indices used in the rest of this thesis. These values are used in Section 7.5 to dictate patterns for combinatorial pattern-matching. The number of ways these patterns match define the resource functions these annotations indices correspond to.

The exceptions to the rule that $MInd(\tau)$ gives the values of τ are functions and trees. These exceptions occur for two distinct reasons which I outline in the following paragraphs.

For functions, I use the unit value $\langle\rangle$ because otherwise there would be many function values that all behave identically. Functions do not carry any potential energy, so the choice of index does not matter.

For trees, I use indices that coincide with list values rather than tree values. This choice follows the development in Hoffmann's thesis[3] [76], where trees are treated as lists of their elements in preorder for potential energy purposes, as defined in Definition 7.3.1. The later work of Grosen et al. shows that trees (and regular recursive types in general) can be given natural multivariate annotation indices that coincide with their values [65], but I leave such generalization for future work. Further, the most mature AARA implementation Resource Aware ML (RaML) [81] supports multivariate resource functions using the trees-as-lists method, so this chapter's work is suited for implementation in RaML.

Definition 7.3.1 (preorder)**.** *The preorder traversal of a tree is given by the map* `pre` *from trees to lists. Letting* $@$ *be the list append operation,* `pre` *is defined by the following:*

$$\mathtt{pre}(\mathtt{leaf}) = [\,]$$

$$\mathtt{pre}(\mathtt{node}(v_1,\, v_2,\, v_3)) = v_1 :: (\mathtt{pre}(v_2) @ \mathtt{pre}(v_3))$$

Despite being modeled after values instead of paths, the multivariate annotation indices given in Figure 7.2 are all quite close to their univariate counterparts. I detail the interesting differences in the following paragraphs.

[3]Hoffmann's thesis gives the tree details that are missing from Hoffmann, Aehlig, and Hofmann's work [80].

The first key difference is that products, and by extension whole type contexts, are essentially indexed by Cartesian products of their elements' indices. Because these structures index multiple elements, these annotation indices have the ability correspond to resource functions that depend on the properties of multiple values. This dependency on multiple values is what makes multivariate resource functions possible.

Another point of interest is that type contexts do not have any special annotation index for free energy like $\mathtt{c}$. This circumstance occurs because all types already naturally include their own indices for that purpose, and the labelled collection of these indices is the appropriate such index for the type context. For example, the free energy index of $\mathbb{1}$ is $\langle\rangle$, and that of $\mathbb{1}\otimes\mathbb{1}$ is $\langle\langle\rangle,\, \langle\rangle\rangle$. In turn, the free energy index of the context $x : \mathbb{1}, y : \mathbb{1}\otimes\mathbb{1}$ is the value context $x \mapsto \langle\rangle, y \mapsto \langle\langle\rangle,\, \langle\rangle\rangle$. The free energy indices of an arbitrary type/context are given by $\mathtt{C}$ in Definition 7.3.2. The fact that these indices pick out constant resource functions comes to fruition with the definition of potential energy in Section 7.5.

Definition 7.3.2 (free energy annotation indices)**.** *The function* $\mathtt{C}(\tau)$ *gives the set of annotation indices yielding the constant resource function for a given type* τ*.*

$$\mathtt{C}(\tau) = \begin{cases} \{\langle\rangle\} & \tau = \mathbb{1} \vee \tau = \sigma \xrightarrow{\vec{a}|\vec{b}} \rho \\ \langle \mathtt{C}(\sigma),\, \mathtt{C}(\rho)\rangle & \tau = \sigma \otimes \rho \\ \mathtt{l}(\mathtt{C}(\sigma)) \cup \mathtt{r}(\mathtt{C}(\rho)) & \tau = \sigma \oplus \rho \\ \{[\,]\} & \tau = L(\sigma) \vee \tau = T(\sigma) \end{cases}$$

Additionally, to extend over contexts, $\mathtt{C}(\Gamma) = \prod_{x\in\mathtt{dom}(\Gamma)} \{x \mapsto i \mid i \in \mathtt{C}(\Gamma(x))\}$.

Finally, one might notice that the definition of Figure 7.2 actually yields infinitely many annoation indices for a given list or tree. In other chapters of this thesis, the systems bake in a finite cutoff like D_{max}. In this chapter, however, the cutoff is left implicit. There are more parameters one might consider to bound the multivariate annotation indices, and determining the best cutoff can be left up to the user. It only matters that some cutoff is used, as AARA's current method of automation can only handle the inference of finitely many annotations.

7.4 Types

The types for multivariate AARA are almost identical to the univariate types from Section 5.3.1. However, there is one change concerning function types $\tau \xrightarrow{\vec{a}|\vec{b}} \sigma$. This change is that the annotation indices in the domains of the annotation vectors $\vec{a}$ and $\vec{b}$ need to account for the new annotation index system. For this purpose, $\vec{a}$ is indexed by $MInd(\mathtt{arg} : \tau)$ and $\vec{b}$ is indexed by $MInd(\mathtt{arg} : \tau, \mathtt{ret} : \sigma)$. There is no more need for a free energy index like $\mathtt{c}$ for the same reasons discussed in Section 7.3.

$$\Phi(V : \Gamma \mid a) = \begin{cases} \infty & \bullet \textit{ used in } V \\ \sum_{i \in MInd(\Gamma)} a(i) \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) & \textit{otherwise} \end{cases} \qquad \phi_{\langle\rangle}(v) = 1$$

$$\phi_{\mathtt{r}(i)}(\mathtt{r}(v)) = \phi_i(v) \qquad \phi_{\mathtt{l}(i)}(\mathtt{l}(v)) = \phi_i(v) \qquad \phi_{\mathtt{r}(i)}(\mathtt{l}(v)) = 0 \qquad \phi_{\mathtt{l}(i)}(\mathtt{r}(v)) = 0$$

$$\phi_{\langle i,\, j\rangle}(\langle v_1,\, v_2\rangle) = \phi_i(v_1) \cdot \phi_i(v_2) \qquad \phi_{[\,]}(v) = 1 \qquad \phi_{j \,::\, js}([\,]) = 0$$

$$\phi_{j \,::\, js}(v :: vs) = \phi_{j \,::\, js}(vs) + \phi_j(v) \cdot \phi_{js}(vs) + \sum_{i \in j \,::\, js} \phi_i(v) \cdot \phi_{j \,::\, js}(vs)$$

$$\phi_i(v) = \phi_i(\mathtt{pre}(v)) \;\; (\textit{for } v : T(\tau))$$

Figure 7.3: Multivariate potential energy definition

7.5 Potential Energy

The potential energy of a well-formed context $V : \Gamma$ with annotation map a is given by the expression $\Phi(V : \Gamma \mid a)$ defined in Figure 7.3. Because of the multivariate nature of this energy, it does not make as much sense to talk of the energy of a single value as is done in other chapters; the energy depends on all values that exist in a context.

As a technical detail, this definition first includes a singular special case[4] for handling the nonterminal dummy value $\bullet$ (see Sections 2.4 and 3.6). With this case handled, I assume that no other values are or contain the dummy value for the remainder of this section.

The definition in Figure 7.3 is then further broken down by annotation index—rather than by value as in other chapters—to better show off the way that each annotation index contributes. This contribution is formalized by $\phi_i(v)$, which is a heavily value-dependent resource function. One should think of $\phi_i(v)$ as something like $\left\{ {v+1 \atop i+1} \right\}$, a generalization of the offset Stirling numbers over arbitrary programatic values rather than just numbers. Indeed, letting $\mathbb{N} = L(\mathbb{1})$ recovers the numerical offset Stirling numbers.

Despite the different presentation from other chapters, one still finds that this definition is linear and monotone in its annotation values. The linearity is structurally apparent, as $\Phi(V : \Gamma \mid a)$ is defined as a linear combination with coefficients from a. The monotonicity then follows because the elements of the linear combination are all inductively constructed out of sums and products of nonnegative numbers. Thus, suitable analogues of Lemmas 3.4.5 and 3.4.6 still apply.

The resource functions for lists in Figure 7.3 generate the desired exponential functions by following the combinatorial meaning of the recurrence for offset Stirling numbers discussed in Section 7.2. (Note that in this definition, I write $i \in j :: js$ as shorthand for i in the *multiset* of list elements in $j :: js$, which may include repetitions.) One can see exactly how these resource

[4]This special case avoids concerns about the well-definedness of $0 \cdot \infty$. However, I believe it would also work to define $0 \cdot \infty = \infty$ for the purposes of resource functions, so that the results of this section would follow with no special cases.

functions generalize the desired exponential resource functions via Lemma 7.5.1.

Lemma 7.5.1 (multivariate list energy)**.** *Let $S_k(n)$ give the set of k-element sets of nonempty, disjoint subsets of integers in $[1, n]$. Order the integer sets in each set of sets $s \in S_k(n)$ according to the order of their largest elements, and let s_j pick out the j^{th} smallest such integer set in s. Then the energy of a list $[v_1, ..., v_n]$ at a given multivariate annotation index $[i_1, ..., i_k]$ can be given the following closed form:*

$$\phi_{[i_1,...,i_k]}([v_1, ..., v_n]) = \sum_{s \in S_k(n)} \prod_{j=1}^{k} \prod_{m \in s_j} \phi_{i_j}(v_m)$$

Proof. This identity is proven by induction on n and k.

n≥0,k=0 In this case, $S_0(n) = \{\emptyset\}$ because $\emptyset$ is the unique 0-element set.

$$\begin{aligned} \phi_{[\,]}([v_1, ..., v_n]) &= 1 && \textit{def} \\ &= \prod_{j=1}^{0} \prod_{m \in \emptyset} \phi_{i_j}(v_m) && \textit{empty product} \\ &= \sum_{s \in \{\emptyset\}} \prod_{j=1}^{0} \prod_{m \in \emptyset} \phi_{i_j}(v_m) && \textit{singleton sum} \end{aligned}$$

n=0,k>0 In this case, $S = \emptyset$ because there are no nonempty subsets of the empty range.

$$\begin{aligned} \phi_{[i_1,...,i_k]}([\,]) &= 0 && \textit{def} \\ &= \sum_{s \in \emptyset} \prod_{j=1}^{k} \prod_{m \in s_j} \phi_{i_j}(v_m) && \textit{empty sum} \end{aligned}$$

n>0,k>0 This case follows from some bound manipulation alongside the inductive hypothesis.

$$\begin{aligned}
&\phi_{[i_1,...,i_k]}([v_1,...,v_n]) \\
&= \phi_{[i_1,...,i_k]}([v_2,...,v_n]) + \phi_{i_1}(v_1)\cdot\phi_{[i_2,...,i_k]}([v_2,...,v_n]) \\
&\quad + \sum_{\ell=1}^{k}\phi_{i_\ell}(v_1)\cdot\phi_{[i_1,...,i_k]}([v_2,...,v_n]) && \textit{def} \\
&= \sum_{s\in S_k(n-1)}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_{m+1}) + \phi_{i_1}(v_1)\cdot\sum_{s\in S_{k-1}(n-1)}\prod_{j=1}^{k-1}\prod_{m\in s_j}\phi_{i_j}(v_{m+1}) \\
&\quad + \sum_{\ell=1}^{k}\phi_{i_\ell}(v_1)\cdot\sum_{s\in S_k(n-1)}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_{m+1}) && IH \\
&= \sum_{\substack{s\in S_k(n)\\ s.t.\,\forall j.\,1\notin s_j}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) + \phi_{i_1}(v)\cdot\sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}=s_1}}\prod_{j=2}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) \\
&\quad + \sum_{\ell=1}^{k}\phi_{i_\ell}(v_1)\cdot\sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}\neq s_1\wedge\exists j.\,1\in s_j}}\prod_{j=1}^{k}\prod_{\substack{m\in s_j\\ s.t.\,m\neq 1}}\phi_{i_j}(v_m) && \textit{add \& avoid } v_1 \\
&= \sum_{\substack{s\in S_k(n)\\ s.t.\,\forall j.\,1\notin s_j}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) + \sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}=s_1}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) \\
&\quad + \sum_{\ell=1}^{k}\sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}\neq s_1\wedge\exists j.\,1\in s_j}}\phi_{i_\ell}(v_1)\cdot\prod_{j=1}^{k}\prod_{\substack{m\in s_j\\ s.t.\,m\neq 1}}\phi_{i_j}(v_m) && \textit{distribution} \\
&= \sum_{\substack{s\in S_k(n)\\ s.t.\,\forall j.\,1\notin s_j}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) + \sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}=s_1}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) \\
&\quad + \sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}\neq s_1\wedge\exists j.\,1\in s_j}}\sum_{\ell=1}^{k}\phi_{i_\ell}(v_1)\cdot\prod_{j=1}^{k}\prod_{\substack{m\in s_j\\ s.t.\,m\neq 1}}\phi_{i_j}(v_m) && \textit{commute sums} \\
&= \sum_{\substack{s\in S_k(n)\\ s.t.\,\forall j.\,1\notin s_j}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) + \sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}=s_1}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) \\
&\quad + \sum_{\substack{s\in S_k(n)\\ s.t.\,\{1\}\neq s_1\wedge\exists j.\,1\in s_j}}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) && \textit{distribution} \\
&= \sum_{s\in S_k(n)}\prod_{j=1}^{k}\prod_{m\in s_j}\phi_{i_j}(v_m) && \textit{disjoint domains}
\end{aligned}$$

□

Additionally, it can now be shown via Lemma 7.5.2 that the free energy annotation indices act as desired. Note that, due to sum types, it may be that the *sum* of multiple indices' contributions is what yields the constant resource function $\lambda x.\, 1$, as opposed to just a singular annotation index.

Lemma 7.5.2 (free energy annotation indices yield constant)**.** *If $v : \tau$, then*

$$\sum_{i \in \mathtt{C}(\tau)} \phi_i(v) = 1$$

Proof. This statement is proven by induction over the derivation of the well-formedness statement $v : \tau$.

units and functions Suppose the well-formedness statement $v : \tau$ concludes with one of the rules *V-Unit* or *V-Fun*. Then $\mathtt{C}(\tau) = \{\langle\rangle\}$, and the following equalities hold:

$$\begin{aligned}
\sum_{i \in \mathtt{C}(\tau)} \phi_i(v) &= \sum_{i \in \{\langle\rangle\}} \phi_i(v) && \mathit{def} \\
&= \phi_{\langle\rangle}(v) && \mathit{substitution} \\
&= 1 && \mathit{def}
\end{aligned}$$

lists and trees Suppose the well-formedness statement $v : \tau$ concludes with one of the rules *V-Nil,V-Cons,V-Leaf*, or *V-Node*. Then $\mathtt{C}(\tau) = \{[\,]\}$, and the following equalities hold:

$$\begin{aligned}
\sum_{i \in \mathtt{C}(\tau)} \phi_i(v) &= \sum_{i \in \{[\,]\}} \phi_i(v) && \mathit{def} \\
&= \phi_{[\,]}(v) && \mathit{substitution} \\
&= 1 && \mathit{def}
\end{aligned}$$

V-Pair Suppose the well-formedness derivation concludes with *V-Pair* so that $v = \langle v_1,\, v_2\rangle$ and $\tau = \sigma \otimes \rho$. Then $v_1 : \sigma$ and $v_2 : \rho$ by inversion, and the following equalities hold:

$$\begin{aligned}
\sum_{i \in \mathtt{C}(\sigma\otimes\rho)} \phi_i(\langle v_1,\, v_2\rangle) &= \sum_{i \in \langle \mathtt{C}(\sigma),\, \mathtt{C}(\rho)\rangle} \phi_i(\langle v_1,\, v_2\rangle) && \textit{def} \\
&= \sum_{i \in \mathtt{C}(\sigma), j \in \mathtt{C}(\rho)} \phi_{\langle i,\, j\rangle}(\langle v_1,\, v_2\rangle) && \textit{substitution} \\
&= \sum_{i \in \mathtt{C}(\sigma), j \in \mathtt{C}(\rho)} \phi_i(v_1) \cdot \phi_j(v_2) && \textit{def} \\
&= \sum_{i \in \mathtt{C}(\sigma)} \phi_i(v_1) \cdot \sum_{j \in \mathtt{C}(\rho)} \phi_j(v_2) && \textit{algebra} \\
&= 1 \cdot 1 && IH \\
&= 1 && \textit{algebra}
\end{aligned}$$

V-SumL Suppose the well-formedness derivation concludes with *V-SumL* so that $v = \mathtt{l}(v')$ and $\tau = \sigma \oplus \rho$. Then $v' : \sigma$ by inversion, and the following equalities hold:

$$\begin{aligned}
\sum_{i \in \mathtt{C}(\sigma\oplus\rho)} \phi_i(\mathtt{l}(v')) &= \sum_{i \in \mathtt{l}(\mathtt{C}(\sigma) \cup \mathtt{r}(\mathtt{C}(\rho)))} \phi_i(\mathtt{l}(v')) && \textit{def} \\
&= \sum_{i \in \mathtt{l}(\mathtt{C}(\sigma))} \phi_i(\mathtt{l}(v')) + \sum_{i \in \mathtt{r}(\mathtt{C}(\rho))} \phi_i(\mathtt{l}(v')) && \textit{algebra} \\
&= \sum_{i \in \mathtt{C}(\sigma)} \phi_{\mathtt{l}(i)}(\mathtt{l}(v')) + \sum_{i \in \mathtt{C}(\rho)} \phi_{\mathtt{r}(i)}(\mathtt{l}(v')) && \textit{substitution} \\
&= \sum_{i \in \mathtt{C}(\sigma)} \phi_i(v') + \sum_{i \in \mathtt{C}(\rho)} 0 && \textit{def} \\
&= 1 + \sum_{i \in \mathtt{C}(\rho)} 0 && IH \\
&= 1 && \textit{algebra}
\end{aligned}$$

$$\mathtt{Sh}(\langle\rangle, \langle\rangle) = \{\langle\rangle\} \qquad \mathtt{Sh}(\mathtt{l}(i), \mathtt{l}(j)) = \mathtt{l}(\mathtt{Sh}(i,j)) \qquad \mathtt{Sh}(\mathtt{r}(i), \mathtt{r}(j)) = \mathtt{r}(\mathtt{Sh}(i,j))$$

$$\mathtt{Sh}(\mathtt{l}(i), \mathtt{r}(j)) = \emptyset \qquad \mathtt{Sh}(\mathtt{r}(i), \mathtt{l}(j)) = \emptyset \qquad \mathtt{Sh}(\langle i_1,\, i_2\rangle, \langle j_1,\, j_2\rangle) = \langle \mathtt{Sh}(i_1, j_1),\, \mathtt{Sh}(i_2, j_2)\rangle$$

$$\mathtt{Sh}([i_1, ..., i_m], [j_1, ..., j_n]) = \mathtt{sel}_{i,j}(\mathtt{shuff}(m,n)) \qquad \mathtt{sel}_{i,j}(\mathtt{l}(p)) = \{i_p\}$$

$$\mathtt{sel}_{i,j}(\mathtt{r}(p)) = \{j_p\} \qquad \mathtt{sel}_{i,j}(\langle p, q\rangle) = \mathtt{Sh}(i_p, j_q)$$

Figure 7.4: Sharing annotation index definition

V-SumR Suppose the well-formedness derivation concludes with *V-SumR* so that $v = \mathtt{r}(v')$ and $\tau = \sigma \oplus \rho$. Then $v' : \rho$ by inversion, and the following equalities hold:

$$\begin{aligned}
\sum_{i \in \mathtt{C}(\sigma\oplus\rho)} \phi_i(\mathtt{r}(v')) &= \sum_{i \in \mathtt{l}(\mathtt{C}(\sigma)\cup\mathtt{r}(\mathtt{C}(\rho)))} \phi_i(\mathtt{r}(v')) && \textit{def} \\
&= \sum_{i \in \mathtt{l}(\mathtt{C}(\sigma))} \phi_i(\mathtt{r}(v')) + \sum_{i \in \mathtt{r}(\mathtt{C}(\rho))} \phi_i(\mathtt{r}(v')) && \textit{algebra} \\
&= \sum_{i \in \mathtt{C}(\sigma)} \phi_{\mathtt{l}(i)}(\mathtt{r}(v')) + \sum_{i \in \mathtt{C}(\rho)} \phi_{\mathtt{r}(i)}(\mathtt{r}(v')) && \textit{substitution} \\
&= \sum_{i \in \mathtt{C}(\sigma)} 0 + \sum_{i \in \mathtt{C}(\rho)} \phi_i(v') && \textit{def} \\
&= \sum_{i \in \mathtt{C}(\sigma)} 0 + 1 && \textit{IH} \\
&= 1 && \textit{algebra}
\end{aligned}$$

□

One effect of this way of setting up potential energy is that values of types that do not normally carry any energy, like functions and units, now appear to do so, as they have free energy annotations. However, this appearance does not take into account the multiplicative nature of the multivariate setting. Because resource functions are constructed through multiplication, the correct way for a value to contribute no energy is to multiply by 1 rather than to add 0.

7.6 Sharing

This section is set aside to provide the machinery needed to formalize sharing. The key is to be able to properly show that the space of resource functions is closed under products, which is mostly a matter of combinatorial algebra.

Figure 7.4 contains the definition of the sharing annotation index function Sh, which takes $i, j \in \mathit{MInd}(\tau)$ and returns a multiset of annotation indices. These indices are eventually used to determine how to properly share energy through manipulating annotations, as proven in Lemma 7.6.2. The function `Sh` plays the same role for annotation indices as $c_{i,j,k}$ did in Lemma 7.2.1, which is proven formally in Lemma 7.6.1.

As a notational convention, all operations in the definition of the function `Sh` are treated as if they distribute over multisets. For example, a list containing a multiset should be interpreted as a multiset of lists. The definition case for lists makes use of this notational convention, as well as the the special shuffle function defined in Definition 7.6.1 to appropriately shuffle list indices, as well as the function `sel` to map these list indices to corresponding annotation indices.

Definition 7.6.1 (special shuffle)**.** *The special shuffle function* `shuff` *is used to "shuffle" elements of two lists in a special way to include pairs of the two lists' elements. The function* `shuff` *accomplishes this goal by giving the appropriate shuffles of the list indices, either paired or tagged to indicate which list they should be associated with. Additionally, not every shuffle is desirable—*`shuff` *must satisfy certain ordering conditions.*
Formally, the function `shuff` *takes two numbers n and m and returns all lists of elements from* $\mathtt{l}([1,m]) \cup \mathtt{r}([1,n]) \cup \langle[1,m],\ [1,n]\rangle$ *that satisfy the following conditions:*

- *the list contains no repetitions*
- *each element $p \in [1,m]$ is present somewhere, either as $\mathtt{l}(p)$ or as $\langle p,\ -\rangle$*
- *each element $q \in [1,n]$ is present somewhere, either as $\mathtt{r}(q)$ or as $\langle -,\ q\rangle$*
- *for $p < m$, the last time $\mathtt{l}(p)$ or some $\langle p,\ -\rangle$ is present in the list is prior to the last time $\mathtt{l}(p+1)$ or some $\langle p+1,\ -\rangle$ is*
- *for $q < n$, the last time $\mathtt{r}(q)$ or some $\langle -,\ q\rangle$ is present in the list is prior to the last time $\mathtt{r}(q+1)$ or some $\langle -,\ q+1\rangle$ is*

This definition is motivated by the proof of Lemma 7.6.1.

Example 7.6.1. To exemplify `shuff`, consider $\mathtt{shuff}(1,2)$. This set contains the following lists up to length 3. It also contains many more lists of lengths 4 through 6.

$[\langle 1,\ 1\rangle, \mathtt{r}(2)]$ $[\mathtt{r}(1), \langle 1,\ 2\rangle]$ $[\langle 1,\ 1\rangle, \langle 1,\ 2\rangle]$ $[\mathtt{l}(1), \mathtt{r}(1), \mathtt{r}(2)]$ $[\mathtt{r}(1), \mathtt{l}(1), \mathtt{r}(2)]$

$[\mathtt{r}(1), \mathtt{r}(2), \mathtt{l}(1)]$ $[\langle 1,\ 1\rangle, \mathtt{r}(1), \mathtt{r}(2)]$ $[\mathtt{r}(1), \langle 1,\ 1\rangle, \mathtt{r}(2)]$ $[\langle 1,\ 1\rangle, \mathtt{l}(1), \mathtt{r}(2)]$

$[\langle 1,\ 1\rangle, \mathtt{r}(2), \mathtt{l}(1)]$ $[\mathtt{l}(1), \langle 1,\ 1\rangle, \mathtt{r}(2)]$ $[\langle 1,\ 2\rangle, \mathtt{r}(1), \mathtt{r}(2)]$ $[\mathtt{r}(1), \langle 1,\ 2\rangle, \mathtt{r}(2)]$

$[\mathtt{r}(1), \mathtt{r}(2), \langle 1,\ 2\rangle]$ $[\mathtt{r}(1), \langle 1,\ 2\rangle, \mathtt{l}(1)]$ $[\mathtt{r}(1), \mathtt{l}(1), \langle 1,\ 2\rangle]$ $[\mathtt{l}(1), \mathtt{r}(1), \langle 1,\ 2\rangle]$

$[\langle 1,\ 1\rangle, \langle 1,\ 2\rangle, \mathtt{r}(2)]$ $[\langle 1,\ 1\rangle, \mathtt{r}(2), \langle 1,\ 2\rangle]$ $[\mathtt{r}(2), \langle 1,\ 1\rangle, \langle 1,\ 2\rangle]$ $[\langle 1,\ 1\rangle, \mathtt{r}(1), \langle 1,\ 2\rangle]$

$[\mathtt{r}(1), \langle 1,\ 1\rangle, \langle 1,\ 2\rangle]$ $[\langle 1,\ 1\rangle, \langle 1,\ 2\rangle, \mathtt{l}(1)]$ $[\langle 1,\ 1\rangle, \mathtt{l}(1), \langle 1,\ 2\rangle]$ $[\mathtt{l}(1), \langle 1,\ 1\rangle, \langle 1,\ 2\rangle]$

Lemma 7.6.1 (resource function product closure)**.** *For any $v : \tau$ and $i, j \in \mathit{MInd}(\tau)$,*

$$\phi_i(v) \cdot \phi_j(v) = \sum_{k \in \mathtt{Sh}(i,j)} \phi_k(v)$$

Proof. This statement is proven by induction over the type τ.

units and functions Suppose τ is a unit or function type. Then $\mathit{MInd}(\tau) = \{\langle\rangle\}$, so $i = j = \langle\rangle$ and the following equalities then hold:

$$\begin{aligned}
\phi_{\langle\rangle}(v) \cdot \phi_{\langle\rangle}(v) &= 1 \cdot \phi_{\langle\rangle}(v) && \mathit{def} \\
&= \sum_{k \in \{\langle\rangle\}} \phi_k(v) && \mathit{algebra} \\
&= \sum_{k \in \mathtt{Sh}(\langle\rangle, \langle\rangle)} \phi_k(v) && \mathit{def}
\end{aligned}$$

products Suppose $\tau = \sigma \otimes \rho$. Then $i, j \in \mathit{MInd}(\sigma \otimes \rho)$ ensures that i is some $\langle i_1,\, i_2\rangle$ and j is some $\langle j_1,\, j_2\rangle$. Further, only rule that could conclude $v : \sigma \otimes \rho$ is *V-Pair*, so $v = \langle v_1,\, v_2\rangle$, and both $v_1 : \sigma$ and $v_2 : \rho$ hold by inversion. Finally, the following equalities hold:

$$\begin{aligned}
\phi_{\langle i_1, i_2\rangle}(\langle v_1,\, v_2\rangle) \cdot \phi_{\langle j_1, j_2\rangle}(\langle v_1,\, v_2\rangle) &= \phi_{i_1}(v_1) \cdot \phi_{i_2}(v_2) \cdot \phi_{j_1}(v_1) \cdot \phi_{j_2}(v_2) && \mathit{def} \\
&= (\phi_{i_1}(v_1) \cdot \phi_{j_1}(v_1)) \cdot (\phi_{i_2}(v_2) \cdot \phi_{j_2}(v_2)) && \mathit{algebra} \\
&= (\sum_{k \in \mathtt{Sh}(i_1, j_1)} a_k \cdot \phi_k(v_1)) \cdot (\sum_{\ell \in \mathtt{Sh}(i_2, j_2)} b_\ell \cdot \phi_\ell(v_2)) && \mathit{IH} \\
&= \sum_{\substack{k \in \mathtt{Sh}(i_1, j_1) \\ \ell \in \mathtt{Sh}(i_2, j_2)}} \phi_k(v_1) \cdot \phi_\ell(v_2) && \mathit{algebra} \\
&= \sum_{\substack{k \in \mathtt{Sh}(i_1, j_1) \\ \ell \in \mathtt{Sh}(i_2, j_2)}} \phi_{\langle k,\, \ell\rangle}(\langle v_1,\, v_2\rangle) && \mathit{def} \\
&= \sum_{k \in \langle \mathtt{Sh}(i_1, j_1),\, \mathtt{Sh}(i_2, j_2)\rangle} \phi_k(\langle v_1,\, v_2\rangle) && \mathit{def} \\
&= \sum_{k \in \mathtt{Sh}(\langle i_1, i_2\rangle, \langle j_1, j_2\rangle)} \phi_k(\langle v_1,\, v_2\rangle) && \mathit{def}
\end{aligned}$$

sums Suppose $\tau = \sigma \oplus \rho$. Then $\mathit{MInd}(\sigma \oplus \rho)$ says that i is either $\mathtt{l}(i')$ or $\mathtt{r}(i')$ and j is either $\mathtt{l}(j')$ or $\mathtt{r}(j')$. Further, only two rules that could conclude $v : \sigma \otimes \rho$: *V-SumL* where $v = \mathtt{l}(v')$ for $v' : \sigma$, or *V-SumR* where $v = \mathtt{r}(v')$ for $v' : \rho$.

Consider the subcase where all of i, j, v have the same tag out of $\mathtt{l}(-), \mathtt{r}(-)$. These options are symmetric, so without loss of generality, let it be $\mathtt{l}(-)$. Then the following equalities hold:

$$\begin{aligned}
\phi_{\mathtt{l}(i')}(\mathtt{l}(v')) \cdot \phi_{\mathtt{l}(j')}(\mathtt{l}(v')) &= \phi_{i'}(v') \cdot \phi_{j'}(v') && \textit{def} \\
&= \sum_{k \in \mathtt{Sh}(i',j')} \phi_k(v') && IH \\
&= \sum_{k \in \mathtt{Sh}(i',j')} \phi_{\mathtt{l}(k)}(\mathtt{l}(v')) && \textit{def} \\
&= \sum_{\mathtt{l}(k) \in \mathtt{l}(\mathtt{Sh}(i',j'))} \phi_{\mathtt{l}(k)}(\mathtt{l}(v')) && \textit{def} \\
&= \sum_{k \in \mathtt{Sh}(\mathtt{l}(i'),\mathtt{l}(j'))} \phi_k(\mathtt{l}(v')) && \textit{def}
\end{aligned}$$

Now consider the subcase where i and j have matching tags, but v does not. In particular, The available options are symmetric, so without loss of generality, let it be that $i = \mathtt{l}(i')$, $j = \mathtt{l}(j')$, and $v = \mathtt{r}(v')$. Then the following equalities hold:

$$\begin{aligned}
\phi_{\mathtt{l}(i')}(\mathtt{r}(v')) \cdot \phi_{\mathtt{l}(j')}(\mathtt{r}(v')) &= 0 && \textit{def} \\
&= \sum_{k \in \mathtt{Sh}(i',j')} 0 && \textit{algebra} \\
&= \sum_{k \in \mathtt{Sh}(i',j')} \phi_{\mathtt{l}(k)}(\mathtt{r}(v')) && \textit{def} \\
&= \sum_{\mathtt{l}(k) \in \mathtt{l}(\mathtt{Sh}(i',j'))} \phi_{\mathtt{l}(k)}(\mathtt{r}(v')) && \textit{def} \\
&= \sum_{k \in \mathtt{Sh}(\mathtt{l}(i'),\mathtt{l}(j'))} \phi_k(\mathtt{r}(v')) && \textit{def}
\end{aligned}$$

Finally, consider letting i and j have mismatched tags, so $\mathtt{Sh}(i, j) = \emptyset$. Because they differ, v cannot match both tags. This circumstance allows the needed finite sets to be chosen to be empty as follows:

$$\begin{aligned}
\phi_i(v) \cdot \phi_j(v) &= 0 && \textit{def} \\
&= \sum_{k \in \emptyset} \phi_k(v) && \textit{algebra} \\
&= \sum_{k \in \mathtt{Sh}(i,j)} \phi_k(v) && \textit{def}
\end{aligned}$$

trees This case reduces to the list case in particular. Suppose $\tau = T(\sigma)$. Then $i, j \in \mathit{MInd}(T(\sigma))$ iff $i, j \in \mathit{MInd}(L(\sigma))$ because the set $\mathit{MInd}(T(\sigma))$ is equal to $\mathit{MInd}(L(\sigma))$. Finally, the following equalities hold:

$$\begin{aligned}
\phi_i(v) \cdot \phi_j(v) &= \phi_i(\mathtt{pre}(v)) \cdot \phi_j(\mathtt{pre}(v)) && \textit{def} \\
&= \sum_{k \in \mathtt{Sh}(i,j)} \phi_k(\mathtt{pre}(v)) && IH \\
&= \sum_{k \in \mathtt{Sh}(i,j)} \phi_k(v) && \textit{def}
\end{aligned}$$

lists Suppose $\tau = L(\sigma)$. Then $i, j \in \mathit{MInd}(L(\sigma))$ ensures that $i = [i_1, ..., i_p]$ for some $p \geq 0$. and $j = [j_1, ..., j_q]$ for some $q \geq 0$. Further, the well-formedness rules *V-Cons* and *V-Nil* entail that $v = [v_1, ..., v_n]$ for some $n \geq 0$.

Now let $\sqcap_{s,s'}(\mathtt{l}(p)) = s_p \setminus \bigcup s'$, $\sqcap_{s,s'}(\mathtt{r}(q)) = s'_q \setminus \bigcup s$, and $\sqcap(\langle p, q\rangle) = s_p \cap s'_q$. Then the following equalities hold:

$$\begin{aligned}
&\phi_{[i_1,...,i_p]}([v_1, ..., v_n]) \cdot \phi_{[j_1,...,j_q]}([v_1, ..., v_n]) \\
&= \left(\sum_{s \in S_p(n)} \prod_{p'=1}^{p} \prod_{r \in s_{p'}} \phi_{i_{p'}}(v_r)\right) \left(\sum_{s' \in S_q(n)} \prod_{q'=1}^{q} \prod_{r \in s_{q'}} \phi_{j_{q'}}(v_r)\right) && \textit{Lemma } 7.5.1 \\
&= \sum_{\substack{s \in S_p(n) \\ s' \in S_q(n)}} \left(\prod_{p' \in [1,p], r \in s_{p'}} \phi_{i_{p'}}(v_r)\right) \left(\prod_{q' \in [1,q], r \in s'_{q'}} \phi_{j_{q'}}(v_r)\right) && \textit{distribution} \\
&= \sum_{\substack{s \in S_p(n) \\ s' \in S_q(n)}} \left(\prod_{\substack{p' \in [1,p], q' \in [1,q] \\ r \in s_{p'} \cap s'_{q'}}} \phi_{i_{p'}}(v_r) \cdot \phi_{j_{q'}}(v_r)\right) \\
&\quad \left(\prod_{p' \in [1,p], r \in s_{p'} \setminus \bigcup s'} \phi_{i_{p'}}(v_r)\right) \left(\prod_{q' \in [1,q], r \in s'_{q'} \setminus \bigcup s} \phi_{j_{q'}}(v_r)\right) && \textit{disjoint domains} \\
&= \sum_{\substack{s \in S_p(n) \\ s' \in S_q(n)}} \left(\prod_{\substack{p' \in [1,p], q' \in [1,q] \\ r \in s_{p'} \cap s'_{q'}}} \sum_{k \in \mathtt{Sh}(i_{p'}, j_{q'})} \phi_k(v_r)\right) \\
&\quad \left(\prod_{p' \in [1,p], r \in s_{p'} \setminus \bigcup s'} \phi_{i_{p'}}(v_r)\right) \left(\prod_{q' \in [1,q], r \in s'_{q'} \setminus \bigcup s} \phi_{j_{q'}}(v_r)\right) && IH
\end{aligned}$$

$$= \sum_{\substack{s\in S_p(n)\\ s'\in S_q(n)}} \left(\prod_{\substack{p'\in[1,p],q'\in[1,q]\\ r\in\sqcap_{s,s'}(\langle p',q'\rangle)}} \sum_{k\in\mathtt{sel}_{i,j}(\langle p',q'\rangle)} \phi_k(v_r) \right)$$

$$\left(\prod_{\substack{p'\in[1,p]\\ r\in\sqcap_{s,s'}(\mathtt{l}(p'))}} \sum_{k\in\mathtt{sel}_{i,j}(\mathtt{l}(p'))} \phi_k(v_r) \right) \left(\prod_{\substack{q'\in[1,q]\\ r\in\sqcap_{s,s'}(\mathtt{r}(q'))}} \sum_{k\in\mathtt{sel}_{i,j}(\mathtt{r}(p'))} \phi_k(v_r) \right) \qquad \textit{def}$$

$$= \sum_{\substack{s\in S_p(n)\\ s'\in S_q(n)}} \prod_{\substack{\ell\in\mathtt{l}([1,p])\cup\mathtt{r}([1,q])\cup\langle[1,p],[1,q]\rangle\\ r\in\sqcap_{s,s'}(\ell)}} \sum_{k\in\mathtt{sel}_{i,j}(\ell)} \phi_k(v_r) \qquad \textit{disjoint domains}$$

Now observe that the set of pairs $s \in S_p(n), s' \in S_q(n)$ is in bijection with the set of pairs $t \in S_u(n), [\ell_1, ..., \ell_u] \in \mathtt{shuff}(p,q)$. (In fact, the number of such length-u element of $\mathtt{shuff}(p,q)$ is $c_{p,q,u}$ as defined in Lemma 7.2.1.) This bijection can be shown as follows:

First one can construct t, ℓ from s, s'. Given $s \in S_p(n), s' \in S_q(n)$, the appropriate t can be constructed by the following:

$$t = \bigcup_{\substack{\ell \text{ s.t.}\\ \sqcap_{s,s'}(\ell)\neq\emptyset}} \sqcap_{s,s'}(\ell)$$

and the corresponding $\ell \in \mathtt{shuff}$ is just the list of the union's domain under the usual ordering. Then to show the reverse, it suffices to construct s and s' via their intersections according to $t_{u'} = \sqcap_{s,s'}(\ell_{u'})$. Specifically,

$$s_{p'} = \bigcup_{\substack{u' \text{ s.t.}\\ \ell_{u'}=\mathtt{l}(p')}} t_{u'} \cup \bigcup_{\substack{u' \text{ s.t.}\\ \ell_{u'}=\langle p',q'\rangle}} t_{u'} \qquad s'_{q'} = \bigcup_{\substack{u' \text{ s.t.}\\ \ell_{u'}=\mathtt{r}(q')}} t_{u'} \cup \bigcup_{\substack{u' \text{ s.t.}\\ \ell_{u'}=\langle p',q'\rangle}} t_{u'}$$

The previous equality chain can then be completed as follows:

$$
\begin{aligned}
&= \sum_{\substack{s\in S_p(n)\\ s'\in S_q(n)}} \prod_{\substack{\ell\in\mathtt{l}([1,p])\cup\mathtt{r}([1,q])\cup\langle[1,p],\,[1,q]\rangle\\ r\in\sqcap_{s,s'}(\ell)}} \sum_{k\in\mathtt{sel}_{i,j}(\ell)} \phi_k(v_r) \\
&= \sum_{\substack{t\in S_u(n)\\ [\ell_1,\ldots,\ell_u]\in\mathtt{shuff}(p,q)}} \prod_{\substack{u'\in[1,u]\\ r\in t_{\ell'_u}}} \sum_{k\in\mathtt{sel}_{i,j}(\ell_{u'})} \phi_k(v_r) && \textit{correspondence} \\
&= \sum_{\ell\in\mathtt{shuff}(p,q)} \sum_{t\in S_{|\ell|}(n)} \prod_{u'=1}^{|\ell|} \prod_{r\in t_{u'}} \sum_{k\in\mathtt{sel}_{i,j}(\ell_{u'})} \phi_k(v_r) && \textit{algebra} \\
&= \sum_{\ell\in\mathtt{shuff}(p,q)} \sum_{k\in\mathtt{sel}_{i,j}(\ell)} \sum_{t\in S_{|\ell|}(n)} \prod_{u'=1}^{|\ell|} \prod_{r\in t_{u'}} \phi_{k_{u'}}(v_r) && \textit{distribution} \\
&= \sum_{k\in\mathtt{sel}_{i,j}(\mathtt{shuff}(p,q))} \sum_{t\in S_{|k|}(n)} \prod_{u'=1}^{|k|} \prod_{r\in t_{u'}} \phi_{k_{u'}}(v_r) && \textit{algebra} \\
&= \sum_{k\in\mathtt{sel}_{i,j}(\mathtt{shuff}(p,q))} \phi_k(v) && \textit{Lemma 7.5.1} \\
&= \sum_{k\in\mathtt{Sh}(i,j)} \phi_k(v) && \textit{def}
\end{aligned}
$$

□

As a consequence of Lemma 7.6.1, conical combinations of resource functions are closed under products. This property can be directly extended to define a sharing relation

Definition 7.6.2 (multivariate sharing)**.** *The multivariate sharing function $\hat{\curlyvee}_z^{x,y}$ is a multivariate generalization of the previous univariate sharing function $\curlyvee_z^{x,y}$. The multivariate sharing function is used to transform an annotation map a with domain elements of the form $(V, x \mapsto i, y \mapsto j)$ from $MInd(\Gamma, x : \tau, y : \tau)$ to annotation map with domain elements of the form $(V, z \mapsto k)$ from $MInd(\Gamma, z : \tau)$ in a way that conserves potential energy. This function is defined as follows:*

$$\hat{\curlyvee}_z^{x,y}(a)(V, z \mapsto k) = \sum_{i,j \text{ s.t. } k\in\mathtt{Sh}(i,j)} a(V, x \mapsto i, y \mapsto j)$$

Lemma 7.6.2 (multivariate sharing conserves energy)**.**

$$\Phi((V, x \mapsto v, y \mapsto v) : (\Gamma, x : \tau, y : \tau) \mid a) = \Phi((V, z \mapsto v) : (\Gamma, z : \tau) \mid \hat{\curlyvee}_z^{x,y}(a))$$

Proof. If the dummy value $\bullet$ is used in V or v, then this equality holds with both sides ∞. Otherwise, let $p_{V'}$ stand for $\prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w))$. Then the following equalities hold:

$$
\begin{aligned}
& \Phi((V, x \mapsto v, y \mapsto v) : (\Gamma, x : \tau, y : \tau) \mid a) \\
= & \sum_{i \in \mathit{MInd}(\Gamma, x:\tau, y:\tau)} a(i) \cdot \phi_{i(x)}(v) \cdot \phi_{i(y)}(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{i(w)}(V(w)) && \mathit{def} \\
= & \sum_{\substack{V', i, j \text{ s.t.} \\ (V', x \mapsto i, y \mapsto j) \in \\ \mathit{MInd}(\Gamma, x:\tau, y:\tau)}} a(V', x \mapsto i, y \mapsto j) \cdot \phi_i(v) \cdot \phi_j(v) \cdot p_{V'} && \mathit{def} \\
= & \sum_{\substack{V', i, j \text{ s.t.} \\ (V', x \mapsto i, y \mapsto j) \in \\ \mathit{MInd}(\Gamma, x:\tau, y:\tau)}} a(V', x \mapsto i, y \mapsto j) \cdot \left(\sum_{k \in \mathtt{Sh}(i,j)} \phi_k(v) \right) \cdot p_{V'} && \text{Lemma 7.6.1} \\
= & \sum_{\substack{V', k, \text{ s.t.} \\ (V', z \mapsto k) \in \\ \mathit{MInd}(\Gamma, z:\tau)}} \left(\sum_{i,j \text{ s.t } k \in \mathtt{Sh}(i,j)} a(V', x \mapsto i, y \mapsto j) \right) \cdot \phi_k(v) \cdot p_{V'} && \mathit{distribution} \\
= & \sum_{\substack{V', k, \text{ s.t.} \\ (V', z \mapsto k) \in \\ \mathit{MInd}(\Gamma, z:\tau)}} \hat{\Upsilon}_z^{x,y}(a)(V', z \mapsto k) \cdot \phi_k(v) \cdot p_{V'} && \mathit{def} \\
= & \sum_{i \in \mathit{MInd}(\Gamma, z:\tau)} \hat{\Upsilon}_z^{x,y}(a)(i) \cdot \phi_{i(z)}(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{i(w)}((V, z \mapsto v)(w)) && \mathit{def} \\
= & \Phi((V, z \mapsto v) : \Gamma, z : \tau \mid \hat{\Upsilon}_z^{x,y}(a)) && \mathit{def}
\end{aligned}
$$

□

7.7 Shifting

This section is set aside to provide the machinery needed to formalize shifting. The key is to be able to properly show that the space of resource functions is closed under the shift operation, which is mostly a matter of combinatorial algebra.

Firstly, some special setup is needed for trees. Because a tree acts like the list of its elements in preorder, breaking up a tree into its two subtrees is like splitting a list into two. There are many ways such a split could occur, whereas the analagous breaking up of a list always gives a list of size exactly one less. Luckily, there is a way to shift tree annotations that works independently of the sizes of the subtrees. This method is given across Lemmas 7.7.1 and 7.7.2, and it is a generalization of the identity $\left\{ {n+1 \atop k+1} \right\} = \sum_{\substack{i,j \in [0,k] \\ i+j \geq k}} \left\{ {n+1-m \atop i+1} \right\} \cdot \left\{ {m+1 \atop j+1} \right\} \cdot \binom{i}{i+j-k} \cdot \binom{j}{i+j-k} \cdot (j+i-k)!$ which holds for any $m \in [0, n]$. First, this method uses a special "list cutting" operation `cut` in Definition 7.7.1 that enumerates pairs of lists of indices that divide a list of a given size into two.

Then these indices are used similarly to the indices in the function `shuff` from Section 7.6 to constructively show the needed closure properties.

Definition 7.7.1 (special cutting)**.** *The special cutting operation* `cut` *is used to "cut" a list into two lists in a special way to allow both parts to share some elements. The function* `cut` *accomplishes this goal by taking some number n and returning all pairs of lists a, b of indices up to n that satisfy the following properties, where @ is the append operation for lists:*

- *each list contains no repetitions*
- *each element $i \in [1, n]$ is present in either a or b (or both)*
- *for $i < n$, the last time i is present in $a@b$ is prior to the last time $i + 1$ is*

This definition is motivated by the proof of Lemma 7.7.1

Example 7.7.1. To exemplify `cut`, consider $\mathtt{cut}(2)$. This set contains the following pairs of lists:

$$\langle[1],\,[1,2]\rangle \quad \langle[2],\,[1,2]\rangle \quad \langle[1,2],\,[1,2]\rangle \quad \langle[2,1],\,[1,2]\rangle \quad \langle[1,2],\,[2]\rangle \quad \langle[2,1],\,[2]\rangle$$

$$\langle[\,],\,[1,2]\rangle \qquad \langle[1],\,[2]\rangle \qquad \langle[1,2],\,[\,]\rangle$$

Lemma 7.7.1 (list cutting)**.** *Let @ be the append operation for lists. Further let $\mathtt{pick}_{[j_1,\ldots,j_n]}([i_1, \ldots, i_m]) = [j_{i_1}, \ldots, j_{i_m}]$ for $m \leq n$, and let it distribute over sets and pairs. Then for any pair of lists $v_1 : L(\tau)$ and $v_2 : L(\tau)$ and annotation index $i \in MInd(L(\tau))$,*

$$\phi_i(v_1 @ v_2) = \sum_{\langle j,\,k\rangle \in \mathtt{pick}_i(\mathtt{cut}(|i|))} \phi_j(v_1) \cdot \phi_k(v_2)$$

Proof. This statement is proven by induction on the structure of v_1.

$\mathbf{v_1 = [\,]}$ If $v_1 = [\,]$, then $\phi_j(v_1) = 0$ for all j except for $j = [\,]$, where $\phi_{[\,]}([\,]) = 1$. Moreover, the only pair of lists j, k from $\mathtt{cut}(|i|)$ where $j = [\,]$ has k contain exactly the elements of $[1, |i|]$ in order, as all elements must be present with no repetitions. Then the following equalities hold:

$$\begin{aligned}
\phi_i([\,] @ v_2) &= \phi_i(v_2) && [\,] @ v_2 = v_2 \\
&= \phi_{[\,]}([\,]) \cdot \phi_i(v_2) && \mathit{def} \\
&= \sum_{\langle [\,],\,k\rangle \in \mathtt{pick}_i(\mathtt{cut}(|i|))} \phi_{[\,]}([\,]) \cdot \phi_k(v_2) && \langle [\,],\,k\rangle \in \mathtt{pick}_i(\mathtt{cut}(|i|)) \implies \mathtt{pick}_i(k) = i \\
&= \sum_{\langle j,\,k\rangle \in \mathtt{pick}_i(\mathtt{cut}(|i|))} \phi_j([\,]) \cdot \phi_k(v_2) && j \neq [\,] \implies \phi_j([\,]) = 0
\end{aligned}$$

$\mathbf{v_1 = v :: vs}$ If $v_1 = v :: vs$, then there are two cases to consider depending on whether $i = [\,]$.

If $i = [\,]$, the following equalities hold:

$$\begin{aligned}
\phi_{[\,]}((v :: vs) @ v_2) &= 1 && \textit{def} \\
&= \phi_{[\,]}(v :: vs) \cdot \phi_{[\,]}(v_2) && \textit{def} \\
&= \sum_{\langle j,k\rangle \in \mathtt{pick}_{[\,]}(\{\langle [\,],[\,]\rangle\})} \phi_j(v :: vs) \cdot \phi_k(v_2) && \textit{algebra} \\
&= \sum_{\langle j,k\rangle \in \mathtt{pick}_{[\,]}(\mathit{cut}(0))} \phi_j(v :: vs) \cdot \phi_k(v_2) && \mathit{cut}(0) = \{\langle [\,], [\,]\rangle\}
\end{aligned}$$

Alternatively, if the annotation index in question is some $i :: is$, then the following equalities hold:

$$\begin{aligned}
&\phi_{i :: is}((v :: vs) @ v_2) \\
&= \phi_{i :: is}(v :: (vs @ v_2)) && (v :: vs) @ v_2 = v :: (vs @ v_2) \\
&= \phi_i(v) \cdot \phi_{is}(vs @ v_2) + \phi_{i :: is}(vs @ v_2) + \sum_{i' \in i :: is} \phi_{i'}(v) \cdot \phi_{i :: is}(vs @ v_2) && \textit{def} \\
&= \phi_i(v) \cdot \sum_{\langle j,k\rangle \in \mathtt{pick}_{is}(\mathtt{cut}(|is|))} \phi_j(vs) \cdot \phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i :: is}(\mathtt{cut}(|i :: is|))} \phi_j(vs) \cdot \phi_k(v_2) \\
&\quad + \sum_{i' \in i :: is} \phi_{i'}(v) \cdot \sum_{\langle j,k\rangle \in \mathtt{pick}_{i :: is}(\mathtt{cut}(|i :: is|))} \phi_j(vs) \cdot \phi_k(v_2) && IH \\
&= \phi_i(v) \cdot \sum_{\langle j,k\rangle \in \mathtt{pick}_{is}(\mathtt{cut}(|is|))} \phi_j(vs) \cdot \phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i :: is}(\mathtt{cut}(|i :: is|))} \phi_j(vs) \cdot \phi_k(v_2) \\
&\quad + \sum_{i' \in i :: is} \phi_{i'}(v) \cdot \sum_{\substack{\langle j,k\rangle \in \mathtt{pick}_{i :: is}(\mathtt{cut}(|i :: is|)) \\ s.t.\, i' \in j}} \phi_j(vs) \cdot \phi_k(v_2) \\
&\quad + \sum_{i' \in i :: is} \phi_{i'}(v) \cdot \sum_{\substack{\langle j,k\rangle \in \mathtt{pick}_{i :: is}(\mathtt{cut}(|i :: is|)) \\ s.t.\, i' \notin j}} \phi_j(vs) \cdot \phi_k(v_2) && \textit{disjoint domains}
\end{aligned}$$

$$\begin{aligned}
&= \sum_{\langle j,k\rangle \in \mathtt{pick}_{is}(\mathtt{cut}(|is|))} \phi_i(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \sum_{j'\in j} \phi_{j'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{i'\in i::is} \sum_{\substack{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|)) \\ s.t.\; i'\notin j}} \phi_{i'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) && \textit{distribution} \\
&= \sum_{\langle j,k\rangle \in \mathtt{pick}_{is}(\mathtt{cut}(|is|))} \phi_i(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \sum_{j'\in j} \phi_{j'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{i'\in i::is} \sum_{\substack{\langle i'::j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|)) \\ s.t.\; i'\in k}} \phi_{i'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) && i'\in j@k,\ \textit{no repetitions} \\
&= \sum_{i'\in i::is} \sum_{\substack{\langle i'::j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|)) \\ s.t.\; i'\notin k}} \phi_{i'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \sum_{j'\in j} \phi_{j'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{i'\in i::is} \sum_{\substack{\langle i'::j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|)) \\ s.t.\; i'\in k}} \phi_{i'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) && \substack{\langle i'::j,k\rangle\in\mathtt{cut}(|i::is|) \\ \wedge\; i'\notin k} \implies i'=1 \\
&= \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \sum_{j'\in j} \phi_{j'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{i'\in i::is} \sum_{\langle i'::j,k\rangle \in \mathtt{pick}_{i\,::\,is}(\mathtt{cut}(|i::is|))} \phi_{i'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) && \textit{disjoint domains}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{\langle j,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \sum_{j'\in j} \phi_{j'}(v)\cdot\phi_j(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_j(v)\cdot\phi_{js}(vs)\cdot\phi_k(v_2) && \textit{algebra} \\
&= \sum_{\langle [\,],k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{[\,]}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{j\,::\,js}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle [\,],k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \sum_{j'\in[\,]} \phi_{j'}(v)\cdot\phi_{[\,]}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \sum_{j'\in j::js} \phi_{j'}(v)\cdot\phi_{j\,::\,js}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_j(v)\cdot\phi_{js}(vs)\cdot\phi_k(v_2) && \textit{dijoint domains} \\
&= \sum_{\langle [\,],k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{[\,]}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{j\,::\,js}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \sum_{j'\in j::js} \phi_{j'}(v)\cdot\phi_{j\,::\,js}(vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_j(v)\cdot\phi_{js}(vs)\cdot\phi_k(v_2) && j'\notin[\,] \\
&= \sum_{\langle [\,],k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{[\,]}(v::vs)\cdot\phi_k(v_2) \\
&\quad + \sum_{\langle j::js,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_{j\,::\,js}(v::vs)\cdot\phi_k(v_2) && \textit{distribution, def} \\
&= \sum_{\langle j,k\rangle \in \texttt{pick}_{i\,::\,is}(\texttt{cut}(|i::is|))} \phi_j(v::vs)\cdot\phi_k(v_2) && \textit{disjoint domains}
\end{aligned}
$$

□

With that setup out of the way, this section can progress to defining the shifting operation in Definition 7.7.2. Just as in other chapters, shifting is designed to be energy-conserving. This fact is proven in Lemma 7.7.2. Later sections can then make successful use of multivariate shifting.

Definition 7.7.2 (multivariate shifting)**.** *The multivariate exponential shifting operator $\hat{\lhd}$ is the multivariate exponential version of the shifting operator $\lhd$, and it is used to relate the annotations of a context with a list or tree to those of the same context with the list or tree's parts.*
For lists, $\hat{\lhd}^x_{y,z}$ takes an annotation map a with domain of indices in $MInd(\Gamma, x : L(\tau))$ and returns an annotation map b with domain of indices in $MInd(\Gamma, y : \tau, z : L(\tau))$. Let $\#(i, j)$ give the number of times the annotation index i occurs in the annotation index list j. Then, formally:

$$\hat{\lhd}^x_{y,z}(a)(V, y \mapsto i, z \mapsto j) = a(V, x \mapsto i :: j) + \#(i, j) \cdot a(V, x \mapsto j) + \begin{cases} a(V, x \mapsto j) & i \in \mathtt{C}(\tau) \\ 0 & i \notin \mathtt{C}(\tau) \end{cases}$$

For trees, $\hat{\lhd}^t_{x,y,z}$ takes an annotation a with domain of indices in $MInd(\Gamma, t : T(\tau))$ and returns an annotation map b with domain of indices in $MInd(\Gamma, x : T(\tau), y : \tau, z : T(\tau))$. This case of the shifting definition makes use of the previous case for lists. Formally:

$$\hat{\lhd}^t_{x,y,z}(a)(V, x \mapsto i, y \mapsto j, z \mapsto k) = \sum_{\substack{\ell\, s.t. \\ \langle i, k\rangle \in \mathtt{pick}_\ell(\mathtt{cut}(|\ell|))}} \hat{\lhd}^t_{y,w}(a)(V, y \mapsto j, w \mapsto \ell)$$

Lemma 7.7.2 (multivariate shifting conserves energy)**.**

$$\Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\tau)) \mid a) = \Phi((V, y \mapsto v_1, z \mapsto v_2) : (\Gamma, y : \tau, z : L(\tau)) \mid \hat{\lhd}^x_{y,z}(a))$$

$$\begin{aligned} &\Phi((V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3)) : (\Gamma, t : T(\tau)) \mid a) \\ = \;&\Phi((V, x \mapsto v_1 y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \hat{\lhd}^t_{x,y,z}(a)) \end{aligned}$$

Proof. This statement is proven directly for lists, and via lists for trees.

lists If the nonterminal $\bullet$ is used in V, v_1, or v_2, then this equality holds with both sides ∞. Otherwise, let $p_{V'} = \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w))$. Then the following equalities hold:

$$\begin{aligned} &\Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\tau)) \mid a) \\ = &\sum_{i \in MInd(\Gamma, x : L(\tau))} a(i) \cdot \phi_{i(x)}(v_1 :: v_2) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{i(w)}(V(w)) && \textit{def} \\ = &\sum_{\substack{V', k\, s.t. \\ (V', x \mapsto k) \in \\ MInd(\Gamma, x : L(\tau))}} a(V', x \mapsto k) \cdot \phi_k(v_1 :: v_2) \cdot p_{V'} && \textit{def} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{\substack{V' \, s.t. \\ (V', x \mapsto [\,]) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto [\,]) \cdot \phi_{[\,]}(v_1 :: v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_{i :: j}(v_1 :: v_2) \cdot p_{V'} && \textit{disjoint domains} \\
&= \sum_{\substack{V' \, s.t. \\ (V', x \mapsto [\,]) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto [\,]) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \left(\phi_{i :: j}(v_2) + \phi_i(v_1) \cdot \phi_j(v_2) + \sum_{k \in i :: j} \phi_k(v_1) \cdot \phi_{i :: j}(v_2) \right) \cdot p_{V'} && \textit{def} \\
&= \sum_{\substack{V' \, s.t. \\ (V', x \mapsto [\,]) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto [\,]) \cdot p_{V'} + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_{i :: j}(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_i(v_1) \cdot \phi_j(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \sum_{k \in i :: j} \phi_k(v_1) \cdot \phi_{i :: j}(v_2) \cdot p_{V'} && \textit{algebra} \\
&= \sum_{\substack{V' \, s.t. \\ (V', x \mapsto [\,]) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto [\,]) \cdot \sum_{i \in \mathtt{C}(\tau)} \phi_i(v_1) \cdot \phi_{[\,]}(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \sum_{k \in \mathtt{C}(\tau)} \phi_k(v_1) \cdot \phi_{i :: j}(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_i(v_1) \cdot \phi_j(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \sum_{k \in i :: j} \phi_k(v_1) \cdot \phi_{i :: j}(v_2) \cdot p_{V'} && \textit{Lemma 7.5.2} \\
&= \sum_{\substack{V', j \, s.t. \\ (V', x \mapsto j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto j) \cdot \sum_{i \in \mathtt{C}(\tau)} \phi_i(v_1) \cdot \phi_j(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_i(v_1) \cdot \phi_j(v_2) \cdot p_{V'} \\
&\quad + \sum_{\substack{V', i, j \, s.t. \\ (V', x \mapsto i :: j) \in \\ MInd(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \sum_{k \in i :: j} \phi_k(v_1) \cdot \phi_{i :: j}(v_2) \cdot p_{V'} && \textit{disjoint domains}
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{\substack{V',j\ s.t.\\ (V',x\mapsto j)\in\\ \mathit{MInd}(\Gamma,x:L(\tau))}} \sum_{i\in\mathtt{C}(\tau)} a(V',x\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',i,j\ s.t.\\ (V',x\mapsto i::j)\in\\ \mathit{MInd}(\Gamma,x:L(\tau))}} a(V',x\mapsto i::j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',i,j\ s.t.\\ (V',x\mapsto i::j)\in\\ \mathit{MInd}(\Gamma,x:L(\tau))}} \sum_{k\in\mathit{MInd}(\tau)} \#(k,i::j)\cdot a(V',x\mapsto i::j)\cdot\phi_i(v_1)\cdot\phi_{i::j}(v_2)\cdot p_{V'} && \mathit{algebra}\\
&= \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))\\ \wedge\, i\in\mathtt{C}(\tau)}} a(V',x\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))}} a(V',x\mapsto i::j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',k,j\ s.t.\\ (V',y\mapsto k,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))\\ \wedge\, j\neq[\,]}} \#(k,j)\cdot a(V',x\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'} && \mathit{substitution}\\
&= \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))\\ \wedge\, i\in\mathtt{C}(\tau)}} a(V',x\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))}} a(V',x\mapsto i::j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'}\\
&\quad+ \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))}} \#(i,j)\cdot a(V',x\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'} && \#(i,[\,])=0\\
&= \sum_{\substack{V',i,j\ s.t.\\ (V',y\mapsto i,z\mapsto j)\in\\ \mathit{MInd}(\Gamma,y:\tau,z:L(\tau))}} \hat{\lhd}^{x}_{y,z}(a)(V',y\mapsto i,z\mapsto j)\cdot\phi_i(v_1)\cdot\phi_j(v_2)\cdot p_{V'} && \mathit{distribution},\ \mathit{def}\\
&= \sum_{i\in\mathit{MInd}(\Gamma,y:\tau,z:L(\tau))} \hat{\lhd}^{x}_{y,z}(a)(i)\cdot\phi_{i(y)}(v_1)\cdot\phi_{i(z)}(v_2)\cdot\prod_{w\in\mathtt{dom}(V)}\phi_{i(w)}(V(w)) && \mathit{def}\\
&= \Phi((V,y\mapsto v_1,z\mapsto v_2):(\Gamma,y:\tau,z:L(\tau))\mid\hat{\lhd}^{x}_{y,z}(a)) && \mathit{def}
\end{aligned}
$$

$$\textsc{M-Sub}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a} \geq \vec{a'} \qquad \vec{b} \leq \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

Figure 7.5: Multivariate structural typing rule

trees If the nonterminal $\bullet$ is used in V, v_1, v_2, or v_3, then this equality holds with both sides ∞. Otherwise, let $p_{V'} = \prod_{u \in \mathtt{dom}(V)} \phi_{V'(u)}(V(u))$. Then the following equalities hold:

$$\begin{aligned}
&\Phi((V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3)) : (\Gamma, t : T(\tau)) \mid a) && \\
&= \sum_{i \in \mathit{MInd}(\Gamma, t:T(\tau))} a(i) \cdot \phi_{i(t)}(\mathtt{node}(v_1,\, v_2,\, v_3)) \prod_{u \in \mathtt{dom}(V)} \phi_{i(u)}(V(u)) && \textit{def} \\
&= \sum_{i \in \mathit{MInd}(\Gamma, t:L(\tau))} a(i) \cdot \phi_{i(t)}(\mathtt{pre}(\mathtt{node}(v_1,\, v_2,\, v_3))) \prod_{u \in \mathtt{dom}(V)} \phi_{i(u)}(V(u)) && \textit{def} \\
&= \Phi((V, t \mapsto \mathtt{pre}(\mathtt{node}(v_1,\, v_2,\, v_3))) : (\Gamma, t : L(\tau)) \mid a) && \textit{def} \\
&= \Phi((V, t \mapsto v_1 :: \mathtt{pre}(v_1) @ \mathtt{pre}(v_3)) : (\Gamma, t : L(\tau)) \mid a) && \textit{def} \\
&= \Phi((V, y \mapsto v_2, w \mapsto \mathtt{pre}(v_1) @ \mathtt{pre}(v_3)) : (\Gamma, t : L(\tau)) \mid \hat{\lhd}^t_{y,w}(a)) && \textit{list case} \\
&= \sum_{i \in \mathit{MInd}(\Gamma, y:\tau, w:L(\tau))} \hat{\lhd}^t_{y,w}(a)(i) \cdot \phi_{i(w)}(\mathtt{pre}(v_1) @ \mathtt{pre}(v_3)) \prod_{u \in \mathtt{dom}(V)} \phi_{i(u)}(V(u)) && \textit{def} \\
&= \sum_{\substack{V', j, \ell \text{ s.t.} \\ (V', y \mapsto j, w \mapsto \ell) \in \\ \mathit{MInd}(\Gamma, y:\tau, w:L(\tau))}} \hat{\lhd}^t_{y,w}(a)(V', y \mapsto j, w \mapsto \ell) \cdot \phi_\ell(\mathtt{pre}(v_1) @ \mathtt{pre}(v_3)) \cdot p_{V'} && \textit{def} \\
&= \sum_{\substack{V', j, \ell \text{ s.t.} \\ (V', y \mapsto j, w \mapsto \ell) \in \\ \mathit{MInd}(\Gamma, y:\tau, w:L(\tau))}} \hat{\lhd}^t_{y,w}(a)(V', y \mapsto j, w \mapsto \ell) \cdot \left(\sum_{i,k \in \mathtt{pick}_\ell(\mathtt{cut}(|\ell|))} \phi_i(\mathtt{pre}(v_1)) \cdot \phi_k(\mathtt{pre}(v_3)) \right) \cdot p_{V'} && \textit{Lemma 7.7.1} \\
&= \sum_{\substack{V', i, j, k \text{ s.t.} \\ (V', x \mapsto i, y \mapsto j, z \mapsto k) \in \\ \mathit{MInd}(\Gamma, x:L(\tau), y:\tau, z:L(\tau))}} \sum_{\substack{\ell \text{ s.t.} \\ \langle i, k \rangle \mathtt{pick}_\ell(\mathtt{cut}(|\ell|))}} \hat{\lhd}^t_{y,w}(a)(V', y \mapsto j, w \mapsto \ell) \cdot \phi_i(\mathtt{pre}(v_1)) \cdot \phi_k(\mathtt{pre}(v_3)) \cdot p_{V'} && \textit{distribution} \\
&= \sum_{\substack{V', i, j, k \text{ s.t.} \\ (V', x \mapsto i, y \mapsto j, z \mapsto k) \in \\ \mathit{MInd}(\Gamma, x:L(\tau), y:\tau, z:L(\tau))}} \hat{\lhd}^t_{x,y,z}(a)(V', x \mapsto i, y \mapsto j, z \mapsto k) \cdot \phi_i(\mathtt{pre}(v_1)) \cdot \phi_k(\mathtt{pre}(v_3)) \cdot p_{V'} && \textit{def} \\
&= \sum_{i \in \mathit{MInd}(\Gamma, x:L(\tau), y:\tau, z:L(\tau))} \hat{\lhd}^t_{x,y,z}(a)(i) \cdot \phi_{i(x)}(\mathtt{pre}(v_1)) \cdot \phi_{i(z)}(\mathtt{pre}(v_3)) \cdot \prod_{u \in \mathtt{dom}(V)} \phi_{i(u)}(V(u)) && \textit{def} \\
&= \sum_{i \in \mathit{MInd}(\Gamma, x:T(\tau), y:\tau, z:T(\tau))} \hat{\lhd}^t_{x,y,z}(a)(i) \cdot \phi_{i(x)}(v_1) \cdot \phi_{i(z)}(v_3) \cdot \prod_{u \in \mathtt{dom}(V)} \phi_{i(u)}(V(u)) && \textit{def} \\
&= \Phi((V, x \mapsto v_1 y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \hat{\lhd}^t_{x,y,z}(a)) && \textit{def}
\end{aligned}$$

□

M-VAR
$$\frac{}{\Gamma, x : \tau \mid \hat{\Upsilon}_x^{x,\mathtt{ret}}(\vec{a}) \vdash x : \tau \mid \vec{a}}$$

M-LET
$$\frac{\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c} \qquad \Gamma, x : \sigma \mid [x/\mathtt{ret}]\vec{c} \vdash e_2 : \tau \mid \mathtt{ext}_{x:\sigma}\vec{b}}{\Gamma \mid \vec{a} \vdash \mathtt{let}\ x \ = \ e_1\ \mathtt{in}\ e_2 : \tau \mid \vec{b}}$$

M-FUN
$$\frac{\Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \mathtt{ext}_{\Gamma, f:\tau \overset{\vec{b}|\vec{c}}{\to} \sigma}([x/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid \mathtt{ext}_{\Gamma, f:\tau \overset{\vec{b}|\vec{c}}{\to} \sigma}([x/\mathtt{arg}]\vec{c})}{\Gamma \mid \vec{a} \vdash \mathtt{fun}\ f\ x \ = \ e : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \mathtt{ext}_{\mathtt{ret}:\tau \overset{\vec{b}|\vec{c}}{\to} \sigma}\vec{a}}$$

M-APP
$$\frac{\vec{a} \geq 0}{\Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \vec{a} + \mathtt{ext}_{\Gamma, f:\tau \overset{\vec{b}|\vec{c}}{\to} \sigma}([x/\mathtt{arg}]\vec{b}) \vdash f\ x : \sigma \mid \mathtt{ext}_{\mathtt{ret}:\sigma}(\vec{a}) + \mathtt{ext}_{\Gamma, f:\tau \overset{\vec{b}|\vec{c}}{\to} \sigma}([x/\mathtt{arg}]\vec{c})}$$

M-TICK
$$\frac{\forall i \in \mathtt{C}(\Gamma).\, \vec{a}_i = \vec{b}_{i,\mathtt{ret}\mapsto\langle\rangle} + r \qquad \forall i \notin \mathtt{C}(\Gamma).\, \vec{a}_i = \vec{b}_{i,\mathtt{ret}\mapsto\langle\rangle}}{\Gamma \mid \vec{a} \vdash \mathtt{tick}\{r\} : \mathbb{1} \mid \vec{b}}$$

M-PAIR
$$\frac{}{\Gamma, x : \tau, y : \sigma \mid \hat{\Upsilon}_x^{x,x'}(\hat{\Upsilon}_y^{y,y'}(\mathtt{unp}_{x',y'}^{\mathtt{ret}}(\vec{a}))) \vdash \langle x,\, y\rangle : \tau \otimes \sigma \mid \vec{a}}$$

M-CASEP
$$\frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \mathtt{unp}_{y,z}^{x'}(\vec{a}) \vdash e : \tau \mid \mathtt{unp}_{y,z}^{x'}(\vec{b})}{\Gamma, x : \sigma \otimes \rho \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\, z\rangle \to e : \tau \mid \hat{\Upsilon}_x^{x,x'}(\vec{b})}$$

M-SUML
$$\frac{}{\Gamma, x : \tau \mid \hat{\Upsilon}_x^{x,x'}(\mathtt{unl}_{x'}^{\mathtt{ret}}(\vec{a})) \vdash \mathtt{l}(x) : \tau \oplus \sigma \mid \vec{a}}$$

M-SUMR
$$\frac{}{\Gamma, x : \sigma \mid \hat{\Upsilon}_x^{x,x'}(\mathtt{unr}_{x'}^{\mathtt{ret}}(\vec{a})) \vdash \mathtt{r}(x) : \tau \oplus \sigma \mid \vec{a}}$$

M-CASES
$$\frac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \mathtt{unl}_y^{x'}(\vec{a}) \vdash e_1 : \tau \mid \mathtt{unl}_y^{x'}(\vec{b}) \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid \mathtt{unr}_z^{x'}(\vec{a}) \vdash e_2 : \tau \mid \mathtt{unr}_z^{x'}(\vec{b})\end{array}}{\Gamma, x : \sigma \oplus \rho \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid \hat{\Upsilon}_x^{x,x'}(\vec{b})}$$

Figure 7.6: Multivariate typing rules 1

$$\textsc{M-Nil}\quad \frac{}{\Gamma \mid \mathsf{unn}^{\mathtt{ret}}(\vec{a}) \vdash [\,] : L(\tau) \mid \vec{a}}$$

$$\textsc{M-Cons}\quad \frac{}{\Gamma, x:\tau, y:L(\tau) \mid \hat{\curlyvee}^{x,x'}_{x}(\hat{\curlyvee}^{y,y'}_{y}(\hat{\lhd}^{\mathtt{ret}}_{x',y'}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

$$\textsc{M-CaseL}\quad \frac{\begin{array}{c}\mathsf{unn}^{x}(\mathsf{unn}^{x'}(\vec{a})) = \mathsf{unn}^{x}(\vec{c}) \qquad \mathsf{unn}^{x}(\mathsf{unn}^{x'}(\vec{b})) = \mathsf{unn}^{x}(\vec{d}) \\ \Gamma, x:L(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \qquad \Gamma, x:L(\sigma), y:\sigma, z:L(\sigma) \mid \hat{\lhd}^{x'}_{y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}^{x'}_{y,z}(\vec{b})\end{array}}{\Gamma, x:L(\sigma) \mid \hat{\curlyvee}^{x,x'}_{x}(\vec{a}) \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \hat{\curlyvee}^{x,x'}_{x}(\vec{b})}$$

$$\textsc{M-Leaf}\quad \frac{}{\Gamma \mid \mathsf{unn}^{\mathtt{ret}}(\vec{a}) \vdash \mathtt{leaf} : T(\tau) \mid \vec{a}}$$

$$\textsc{M-Node}\quad \frac{}{\Gamma, x:T(\tau), y:\tau, z:T(\tau) \mid \hat{\curlyvee}^{x,x'}_{x}(\hat{\curlyvee}^{y,y'}_{y}(\hat{\curlyvee}^{z,z'}_{z}(\hat{\lhd}^{\mathtt{ret}}_{x',y',z'}(\vec{a})))) \vdash \mathtt{node}(x,\ y,\ z) : T(\tau) \mid \vec{a}}$$

$$\textsc{M-CaseT}\quad \frac{\begin{array}{c}\mathsf{unn}^{x}(\mathsf{unn}^{x'}(\vec{a})) = \mathsf{unn}^{x}(\vec{c}) \qquad \mathsf{unn}^{x}(\mathsf{unn}^{x'}(\vec{b})) = \mathsf{unn}^{x}(\vec{d}) \qquad \Gamma, t:T(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \\ \Gamma, t:T(\sigma), x:T(\sigma), y:\sigma, z:T(\sigma) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{b})\end{array}}{\Gamma, t:T(\sigma) \mid \hat{\curlyvee}^{t,t'}_{t}(\vec{a}) \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{leaf} \to e_1 \mid \mathtt{node}(x,\ y,\ z) \to e_2 : \tau \mid \hat{\curlyvee}^{t,t'}_{t}(\vec{b})}$$

Figure 7.7: Multivariate typing rules 2

7.8 Typing Rules

The exponential multivariate typing rules are given across Figures 7.5 to 7.7. These new typing rules make use of the essentially the same typing judgment as in the previous chapters, except now it uses multivariate annotations. The typing judgment is:

$$\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$$

This typing judgment means that, with an initial type context Γ annotated by $\vec{a} \geq 0$, the expression e is typed τ and the remainder context $(\Gamma, \texttt{ret} : \tau)$ is annotated by $\vec{b} \geq 0$. To be explicit, $\vec{a}$ is indexed by $MInd(\Gamma)$ and $\vec{b}$ is indexed by $MInd(\Gamma, \texttt{ret} : \tau)$.

To write down the typing rules, other chapters use notation like a, b where a and b annotated disjoint contexts. However, this notation is not appropriate here because every multivariate annotation involves the entire context, so annotations just from the context a are not compatible with combined context of a *and* b. Instead, it is necessary to introduce different notation for how properly manipulate contexts. I explain these notational differences in the following definitions, starting with Definition 7.8.1.

Definition 7.8.1 (annotation extension)**.** *The operation* $\texttt{ext}$ *is used to extend annotation maps over new variables. Given an annotation map a for a type context Γ,* $\texttt{ext}_x$ *adds annotations for a variable x of type τ as follows:*

$$\texttt{ext}_{x:\tau}(a)(V, x \mapsto i) = \begin{cases} a(V) & i \in \texttt{C}(\tau) \\ 0 & otherwise \end{cases}$$

This operation may be used to extend contexts with multiple variables at once by replacing $x : \tau$ with a type context.

Lemma 7.8.1 (extension conserves energy)**.** *Context extension perfectly conserves energy. That is, for all $v : \tau$,*

$$\Phi(V : \Gamma \mid a) = \Phi(V, x \mapsto v : \Gamma, x : \tau \mid \texttt{ext}_{x:\tau}(a))$$

Proof. This identity is proven directly.

$$
\begin{aligned}
\Phi(V : \Gamma \mid a) &= \sum_{i \in \mathit{MInd}(\Gamma)} a(i) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) && \mathit{def} \\
&= \sum_{j \in \mathtt{C}(\tau)} \phi_j(v) \cdot \sum_{i \in \mathit{MInd}(\Gamma)} a(i) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) && \mathit{Lemma}\ 7.5.2 \\
&= \sum_{\substack{i \in \mathit{MInd}(\Gamma) \\ j \in \mathtt{C}(\tau)}} a(i) \cdot \phi_j(v) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) && \mathit{distribution} \\
&= \sum_{\substack{i \in \mathit{MInd}(\Gamma) \\ j \in \mathtt{C}(\tau)}} a(i) \cdot \phi_j(v) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) \\
&\quad + \sum_{\substack{i \in \mathit{MInd}(\Gamma) \\ j \notin \mathtt{C}(\tau)}} 0 \cdot \phi_j(v) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) && \mathit{algebra} \\
&= \sum_{\substack{i \in \mathit{MInd}(\Gamma) \\ j \in \mathit{MInd}(\tau)}} \mathtt{ext}_{x:\tau}(a)(i, x \mapsto j) \cdot \phi_j(v) \cdot \prod_{y \in \mathtt{dom}(V)} \phi_{i(y)}(V(i)) && \mathit{def} \\
&= \sum_{i \in \mathit{MInd}(\Gamma, x:\tau)} \mathtt{ext}_{x:\tau}(a)(i) \prod_{y \in \mathtt{dom}(V, x \mapsto v)} \phi_{i(y)}(V(i)) && \mathit{def} \\
&= \Phi(V, x \mapsto v : \Gamma, x : \tau \mid \mathtt{ext}_{x:\tau}(a)) && \mathit{def}
\end{aligned}
$$

□

Aside from context extension, there are just a few additional annotation-manipulating operations defined across Definitions 7.8.2 to 7.8.4. These operations are all designed to adjust the domain of annotation maps to properly account for the contruction of pairs, variants, and empty data structures, respectively.

Definition 7.8.2 (annotation unpairing)**.** *The operation* unp *is used to ungroup the parts of annotations associated with a pair. Given a pair $x = \langle y,\, z\rangle$, the operation creates an appropriate annotation for its parts as follows:*

$$\mathtt{unp}^{x}_{y,z}(a)(V, y \mapsto i, z \mapsto j) = a(V, x \mapsto \langle i,\, j\rangle)$$

Definition 7.8.3 (annotation untagging)**.** *The operations* $\mathtt{unl}, \mathtt{unr}$ *are used to remove the tag from the parts of annotations associated with a variant. Given a variant labelled* x *these operations create appropriate annotations for the variants* $\mathtt{l}(y) = x$ *and* $\mathtt{r}(z) = x$ *as follows:*

$$\mathtt{unl}^x_y(a)(V, y \mapsto i) = a(V, x \mapsto \mathtt{l}(i))$$

$$\mathtt{unr}^x_z(a)(V, z \mapsto i) = a(V, x \mapsto \mathtt{r}(i))$$

Definition 7.8.4 (annotation un-nil-ing)**.** *The operation* unn *is used to ignore higher-order annotations for empty list and trees. Given an annotation map* a *for a type context* $(\Gamma, x : \tau)$*, where* $\tau = L(\sigma)$ *or* $\tau = T(\sigma)$*,* $\mathtt{unn}^x$ *accomplishes this task as follows for the list* $x = [\,]$ *or tree* $x = \mathtt{leaf}$*:*

$$\mathtt{unn}^x(a)(V) = a(V, x \mapsto [\,])$$

Lemma 7.8.2 ($\mathtt{unp}, \mathtt{unl}, \mathtt{unr}, \mathtt{unn}$ conserve energy)**.** *Each of the operations* $\mathtt{unp}, \mathtt{unl}, \mathtt{unr}, \mathtt{unn}$ *perfectly conserve energy for their intended values.*

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \tau \otimes \sigma) \mid a) = \Phi((V, y \mapsto v_1, z \mapsto v_2) : (\Gamma, y : \tau, z : \sigma) \mid \mathtt{unp}^x_{y,z}(a))$$

$$\Phi((V, x \mapsto \mathtt{l}(v)) : (\Gamma, x : \tau \oplus \sigma) \mid a) = \Phi((V, y \mapsto v) : (\Gamma, y : \tau) \mid \mathtt{unl}^x_y(a))$$

$$\Phi((V, x \mapsto \mathtt{r}(v)) : (\Gamma, x : \tau \oplus \sigma) \mid a) = \Phi((V, y \mapsto v) : (\Gamma, y : \sigma) \mid \mathtt{unr}^x_y(a))$$

$$\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\tau)) \mid a) = \Phi(V : \Gamma \mid \mathtt{unn}^x(a))$$

$$\Phi((V, x \mapsto \mathtt{leaf}) : (\Gamma, x : T(\tau)) \mid a) = \Phi(V : \Gamma \mid \mathtt{unn}^x(a))$$

Proof. Each of these identities is proven directly.

pair `unp`

$$
\begin{aligned}
& \Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \tau \otimes \sigma) \mid a) \\
= & \sum_{\substack{(V', x \mapsto \langle i,\, j\rangle) \\ \in \mathit{MInd}(\Gamma, x:\tau\otimes\sigma)}} a(V', x \mapsto \langle i,\, j\rangle) \cdot \phi_{\langle i,\, j\rangle}(\langle v_1,\, v_2\rangle) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
= & \sum_{\substack{(V', y \mapsto i, z \mapsto j) \\ \in \mathit{MInd}(\Gamma, y:\tau, z:\sigma)}} \mathtt{unp}^x_{y,z} a(V', y \mapsto i, z \mapsto j) \cdot \phi_i(v_1) \cdot \phi_j(v_2) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
= & \, \Phi((V, y \mapsto v_1, z \mapsto v_2) : (\Gamma, y : \tau, z : \sigma) \mid \mathtt{unp}^x_{y,z}(a)) && \textit{def}
\end{aligned}
$$

variant `unl`

$$
\begin{aligned}
& \Phi((V, x \mapsto \mathtt{l}(v)) : (\Gamma, x : \tau \oplus \sigma) \mid a) \\
= & \sum_{\substack{(V', x \mapsto i) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto i) \cdot \phi_i(\mathtt{l}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
= & \sum_{\substack{(V', x \mapsto \mathtt{l}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{l}(i)) \cdot \phi_{\mathtt{l}(i)}(\mathtt{l}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
& + \sum_{\substack{(V', x \mapsto \mathtt{r}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{r}(i)) \cdot \phi_{\mathtt{r}(i)}(\mathtt{l}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{disjoint domains} \\
= & \sum_{\substack{(V', y \mapsto i) \\ \in \mathit{MInd}(\Gamma, y:\tau)}} \mathtt{unl}^x_y(a)(V', y \mapsto i) \cdot \phi_i(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
& + \sum_{\substack{(V', x \mapsto \mathtt{r}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{r}(i)) \cdot 0 \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
= & \sum_{\substack{(V', y \mapsto i) \\ \in \mathit{MInd}(\Gamma, y:\tau)}} \mathtt{unl}^x_y(a)(V', y \mapsto i) \cdot \phi_i(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{algebra} \\
= & \, \Phi((V, y \mapsto v) : (\Gamma, y : \tau) \mid \mathtt{unl}^x_y(a)) && \textit{def}
\end{aligned}
$$

variant unr

$$
\begin{aligned}
&\Phi((V, x \mapsto \mathtt{r}(v)) : (\Gamma, x : \tau \oplus \sigma) \mid a) \\
&= \sum_{\substack{(V', x \mapsto i) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto i) \cdot \phi_i(\mathtt{r}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{\substack{(V', x \mapsto \mathtt{l}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{l}(i)) \cdot \phi_{\mathtt{l}(i)}(\mathtt{r}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto \mathtt{r}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{r}(i)) \cdot \phi_{\mathtt{r}(i)}(\mathtt{r}(v)) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{disjoint domains} \\
&= \sum_{\substack{(V', y \mapsto i) \\ \in \mathit{MInd}(\Gamma, y:\sigma)}} \mathtt{unr}(a)(V', y \mapsto i) \cdot \phi_i(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto \mathtt{r}(i)) \\ \in \mathit{MInd}(\Gamma, x:\tau\oplus\sigma)}} a(V', x \mapsto \mathtt{r}(i)) \cdot 0 \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{\substack{(V', y \mapsto i) \\ \in \mathit{MInd}(\Gamma, y:\sigma)}} \mathtt{unr}(a)(V', y \mapsto i) \cdot \phi_i(v) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{algebra} \\
&= \Phi((V, y \mapsto v) : (\Gamma, y : \sigma) \mid \mathtt{unr}^x_y(a)) && \textit{def}
\end{aligned}
$$

empty list unn

$$
\begin{aligned}
&\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\tau)) \mid a) \\
&= \sum_{\substack{(V', x \mapsto i) \\ \in \mathit{MInd}(\Gamma, x:L(\tau))}} a(V', x \mapsto i) \cdot \phi_i([\,]) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{\substack{(V', x \mapsto [\,]) \\ \in \mathit{MInd}(\Gamma, x:L(\tau))}} a(V', x \mapsto [\,]) \cdot \phi_{[\,]}([\,]) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto i :: j) \\ \in \mathit{MInd}(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_{i :: j}([\,]) \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{disjoint domains} \\
&= \sum_{V' \in \mathit{MInd}(\Gamma)} \mathtt{unn}^x(a)(V') \cdot 1 \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto i :: j) \\ \in \mathit{MInd}(\Gamma, x:L(\tau))}} a(V', x \mapsto i :: j) \cdot 0 \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{V' \in \mathit{MInd}(\Gamma)} \mathtt{unn}^x(a)(V') \cdot \prod_{w \in \mathtt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{algebra} \\
&= \Phi(V : \Gamma \mid \mathtt{unn}^x(a)) && \textit{def}
\end{aligned}
$$

leaf unn

$$
\begin{aligned}
&\Phi((V, x \mapsto \texttt{leaf}) : (\Gamma, x : T(\tau)) \mid a) \\
&= \sum_{\substack{(V', x \mapsto i) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto i) \cdot \phi_i(\texttt{leaf}) \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{\substack{(V', x \mapsto [\,]) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto [\,]) \cdot \phi_{[\,]}(\texttt{leaf}) \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto i :: j) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_{i :: j}(\texttt{leaf}) \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{disjoint domains} \\
&= \sum_{\substack{(V', x \mapsto [\,]) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto [\,]) \cdot \phi_{[\,]}([\,]) \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto i :: j) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto i :: j) \cdot \phi_{i :: j}([\,]) \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{V' \in \mathit{MInd}(\Gamma)} \texttt{unn}^x(a)(V') \cdot 1 \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) \\
&\quad + \sum_{\substack{(V', x \mapsto i :: j) \\ \in \mathit{MInd}(\Gamma, x:T(\tau))}} a(V', x \mapsto i :: j) \cdot 0 \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{def} \\
&= \sum_{V' \in \mathit{MInd}(\Gamma)} \texttt{unn}^x(a)(V') \cdot \prod_{w \in \texttt{dom}(V)} \phi_{V'(w)}(V(w)) && \textit{algebra} \\
&= \Phi(V : \Gamma \mid \texttt{unn}^x(a)) && \textit{def}
\end{aligned}
$$

□

Aside from the aforementioned notation and operations, the meaning of each of the typing rules is essentially no different than in other chapters. However, there are some details of the rule *M-App* worth discussing, which I explain in the next paragraphs.

The subtlety of the function application rule *M-App* is that the multivariate nature of the annotation indices prevents one from directly working with only the annotations of the function argument—annotations apply to whole contexts. For this reason, the only annotations to get transformed according to the function type are those where the context $(\Gamma, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$ is assigned its indices for the constant resource function. Otherwise, the rule is relatively straightforward; it uses pointwise addition to separate out annotations that do not get transformed, as well as the annotation extension operation defined in Definition 7.8.1 to properly handle annotation map domains.

Now, in principle, one could do better than the rule *M-App* through using *cost-free* types. In fact, every chapter's function-application rule could, but there is even more utility to be gained in the multivariate setting, so I take the time to explain that extra utility here. The multivariate

```
1  fun revApp (a,b) =      (* 1,1,1,0 *)
2    case a of
3    | [] -> b             (* 0,0,1,0 *)
4    | x::xs ->            (* 2,2,2,1 *)
5      let tmp = x::b in   (* 1,1,1,0 *)
6      revApp (xs, tmp)    (* 0,0,1,0 *)
```

Figure 7.8: Code for `revApp` with energy comments

AARA literature uses cost-free types to handle annotation transformations for other context indices [65, 80]. Such an approach considers each way that the annotations of the argument could extend to the rest of the context, and then each of these ways are individually transformed according to the cost-free type. This technique allows all resource functions that partially depend on the function's argument to change their dependence to the function's return, which enables more kinds of cost-bound relations. The operation of this approach is similar to how, if $p \cdot x \geq q \cdot x$ for $x \geq 0$, then $r + p \cdot x + p \cdot y \geq r + q \cdot x + q \cdot y$, where r plays the role of $\vec{a}$ and $y \geq 0$ plays the role of a resource function's contribution from the rest of the context aside from the argument. The reason cost-free types are needed is because costful types only give inequalities like $p \cdot x \geq q \cdot x + s$, and this extra difference of s (which may be positive or negative) prevents the same reasoning. However, this thesis purposely has confined cost-free types to Chapter 8, so such optimizations are not present in *M-App*.

Example 7.8.1. The typing rules in this section can be used to type the tail-recursive `revApp` from Figure 7.1 using exponential energy. This code does not exhibit any cost,[5] so the ideal AARA type should reallocate all the energy on the input lists a, b to the output. Such a type is key to allowing the AARA analysis to find tight cost bounds when composing with `revApp`.

Using the multivariate exponential system, `revApp` can be typed as $L(\mathbb{1}) \xrightarrow{\vec{c}|\vec{d}} L(\mathbb{1})$ where $\vec{c}$ assigns an annotation of 1 for each of the following argument indices: $\langle[\langle\rangle], [\,]\rangle$, $\langle[\,], [\langle\rangle]\rangle$, and $\langle[\langle\rangle], [\langle\rangle]\rangle$. All other indices in $\vec{c}$ can be given an annotation of 0. The indices assigned 1 correspond to the resource functions $\left\{{|a|+1 \atop 2}\right\}$, $\left\{{|b|+1 \atop 2}\right\}$, and $\left\{{|a|+1 \atop 2}\right\} \cdot \left\{{|b|+1 \atop 2}\right\}$, respectively. Then the return annotation $\vec{d}$ can be given an annotation of 1 for the index $[\langle\rangle]$, corresponding to $\left\{{|\texttt{ret}|+1 \atop 2}\right\}$, and all other indices can be given an annotation of 0. This return annotation for `revApp` is the correct one desired for use in the typing of `rev`, as it successfully assigns base-2 energy to the returned list. Indeed, $\left\{{|a|+1 \atop 2}\right\} + \left\{{|b|+1 \atop 2}\right\} + \left\{{|a|+1 \atop 2}\right\} \cdot \left\{{|b|+1 \atop 2}\right\} = \left\{{|a|+|b|+1 \atop 2}\right\}$.

The thrust of the type derivation for `revApp` can be witnessed by the energy comments given in Figure 7.8. Notationally, I write w, x, y, z to indicate the annotations for indices corresponding to the resource functions $\left\{{|a|+1 \atop 2}\right\} \cdot \left\{{|b|+1 \atop 2}\right\}$, $\left\{{|a|+1 \atop 2}\right\}$, $\left\{{|b|+1 \atop 2}\right\}$, and the constant 1, respectively. Because there is no energy in the remainder, I elide any comments concerning the remainder.

Some key lines to view in Figure 7.8 include lines 4 and 5 where shifting and unshifting occur. Additionally, lines 3 and 6 are where the amount of $\left\{{|b|+1 \atop 2}\right\}$ energy is given for the return. There would be some reindexing in these lines to remove dependence on other indices, but I do not reflect that in the comments aside from weakening other annotations to 0. This weakening

[5]In principle one could add ticks to apply costs to `revApp`, but I do not do so to keep the example simple.

does not actually lose any energy because the list `a` is empty at the point of return, so each resource function depending on `a` is 0.

7.9 Soundness

The soundness of AARA with multivariate exponential resource functions is similar to Theorems 5.4.1 and 6.4.3 in that the initial potential energy of a context bounds the peak cost of evaluation, and the difference between initial and final energies bounds the net cost. However, some additional care must be taken to work the multivariate annotations, as opposed to the previous univariate annotations. This additional care results in Theorem 7.9.1.

Theorem 7.9.1 (exponential multivariate soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p$ *(diff. bounds net)*

Proof. The soundness proof proceeds by lexicographic induction over the derivation of the evaluation judgment followed by the typing judgment.

Much of this proof is closely analogous to that for Theorem 6.4.3. However, this proof also begins with a special case for the nonterminal dummy value $\bullet$ to match the special case of the definition for potential energy.

Now each case in the induction is given in more detail:

nonterminal This case deals with the cost bounds where the nonterminal dummy value $\bullet$ is present in V so that other cases need not consider the dummy value in V. However, this case does not cover the derivation of $v : \tau$, which can still be handled by the other cases.

Suppose that the nonterminal dummy value $\bullet$ is present in V. Then $\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) = \infty$. Because ∞ is greater than or equal to anything, both the peak and net cost bounds are satisfied.

M-Sub This case deals with the structural typing rule *M-Sub* so that future considerations of typing judgment derivation structure need not consider the case that the derivation ends with the application of *M-Sub*.

Suppose the last rule applied for the typing judgment is *M-Sub*.

$$\text{M-SUB}\quad \frac{\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'} \qquad \vec{a} \geq \vec{a'} \qquad \vec{b} \leq \vec{b'}}{\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}}$$

Then the premisses of this rule hold by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid \vec{a'} \vdash e : \tau \mid \vec{b'}$ to learn:

(1) $v : \tau$
(2) $\Phi(V : \Gamma \mid \vec{a'}) \geq p$
(3) $\Phi(V : \Gamma \mid \vec{a'}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from inequalities (2) and (3) by applying the pointwise monotonicity of potential energy alongside the pointwise annotation inequalities $\vec{a} \geq \vec{a'}$ and $\vec{b} \leq \vec{b'}$.

E-Nont Suppose the last rule applied for the evaluation judgment is *E-Nont*.

$$\textsc{E-Nont}\quad \frac{}{V \vdash e \Downarrow \bullet \mid (0, \infty)}$$

Then $p = 0$, $q = \infty$, and $v = \bullet$. Because $\bullet : \tau$ by *V-Nont*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because ∞ is greater than or equal to anything, the net cost bound also satisfied.

E-Tick Suppose the last rule applied for the evaluation judgment is *E-Tick*.

$$\textsc{E-Tick}\quad \frac{}{V \vdash \texttt{tick}\{r\} \Downarrow \langle\rangle \mid (\max(0, r), \max(0, -r))}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-Tick}\quad \frac{\forall i \in \mathtt{C}(\Gamma).\, \vec{a}_i = \vec{b}_{i,\texttt{ret}\mapsto\langle\rangle} + r \qquad \forall i \notin \mathtt{C}(\Gamma).\, \vec{a}_i = \vec{b}_{i,\texttt{ret}\mapsto\langle\rangle}}{\Gamma \mid \vec{a} \vdash \texttt{tick}\{r\} : \mathbb{1} \mid \vec{b}}$$

The premisses of this rule hold by inversion.

Because $\langle\rangle : \mathbb{1}$ by *V-Unit*, the needed well-formedness judgment holds.

Then the following two inequalities confirm the peak cost bound. Firstly, $\Phi(V : \Gamma \mid \vec{a}) \geq 0$

because $\vec{a} \geq 0$. Then also:

$$\begin{aligned}
\Phi(V : \Gamma \mid \vec{a}) &= \sum_{i \in \mathit{MInd}(\Gamma)} \vec{a}_i \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{def} \\
&\geq \sum_{i \in \mathtt{C}(\Gamma)} \vec{a}_i \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathtt{C}(\Gamma) \subseteq \mathit{MInd}(\Gamma) \\
&= \sum_{i \in \mathtt{C}(\Gamma)} (\vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} + r) \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{def} \\
&= r \cdot \sum_{i \in \mathtt{C}(\Gamma)} \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) \\
&\quad + \sum_{i \in \mathtt{C}(\Gamma)} \vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{distribution} \\
&= r + \sum_{i \in \mathtt{C}(\Gamma)} \vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{Lemma}\ 7.5.2 \\
&\geq r && \vec{b} \geq 0
\end{aligned}$$

Thus, $\Phi(V : \Gamma \mid \vec{a}) \geq \max(0, r)$.

Finally, the following inequalities confirm the net cost bound.

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) + \max(0, -r) \\
&= \max(0, -r) + \sum_{i \in \mathit{MInd}(\Gamma)} \vec{a}_i \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{def} \\
&= \max(0, -r) + \sum_{i \in \mathtt{C}(\Gamma)} \vec{a}_i \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) \\
&\quad + \sum_{i \in \mathit{MInd}(\Gamma) \setminus \mathtt{C}(\Gamma)} \vec{a}_i \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{disjoint\ domains} \\
&= \max(0, -r) + \sum_{i \in \mathtt{C}(\Gamma)} (\vec{b}_{i,\mathtt{ret} \mapsto [\,]} + r) \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) \\
&\quad + \sum_{i \in \mathit{MInd}(\Gamma) \setminus \mathtt{C}(\Gamma)} \vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{premiss} \\
&= \max(0, -r) + r \cdot \sum_{i \in \mathtt{C}(\Gamma)} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) \\
&\quad + \sum_{i \in \mathit{MInd}(\Gamma)} \vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{algebra} \\
&= \max(0, -r) + r + \sum_{i \in \mathit{MInd}(\Gamma)} \vec{b}_{i,\mathtt{ret} \mapsto \langle\rangle} \cdot \prod_{x \in \mathtt{dom}(V)} \phi_{i(x)}(V(x)) && \mathit{Lemma}\ 7.5.2
\end{aligned}$$

$$= \max(0, r) + \sum_{i \in MInd(\Gamma)} \vec{b}_{i,\texttt{ret}\mapsto\langle\rangle} \cdot \prod_{x \in \texttt{dom}(V)} \phi_{i(x)}(V(x)) \qquad \textit{algebra}$$

$$= \max(0, r) + \sum_{(i,\texttt{ret}\mapsto\langle\rangle) \in MInd(\Gamma,\texttt{ret}:\langle\rangle)} \vec{b}_{i,\texttt{ret}\mapsto\langle\rangle} \cdot \phi_{\langle\rangle}(\langle\rangle) \cdot \prod_{x \in \texttt{dom}(V)} \phi_{i(x)}(V(x)) \qquad \textit{def}$$

$$= \max(0, r) + \Phi(V, \texttt{ret} \mapsto \langle\rangle : \Gamma, \texttt{ret} : \langle\rangle \mid \vec{b}) \qquad \textit{def}$$

E-Var Suppose the last rule applied for the evaluation judgment is *E-Var*

$$\frac{\textsc{E-Var}}{V, x \mapsto v \vdash x \Downarrow v \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{M-Var}}{\Gamma, x : \tau \mid \hat{\Upsilon}_x^{x,\texttt{ret}}(\vec{a}) \vdash x : \tau \mid \vec{a}}$$

Then $p = q = 0$ and $(V, x \mapsto v) : (\Gamma, x : \tau)$. Because $v : \tau$ follows from inverting *V-Context*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because sharing perfectly conserves potential (Lemma 7.6.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \hat{\Upsilon}_x^{x,\texttt{ret}}(\vec{a})) = \Phi((V, x \mapsto v, \texttt{ret} \mapsto v) : (\Gamma, x : \tau, \texttt{ret} : \tau) \mid \vec{a})$$

E-Let Suppose the last rule applied for the evaluation judgment is *E-Let*.

$$\textsc{E-Let}\quad \frac{V \vdash e_1 \Downarrow v' \mid (p, q) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)}{V \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v \mid (p + \max(0, r - q), s + \max(0, q - r))}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-Let}\quad \frac{\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c} \qquad \Gamma, x : \sigma \mid [x/\texttt{ret}]\vec{c} \vdash e_2 : \tau \mid \texttt{ext}_{x:\sigma}\vec{b}}{\Gamma \mid \vec{a} \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau \mid \vec{b}}$$

The premisses of both of these rules hold by inversion.

Because $V : \Gamma$ holds by assumption, the inductive hypothesis can be applied with the judgments $V \vdash e_1 \Downarrow v' \mid (p, q)$ and $\Gamma \mid \vec{a} \vdash e_1 : \sigma \mid \vec{c}$. to learn:

(1) $v' : \sigma$

(2) $\Phi(V : \Gamma \mid \vec{a}) \geq p$

(3) $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p$

Because $v' : \sigma$ holds as (1) from the previous induction and both $V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)$ and $\Gamma, x : \sigma \mid [x/\texttt{ret}]\vec{c} \vdash e_2 : \tau \mid \texttt{ext}_{x:\sigma}(\vec{b})$ hold from inversion, the inductive hypothesis can be applied again to learn:

(4) $v : \tau$
(5) $\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) \geq r$
(6) $\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + s$
$\geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma) \mid \texttt{ext}_{x:\sigma}(\vec{b})) + r$

The well-formedness judgment (4) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, proceed by cases on whether $q \geq r$.

If $q \geq r$, then the cost behaviour to consider is $(p, s + (q - r))$. Then (2) confirms the peak cost bound, and the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) + s + (q - r) \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p + s - r && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + p + s - r && \textit{relabelling} \\
&\geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma, \texttt{ret} : \tau) \mid \texttt{ext}_{x:\sigma}(\vec{b})) + p && (6) \\
&= \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p && \textit{Lemma } 7.8.1
\end{aligned}$$

If $q < r$, then the cost behaviour to consider is $(p + (r - q), s)$, and $q \neq \infty$ (so can be subtracted). Then the following inequalities confirm the peak cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + p - q && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + p - q && \textit{relabelling} \\
&\geq p + (r - q) && (5)
\end{aligned}$$

And finally, the following inequalities confirm the net cost bound:

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) + s \\
&\geq \Phi((V, \texttt{ret} \mapsto v') : (\Gamma, \texttt{ret} : \sigma) \mid \vec{c}) + s + p - q && (3) \\
&= \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\texttt{ret}]\vec{c}) + s + p - q && \textit{relabelling} \\
&\geq \Phi((V, x \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma, \texttt{ret} : \tau) \mid \texttt{ext}_{x:\sigma}(\vec{b})) + p + (r - q) && (6) \\
&\geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p + (r - q) && \textit{Lemma } 7.8.1
\end{aligned}$$

E-Fun Suppose the last rule applied for the evaluation judgment is *E-Fun*.

E-FUN
$$\frac{}{V \vdash \texttt{fun}\ f\ x\ =\ e \Downarrow \texttt{C}(V;\ f,\ x.\, e) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

M-FUN
$$\frac{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \texttt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\texttt{arg}]\vec{b}) \vdash e : \sigma \mid \texttt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\texttt{arg}]\vec{c})}{\Gamma \mid \vec{a} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \texttt{ext}_{\texttt{ret}:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}\ \vec{a}}$$

The assumed typing judgment for the expression being evaluated therefore takes the form of this rule's conclusion.

Because $\mathtt{C}(V;\ f,\ x.\,e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma$ follows from *V-Fun* and the assumed typing judgment, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because context extension does not perfectly conserve potential energy (Lemma 7.8.1), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \mathtt{ret} \mapsto \mathtt{C}(V;\ f,\ x.\,e)) : (\Gamma, \mathtt{ret} : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\mathtt{ret}:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}\, \vec{a})$$

E-App Suppose the last rule applied for the evaluation judgment is *E-App*.

$$\textsc{E-App}\quad \frac{V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e) \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e) \vdash f\ x \Downarrow v \mid (p, q)}$$

Then this rule's premiss holds by inversion and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-App}\quad \frac{\vec{a} \geq 0}{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \vec{a} + \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b}) \vdash f\ x : \sigma \mid \mathtt{ext}_{\mathtt{ret}:\sigma}(\vec{a}) + \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{c})}$$

Because $(V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$ by assumption, the rule *V-Context* can be inverted to learn $\mathtt{C}(V';\ g,\ y.\,e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma$. Then further, the rule *V-Fun* can be inverted to learn that this function body can be typed in some context Γ' where $V' : \Gamma'$. Using *V-Context*, one can then use this well-formedness judgment to derive

$$(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$$

Now inspect the derivation of the type of the function closure's body. Only structural rules (like *R-Sub*) and *R-Fun* can conclude a typing derivation for a function, and the application a structural rule itself requires another typing derivation for the same function. Thus it can be shown by induction that the typing derivation must conclude by the rule *M-Fun* followed by some number of uses of structural rules. The typing derivation therefore contains the following rule application:

$$\textsc{M-Fun}\quad \frac{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{c})}{\Gamma \mid \vec{a} \vdash \mathtt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \mathtt{ext}_{\mathtt{ret}:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}\, \vec{a}}$$

This rule's premiss holds by inversion.

Each of the following judgments have now been found:

- $V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (p, q)$
- $(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$
- $\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{c})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \sigma$

(2) $\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{b})) \geq p$

(3)
$$\begin{aligned}&\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{b})) + q \\ &\geq \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{c})) \\ &+p\end{aligned}$$

The well-formedness judgment (1) $v : \sigma$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, first simplify inequalities (2) and (3) into inequalities (4) and (5), respectively, via the following steps:

For inequality (4):

$$\begin{aligned}p &\leq \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{b})) && (2) \\ &= \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) && \textit{Lemma 7.8.1} \\ &= \sum_{(y\mapsto i) \in \mathit{MInd}(y:\tau)} [y/\mathtt{arg}]\vec{b}_{y\mapsto i} \cdot \phi_i(v') && \textit{def} \\ &= \sum_{i \in \mathit{MInd}(\tau)} \vec{b}_{\mathtt{arg}\mapsto i} \cdot \phi_i(v') && \textit{def}\end{aligned}$$

And for inequality (5):

$$\begin{aligned}&\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{b})) + q \\ &\geq \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid \mathtt{ext}_{\Gamma', g:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([y/\mathtt{arg}]\vec{c})) + p && (3) \\ \iff &\Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + q \geq \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid [y/\mathtt{arg}]\vec{c}) + p && \textit{Lemma 7.8.1} \\ \iff &q + \sum_{(y\mapsto i) \in \mathit{MInd}(y:\tau)} [y/\mathtt{arg}]\vec{b}_{y\mapsto i} \cdot \phi_i(v') \geq p + \sum_{(y\mapsto i, \mathtt{ret}\mapsto j) \in \mathit{MInd}(y:\tau, \mathtt{ret}:\sigma)} [y/\mathtt{arg}]\vec{c}_{y\mapsto i, \mathtt{ret}\mapsto j} \cdot \phi_i(v') \cdot \phi_j(v) && \textit{def} \\ \iff &q + \sum_{i \in \mathit{MInd}(\tau)} \vec{b}_{\mathtt{arg}\mapsto i} \cdot \phi_i(v') \geq p + \sum_{i \in \mathit{MInd}(\tau),\, j \in \mathit{MInd}(\sigma)} \vec{c}_{\mathtt{arg}\mapsto i, \mathtt{ret}\mapsto j} \cdot \phi_i(v') \cdot \phi_j(v) && \textit{def}\end{aligned}$$

Now define r to be $\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \vec{a})$, which must be nonnegative because $\vec{a} \geq 0$.

Then the peak cost bound follows via the following inequalities:

$$\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \vec{a} + \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b}))$$

$$\begin{aligned}
&= r + \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b})) && \textit{linearity}\\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid [x/\mathtt{arg}]\vec{b}) && \textit{Lemma 7.8.1}\\
&= r + \sum_{x \mapsto i \in \mathit{MInd}(x:\tau)} [x/\mathtt{arg}]\vec{b}_{x \mapsto i} \cdot \phi_i(v') && \textit{def}\\
&= r + \sum_{i \in \mathit{MInd}(\tau)} \vec{b}_{\mathtt{arg} \mapsto i} \cdot \phi_i(v') && \textit{def}\\
&\geq r + p && (4)\\
&\geq p && r \geq 0
\end{aligned}$$

And finally, the net cost bound follows via the following inequalities:

$$\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \vec{a} + \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b})) + q$$

$$\begin{aligned}
&= r + q + \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{b})) && \textit{linearity}\\
&= r + q + \Phi((x \mapsto v') : (x : \tau) \mid [x/\mathtt{arg}]\vec{b}) && \textit{Lemma 7.8.1}\\
&= r + \sum_{x \mapsto i \in \mathit{MInd}(x:\tau)} [x/\mathtt{arg}]\vec{b}_{x \mapsto i} \cdot \phi_i(v') && \textit{def}\\
&= r + q + \sum_{i \in \mathit{MInd}(\tau)} \vec{b}_{\mathtt{arg} \mapsto i} \cdot \phi_i(v') && \textit{def}\\
&\geq r + p + \sum_{i \in \mathit{MInd}(\tau), j \in \mathit{MInd}(\sigma)} \vec{c}_{\mathtt{arg} \mapsto i, \mathtt{ret} \mapsto j} \cdot \phi_i(v') \cdot \phi_j(v) && (5)\\
&= r + p + \sum_{(x \mapsto i, \mathtt{ret} \mapsto j) \in \mathit{MInd}(x:\tau, \mathtt{ret}:\sigma)} [x/\mathtt{arg}]\vec{c}_{x \mapsto i, \mathtt{ret} \mapsto j} \cdot \phi_i(v') \cdot \phi_j(v) && \textit{def}\\
&= r + p + \Phi((x \mapsto v', \mathtt{ret} \mapsto v) : (x : \tau, y : \sigma) \mid [x/\mathtt{arg}]\vec{c}) && \textit{def}\\
&= r + p + \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, y : \sigma) \mid \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{c})) && \textit{Lemma 7.8.1}\\
&= \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, y : \sigma) \mid \vec{a} + \mathtt{ext}_{\Gamma, f:\tau \xrightarrow{\vec{b}|\vec{c}} \sigma}([x/\mathtt{arg}]\vec{c})) + p && \textit{linearity}
\end{aligned}$$

E-Pair Suppose the last rule applied for the evaluation judgment is *E-Pair*.

$$\frac{\textsc{E-Pair}}{V, x \mapsto v_1, y \mapsto v_2 \vdash \langle x,\ y\rangle \Downarrow \langle v_1,\ v_2\rangle \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{M-Pair}}{\Gamma, x : \tau, y : \sigma \mid \hat{\Upsilon}_x^{x,x'}(\hat{\Upsilon}_y^{y,y'}(\mathtt{unp}_{x',y'}^{\mathtt{ret}}(\vec{a}))) \vdash \langle x,\ y\rangle : \tau \otimes \sigma \mid \vec{a}}$$

Because $\langle v_1,\ v_2\rangle : \tau \otimes \sigma$ follows from *V-Pair* and the assumed well-formedness judgment $(V, x_1 \mapsto v_1, x_2 \mapsto v_2) : (\Gamma, x : \tau, y : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because unp and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), the net cost bound is also satisfied with the following equality:

$$\begin{aligned}&\Phi(V, x \mapsto v_1, y \mapsto v_2 : \Gamma, x : \tau, y : \sigma \mid \hat{\Upsilon}_x^{x,x'}(\hat{\Upsilon}_y^{y,y'}(\mathtt{unp}_{x',y'}^{\mathtt{ret}}(\vec{a}))))\\ &= \Phi(V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto \langle v_1,\, v_2\rangle : \Gamma, x : \tau, y : \sigma, \mathtt{ret} : \tau \otimes \sigma \mid \vec{a})\end{aligned}$$

E-CaseP Suppose the last rule applied for the evaluation judgment is *E-CaseP*.

$$\textsc{E-CaseP}\quad \frac{V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto \langle v_1,\, v_2\rangle \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\, z\rangle \to e \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-CaseP}\quad \frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \mathtt{unp}_{y,z}^{x'}(\vec{a}) \vdash e : \tau \mid \mathtt{unp}_{y,z}^{x'}(\vec{b})}{\Gamma, x : \sigma \otimes \rho \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\, z\rangle \to e : \tau \mid \hat{\Upsilon}_x^{x,x'}(\vec{b})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho)$ by assumption, the rule *V-Context* can be inverted to learn $\langle v_1,\, v_2\rangle : \sigma \otimes \rho$. Then further, the rule *V-Pair* can be inverted to learn both $v_1 : \sigma$ and $v_2 : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)$
- $(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$
- $\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid \mathtt{unp}_{y,z}^{x'}(\vec{a}) \vdash e : \tau \mid \mathtt{unp}_{y,z}^{x'}(\vec{b})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \mathtt{unp}_{y,z}^{x'}(\vec{a})) \geq p$

(3) $\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \mathtt{unp}_{y,z}^{x'}(\vec{a})) + q$
$\geq \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \mathtt{ret} : \tau) \mid \mathtt{unp}_{y,z}^{x'}(\vec{b})) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because unp and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid \mathtt{unp}_{y,z}^{x'}(\vec{a}))$$

$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho) \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}))$$

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid \texttt{unp}_{y,z}^{x'}(\vec{b}))$$
$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, \texttt{ret} : \tau) \mid \hat{\Upsilon}_x^{x,x'}(\vec{b}))$$

E-SumL Suppose the last rule applied for the evaluation judgment is *E-SumL*.

$$\textsc{E-SumL}\quad \frac{}{V, x \mapsto v \vdash \texttt{l}(x) \Downarrow \texttt{l}(v) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-SumL}\quad \frac{}{\Gamma, x : \tau \mid \hat{\Upsilon}_x^{x,x'}(\texttt{unl}_{x'}^{\texttt{ret}}(\vec{a})) \vdash \texttt{l}(x) : \tau \oplus \sigma \mid \vec{a}}$$

Because $\texttt{l}(v) : \tau \oplus \sigma$ follows from *V-SumL* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \tau)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because `unl` and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \hat{\Upsilon}_x^{x,x'}(\texttt{unl}_{x'}^{\texttt{ret}}(\vec{a})))$$
$$= \Phi((V, x \mapsto v, \texttt{ret} \mapsto \texttt{l}(v)) : (\Gamma, x : \tau, \texttt{ret} \mapsto \tau \oplus \sigma) \mid \vec{a})$$

E-SumR Suppose the last rule applied for the evaluation judgment is *E-SumR*.

$$\textsc{E-SumR}\quad \frac{}{V, x \mapsto v \vdash \texttt{r}(x) \Downarrow \texttt{r}(v) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-SumR}\quad \frac{}{\Gamma, x : \sigma \mid \hat{\Upsilon}_x^{x,x'}(\texttt{unr}_{x'}^{\texttt{ret}}(\vec{a})) \vdash \texttt{r}(x) : \tau \oplus \sigma \mid \vec{a}}$$

Because $\texttt{r}(v) : \tau \oplus \sigma$ follows from *V-SumR* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because `unr` and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \sigma) \mid \hat{\Upsilon}_x^{x,x'}(\texttt{unr}_{x'}^{\texttt{ret}}(\vec{a})))$$
$$= \Phi((V, x \mapsto v, \texttt{ret} \mapsto \texttt{r}(v)) : (\Gamma, x : \sigma, \texttt{ret} \mapsto \tau \oplus \sigma) \mid \vec{a})$$

E-CaseS-L Suppose the last rule applied for the evaluation judgment is *E-CaseS-L*.

$$\dfrac{V, x \mapsto \texttt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)}{V, x \mapsto \texttt{l}(v') \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 \Downarrow v \mid (p, q)}\ \textsc{E-CaseS-L}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\dfrac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \texttt{unl}_y^{x'}(\vec{a}) \vdash e_1 : \tau \mid \texttt{unl}_y^{x'}(\vec{b}) \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid \texttt{unr}_z^{x'}(\vec{a}) \vdash e_2 : \tau \mid \texttt{unr}_z^{x'}(\vec{b})\end{array}}{\Gamma, x : \sigma \oplus \rho \mid \hat{\curlyvee}_x^{x,x'}(\vec{a}) \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 : \tau \mid \hat{\curlyvee}_x^{x,x'}(\vec{b})}\ \textsc{M-CaseS}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \texttt{l}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \sigma$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$$

Each of the following judgments have now been found:

- $V, x \mapsto \texttt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$
- $\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \texttt{unl}_y^{x'}(\vec{a}) \vdash e_1 : \tau \mid \texttt{unl}_y^{x'}(\vec{b})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \texttt{unl}_y^{x'}(\vec{a})) \geq p$
(3) $\Phi((V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \texttt{unl}_y^{x'}(\vec{a})) + q$
$\geq \Phi((V, x \mapsto \texttt{l}(v'), y \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \texttt{ret} : \tau) \mid \texttt{unl}_y^{x'}(\vec{b})) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because `unl` and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \texttt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \texttt{unl}_y^{x'}(\vec{a}))$$
$$= \Phi((V, x \mapsto \texttt{l}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \hat{\curlyvee}_x^{x,x'}(\vec{a}))$$

$$\Phi((V, x \mapsto \texttt{l}(v'), y \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \texttt{ret} : \tau) \mid \texttt{unl}_y^{x'}(\vec{b}))$$
$$= \Phi((V, x \mapsto \texttt{l}(v'), \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \texttt{ret} : \tau) \mid \hat{\curlyvee}_x^{x,x'}(\vec{b}))$$

E-CaseS-R Suppose the last rule applied for the evaluation judgment is *E-CaseS-R*.

$$\textsc{E-CaseS-R}\quad \frac{V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_2 \Downarrow v \mid (p, q)}{V, x_s \mapsto \mathtt{r}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-CaseS}\quad \frac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid \mathtt{unl}_y^{x'}(\vec{a}) \vdash e_1 : \tau \mid \mathtt{unl}_y^{x'}(\vec{b}) \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid \mathtt{unr}_z^{x'}(\vec{a}) \vdash e_2 : \tau \mid \mathtt{unr}_z^{x'}(\vec{b})\end{array}}{\Gamma, x : \sigma \oplus \rho \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid \hat{\Upsilon}_x^{x,x'}(\vec{b})}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$
- $\Gamma, x : \sigma \oplus \rho, z : \rho \mid \mathtt{unr}_z^{x'}(\vec{a}) \vdash e_2 : \tau \mid \mathtt{unr}_z^{x'}(\vec{b})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \mathtt{unr}_z^{x'}(\vec{a})) \geq p$
(3) $\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \mathtt{unr}_z^{x'}(\vec{a})) + q$
$\geq \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid \mathtt{unr}_z^{x'}(\vec{b})) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because unr and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \mathtt{unr}_z^{x'}(\vec{a}))$$
$$= \Phi((V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \hat{\Upsilon}_x^{x,x'}(\vec{a}))$$

$$\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid \mathtt{unr}_z^{x'}(\vec{b}))$$
$$= \Phi((V, x \mapsto \mathtt{r}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \mathtt{ret} : \tau) \mid \hat{\Upsilon}_x^{x,x'}(\vec{b}))$$

E-Nil Suppose the last rule applied for the evaluation judgment is *E-Nil*.

$$\frac{\textsc{E-Nil}}{V \vdash [\,] \Downarrow [\,] \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{M-Nil}}{\Gamma \mid \mathtt{unn}^{\mathtt{ret}}(\vec{a}) \vdash [\,] : L(\tau) \mid \vec{a}}$$

Because $[\,] : L(\tau)$ follows from *V-Nil*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because $\mathtt{unn}$ perfectly conserves potential energy (Lemma 7.8.2), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \mathtt{unn}^{\mathtt{ret}}(\vec{a})) = \Phi((V, \mathtt{ret} \mapsto [\,]) : (\Gamma, \mathtt{ret} : L(\tau)) \mid \vec{a})$$

E-Cons Suppose the last rule applied for the evaluation judgment is *E-Cons*.

$$\frac{\textsc{E-Cons}}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{M-Cons}}{\Gamma, x : \tau, y : L(\tau) \mid \hat{\Upsilon}_x^{x,x'}(\hat{\Upsilon}_y^{y,y'}(\hat{\lhd}_{x',y'}^{\mathtt{ret}}(\vec{a}))) \vdash x :: y : L(\tau) \mid \vec{a}}$$

Because $v_1 :: v_2 : L(\tau)$ follows from *V-Cons* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because shifting and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.7.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid \hat{\Upsilon}_x^{x,x'}(\hat{\Upsilon}_y^{y,y'}(\hat{\lhd}_{x',y'}^{\mathtt{ret}}(\vec{a}))))$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto v_1 :: v_2) : (\Gamma, x : \tau, y : L(\tau), \mathtt{ret} : L(\tau)) \mid \vec{a})$$

E-CaseL-Nil Suppose the last rule applied for the evaluation judgment is *E-CaseL-Nil*.

$$\textsc{E-CaseL-Nil}\quad \frac{V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)}{V, x \mapsto [\,] \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\begin{array}{c}\mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{a})) = \mathrm{unn}^x(\vec{c}) \qquad \mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{b})) = \mathrm{unn}^x(\vec{d}) \\ \Gamma, x : L(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \qquad \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \hat{\lhd}^{x'}_{y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}^{x'}_{y,z}(\vec{b})\end{array}}{\Gamma, x : L(\sigma) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{a}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{b})}\ \textsc{M-CaseL}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$
- $\Gamma, x : L(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{c}) \geq p$

(3) $\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{c}) + q$
$\geq \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{d}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because unn and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.8.2), $\mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{a})) = \mathrm{unn}^x(\vec{c})$, and $\mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{b})) = \mathrm{unn}^x(\vec{d})$, both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{c}) = \Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{a}))$$

$$\begin{aligned}&\Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \vec{d}) \\ &= \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{b}))\end{aligned}$$

E-CaseL-Cons Suppose the last rule applied for the evaluation judgment is *E-CaseL-Cons*.

$$\frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)}{V, x \mapsto v_1 :: v_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p, q)}\ \textsc{E-CaseL-Cons}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\begin{array}{c}\mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{a})) = \mathrm{unn}^x(\vec{c}) \qquad \mathrm{unn}^x(\mathrm{unn}^{x'}(\vec{b})) = \mathrm{unn}^x(\vec{d}) \\ \Gamma, x : L(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \qquad \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \hat{\lhd}^{x'}_{y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}^{x'}_{y,z}(\vec{b})\end{array}}{\Gamma, x : L(\sigma) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{a}) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{b})}\ \textsc{M-CaseL}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 : L(\sigma)$. Then further, the rule *V-Cons* can be inverted to learn both $v_1 : \sigma$ and $v_2 : L(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$$

Each of the following judgments has now been found:

- $V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$
- $\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{b})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{a})) \geq p$

(3) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{a})) + q$
$\geq \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{b})) + p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because shifting and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.7.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{a}))$$
$$= \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{a}))$$

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \hat{\triangleleft}^{x'}_{y,z}(\vec{b}))$$
$$= \Phi((V, x \mapsto v_1 :: v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid \hat{\Upsilon}^{x,x'}_{x}(\vec{b}))$$

E-Leaf Suppose the last rule applied for the evaluation judgment is *E-Leaf*.

$$\textsc{E-Leaf}\quad \frac{}{V \vdash \texttt{leaf} \Downarrow \texttt{leaf} \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{M-Leaf}\quad \frac{}{\Gamma \mid \texttt{unn}^{\texttt{ret}}(\vec{a}) \vdash \texttt{leaf} : T(\tau) \mid \vec{a}}$$

Because $\texttt{leaf} : T(\tau)$ follows from *V-Leaf*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative, the peak cost bound is satisfied. And finally, because unn perfectly conserves potential energy (Lemma 7.8.2), the net cost bound is also satisfied with the following equality:

$$\Phi(V : \Gamma \mid \texttt{unn}^{\texttt{ret}}(\vec{a})) = \Phi((V, \texttt{ret} \mapsto \texttt{leaf}) : (\Gamma, \texttt{ret} : T(\tau)) \mid \vec{a})$$

E-Node Suppose the last rule applied for the evaluation judgment is *E-Node*.

E-NODE
$$\frac{}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\, y,\, z) \Downarrow \texttt{node}(v_1,\, v_2,\, v_3) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

M-NODE
$$\frac{}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \hat{\Upsilon}^{x,x'}_{x}(\hat{\Upsilon}^{y,y'}_{y}(\hat{\Upsilon}^{z,z'}_{z}(\hat{\lhd}^{\texttt{ret}}_{x',y',z'}(\vec{a})))) \vdash \texttt{node}(x,\, y,\, z) : T(\tau) \mid \vec{a}}$$

Because $\texttt{node}(v_1,\, v_2,\, v_3) : T(\tau)$ follows from *V-Node* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative, the peak cost bound is satisfied. Finally, because shifting and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.7.2), the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \hat{\Upsilon}^{x,x'}_{x}(\hat{\Upsilon}^{y,y'}_{y}(\hat{\Upsilon}^{z,z'}_{z}(\hat{\lhd}^{\texttt{ret}}_{x',y',z'}(\vec{a})))))$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto \texttt{node}(v_1,\, v_2,\, v_3)) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau), \texttt{ret} : T(\tau)) \mid \vec{a})$$

E-CaseT-Leaf Suppose the last rule applied for the evaluation judgment is *E-CaseT-Leaf*.

E-CASET-LEAF
$$\frac{V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{leaf} \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\, y,\, z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

M-CASET
$$\frac{\begin{array}{c}\mathrm{unn}^{x}(\mathrm{unn}^{x'}(\vec{a})) = \mathrm{unn}^{x}(\vec{c}) \qquad \mathrm{unn}^{x}(\mathrm{unn}^{x'}(\vec{b})) = \mathrm{unn}^{x}(\vec{d}) \qquad \Gamma, t : T(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{b})\end{array}}{\Gamma, t : T(\sigma) \mid \hat{\Upsilon}^{t,t'}_{t}(\vec{a}) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\, y,\, z) \to e_2 : \tau \mid \hat{\Upsilon}^{t,t'}_{t}(\vec{b})}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$
- $\Gamma, t : T(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d}$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{c}) \geq p$
(3) $\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{c}) + q$
$\geq \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{d}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, both unn and sharing both perfectly conserve potential energy (Lemmas 3.4.1 and 7.8.2), $\texttt{unn}^x(\texttt{unn}^{x'}(\vec{a})) = \texttt{unn}^x(\vec{c})$, and $\texttt{unn}^x(\texttt{unn}^{x'}(\vec{b})) = \texttt{unn}^x(\vec{d})$, both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \vec{c}) = \Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid \hat{\curlyvee}_t^{t,t'}(\vec{a}))$$

$$\Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \vec{d})$$
$$= \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid \curlyvee_t^{t,t'}(\vec{b}))$$

E-CaseT-Node Suppose the last rule applied for the evaluation judgment is *E-CaseT-Node*.

E-CASET-NODE
$$\frac{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

M-CASET
$$\frac{\begin{array}{c}\texttt{unn}^x(\texttt{unn}^{x'}(\vec{a})) = \texttt{unn}^x(\vec{c}) \qquad \texttt{unn}^x(\texttt{unn}^{x'}(\vec{b})) = \texttt{unn}^x(\vec{d}) \qquad \Gamma, t : T(\sigma) \mid \vec{c} \vdash e_1 : \tau \mid \vec{d} \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \hat{\lhd}_{x,y,z}^{t'}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}_{x,y,z}^{t'}(\vec{b})\end{array}}{\Gamma, t : T(\sigma) \mid \hat{\curlyvee}_t^{t,t'}(\vec{a}) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid \hat{\curlyvee}_t^{t,t'}(\vec{b})}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 v_3 : T(\sigma)$. Then further, the rule *V-Node* can be inverted to learn all of $v_1 : T(\sigma)$, $v_2 : \sigma$, and $v_3 : T(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$$

Each of the following judgments has now been found:

- $V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$
- $\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \hat{\lhd}_{x,y,z}^{t'}(\vec{a}) \vdash e_2 : \tau \mid \hat{\lhd}_{x,y,z}^{t'}(\vec{b})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \hat{\lhd}_{x,y,z}^{t'}(\vec{a}))$
$\geq p$

(3) $\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{a})) + q$
$\geq \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \mathtt{ret} : \tau) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{b}))$
$+ p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because shifting and sharing both perfectly conserve potential energy (Lemmas 7.6.2 and 7.7.2), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{a}))$$

$$= \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma)) \mid \hat{\curlyvee}^{t,t'}_{t}(\vec{a}))$$

$$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \mathtt{ret} : \tau) \mid \hat{\lhd}^{t'}_{x,y,z}(\vec{b}))$$

$$= \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \mathtt{ret} : \tau) \mid \hat{\curlyvee}^{t,t'}_{t}(\vec{b}))$$

□

7.10 Automation

By design, the multivariate resource functions described in this chapter are still expressible as linear combinations of basis resource functions, and therefore they do not significantly impact how AARA is automated. Thus, type inference still follows the following two steps:

1. basic type inference
2. collect and solve linear contraints

The main difference between type inference here and in other chapters lies in the details of step 2, where the number of linear constraints generated is related to the number of resource functions considered. This number is highly dependent upon how the user might choose to limit the resource functions considered, but there are a few features to be aware of that encourage large numbers of resource functions: The number of resource functions for a given context grows multiplicatively with the number of resource functions for a variable in the context, and both sharing and tree shifting involve combinatorially defined objects (`shuff` and `cut`). This circumstance could potentially pressure a user to support many more resource functions than a comparable univariate system. But in practice, this situation is not much of a concern. While many wild and complex resource functions are possible, usually only relatively simple ones like $\left\{ {n+1 \atop i+1} \right\} \cdot \left\{ {m+1 \atop j+1} \right\}$ are needed. The AARA implementation Resource Aware ML also uses multivariate polynomial resource functions with similar pressures, and it is still practical and efficient.

In total then, type inference in this system is efficient. After basic type inference, the rest of type inference can be reduced to collecting and solving linear constraints, which—given a fixed collection of resource functions—only takes polynomial time in the size of the source code.

7.11 Related Work

This chapter presents a multivariate variant of AARA, but multivariate AARA has been developed before. After discussing these multivariate works, this section goes on to discuss multivariate analyses in other domains.

Hoffmann et al.

This chapter's work should be seen as the exponential counterpart to Hoffmann et al.'s development [80], as both only cover lists and trees, and both treat trees as lists of their elements. However, Hoffmann et al.'s system predates remainder contexts, and so it does not use them, unlike the exponential system presented in this chapter. Another difference is that Hoffmann et al.'s work includes the cost-free type optimizations for function applications that I discuss in Section 7.8.

It is likely that the work of this chapter can be directly combined with Hoffmann et al.'s system through some sort of mixing construction, as in Section 6.6. This mixing might also enable more kinds of optimizations like demotion rules. However, I leave such mixing for future work.

Grosen et al.

Grosen et al.[6] have shown that the multivariate polynomial system can be extended naturally over arbitrary regular recursive data structures so that the indices of data structures correspond almost perfectly to their values [65]. This extension generates natural notions of resource functions for all such data structures, including the existing resource functions for lists and more interesting ones for mutually-recursive data structures like rose trees (see, e.g., Figure 1 in that work for some rose tree examples). Such resource functions can correspond to properties like how left-tilted a tree is by using an annotation index that counts the total number of left descendants from all nodes. Such an annotation index looks like $\mathtt{node}(\mathtt{node}(\mathtt{end}, \langle\rangle, \mathtt{end}), \langle\rangle, \mathtt{end})$, which uses features this chapter's annotation index system does not have, including tree structure and the index `end` that matches against recursive type constructors.

It is likely that a similar generalization exists for the exponential system of this chapter. However, I leave this development for future work. One key hurdle that would need to be overcome is to find the proper generalizations of the definitions of sharing and shifting, which are already rather involved just for lists.

Hofmann and Moser

Hofmann and Moser provide a different approach in their multivariate AARA-based system for term rewriting [88]. In their system, resource functions are defined via tree automata. It seems likely that such a setup could also be applied in traditional AARA.

The tree automata of Hofmann and Moser are bottom-up and nondeterministic. They consume abitrary constructor-defined data structures and generate resource functions based on the

[6]One of the authors is myself.

number of accepting runs of the automaton on a given value. These automata can generate multivariate polynomial resource functions of Hoffmann et al. as well as exponentials like this chapter. It is not clear how their work compares to Grosen et al.'s multivariate resource functions.

However, it is less clear how to select automata measuring meaningful properties of data structures. Whereas the pattern-matching approach employed by other multivariate AARA systems yields a clear combinatorial interpretation in terms of values' structures, the properties counted by tree automata are somewhat opaque. Hofmann and Moser do explain how one could use, e.g., ranking functions on automaton state to ensure polynomial resource functions, but it would appear additional techniques would be needed to pick out, e.g., exponentials.

Other Automatic Multivariate Techniques

Other automatic cost analyses do not usually infer multivariate cost bounds. There are a variety of reasons for this circumstance, a few of which I mention here.

One reason is simply that automatic cost analysis is a relatively young area of research, and so only a few systems are mature enough to care about solving the problems that multivariate bounds solve. Hoffmann et al.'s work came out in 2012 claiming to be the first to provide such bounds, and this thesis is only being written in 2024. Many alternative approaches still only focus on the derivation of univariate cost bounds.

Another reason could be that the use of multivariate bounds could be somewhat specific to functional programming. The accumulator code pattern for tail-recursion is clearly not relevant to imperative programs, but most work in automatic program analysis targets imperative programs.

Regardless of these reasons, some other automatic techniques should in principle be able to approach multivariate bounds. For example, techniques making use of Gröbner bases [48, 128] should already be able to handle multivariate polynomials.

Chapter 8

Linear Maps for Cost-Free Typing

This chapter[1] grapples with a technical complication to AARA that concerns the tradeoffs AARA makes between efficiency and analytical power: *cost-free types*. These are types which are derived in cost model with no cost. Such cost-free types are surprisingly useful across many facets of the AARA analysis, ranging from compositional or non-tail recursive code patterns[78] to logarithmic or multivariate resource functions [80, 89] to parallel execution models[79] and more. However, the established method of inferring and using cost-free types, particularly for non-tail recursive code, is quite computationally expensive due to a phenomenon known as *resource-polymorphic recursion* [78]. This method is also specialized to polynomial AARA, and thus it is not applicable to exponential AARA or other systems from Chapter 6. Thus, the use of cost-free types involves multiple tradeoffs between analytical power and efficiency. The work of this chapter aims to allow for different tradeoffs by laying out an alternative method of cost-free type inference. Where applicable, this new method provides an exponential speedup over the established method.

The new method of cost-free type inference presented in this chapter is based on linear maps, i.e., matrices. Many of AARA's basic annotation manipulations are linear maps. This fact is already used in other chapters, especially Chapter 6 where shifting is explicitly represented with a linear map. The idea of this chapter is to directly infer such linear maps for entire functions, rather than inferring the annotation vectors that such maps act upon. Such a functional transformation of annotations only needs to be inferred once to cover essentially all annotation-to-annotation mappings that could be desired. The established method of cost-free type inference does not have this once-and-done property.

This chapter's work is terminal in this thesis, so no other chapter builds off of it. Moreover, this is the only chapter in this thesis to use cost-free types.

8.1 Cost-Freedom

Before one can understand the contributions of this chapter, it is necessary to understand what cost-free types are and how AARA can use them. Thus, I provide this section to define cost-free types and describe some of their uses.

[1]This chapter is based on joint work between myself, Jan Hoffmann, Thomas Reps, and Jessie Grosen.

Definition 8.1.1 (cost-free type)**.** *A cost-free type is an AARA type τ derived in the trivial cost model where no resources are ever spent or accumulated. For example, cost-free types include τ derived by*

$$\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$$

where every type in Γ is cost-free and where every `tick`$\{r\}$ *in e has $r = 0$.*

Cost-free types are defined in Definition 8.1.1. In contrast, I use the term "costful" to describe types which AARA derives in an arbitrary cost model. As a result, cost-free types are a special case of costful types.

Despite the unassuming definition of cost-free types, they encode a lot of useful information about the behaviour of the underlying code that general costful types do not. For example, suppose that a function can be assigned the cost-free type $L^1(\mathbb{Z}) \to L^2(\mathbb{Z}) \sim L^0(\mathbb{Z})$ in polynomial AARA. The input list has one unit of energy per element and the output has two. Because energy cannot be gained in a cost-free cost model, this change in density can only mean that the output list is no more than half the length of the input list. As a result, the cost-free cost analysis can act as a sort of size analysis.

However, the most important use of cost-free types is for function specialization, which is an important ingredient in deriving tight cost bounds. When a function is assigned a type, that type fixes the energy annotations of its inputs. Thus, to get the best cost bounds, this function type should be specialized to the arguments that the function actually takes at its call sites. However, if a function is called in multiple places, those argument typings may differ, so no single function typing would be optimal. It would therefore be valuable to be able to type functions in a more flexible manner.

Cost-free types enable more flexible function typing. Because cost-free function types describe functions without costs, such cost-free types only describe how potential energy can get moved around. That is, cost-free function types describe how AARA can *reallocate* potential energy from input to output data structures. This ability is perfect for specializing function types to the excess potential energy on input data structures, as all excess can be reallocated to the output according to the cost-free function type. Formally, if the function f can be given the costful type $\tau \overset{\vec{a}|\vec{b}}{\to} \sigma$ and the cost-free type $\tau \overset{\vec{c}|\vec{d}}{\to} \sigma$, then $\tau \overset{\vec{a}+k\cdot\vec{c}|\vec{b}+k\cdot\vec{d}}{\to} \sigma$ is also a valid costful type for $k \geq 0$.[2] Thus, cost-free types serve much the same role as, e.g., homogeneous solutions to differential equations. Just as any of the solutions of the homogeneous equation can be added to the solution of the inhomogeneous equation to obtain another solution to the inhomogeneous equation, any cost-free type can be added to a costful type to obtain another valid costful type.

To see the use of cost-free types, consider an identity function on integer lists which is costfully typed $L^1(\mathbb{Z}) \to L^0(\mathbb{Z}) \sim L^0(\mathbb{Z})$ so that it takes one unit of energy per element of the input list. This type alone is insufficient to find a good costful type for the simple function composition `id` $\circ$ `id`. This composition should have the costful type $L^2(\mathbb{Z}) \to L^0(\mathbb{Z}) \sim L^0(\mathbb{Z})$ because one unit of energy per element is taken twice from the input list. However, deriving

[2]It is critical that $k \geq 0$ so that cost-free types are *added* and not subtracted. Among other reasons, repeatedly subtracting a cost-free type that lossily reallocates potential energy can unsoundly result in an arbitrarily low, even negative, net cost bound.

```
1    fun half lst = case lst of
2      | [] -> []
3      | x1::xs1 -> case xs1 of
4        | [] -> []
5        | x2::xs2 -> let tmp = half xs2 in x1::tmp
```

Figure 8.1: Code for `half`

this type requires the additional costful typing `id` : $L^2(\mathbb{Z}) \to L^1(\mathbb{Z}) \sim L^0(\mathbb{Z})$ for the righthand instance of `id`. This new costful type can be obtained from the first by adding the cost-free type $L^1(\mathbb{Z}) \to L^1(\mathbb{Z}) \sim L^0(\mathbb{Z})$, indicating extra input energy can be reallocated to the output at a 1:1 rate.

While that example was rather easy, by far the most pathological sort of function specialization concerns a phenomenon known as *resource-polymorphic recursion*. Resource-polymorphic recursion refers to situations in which the type annotations required of a recursively called function differ from those used at the function entry.[3] This situation typically occurs when the call is non-tail-recursive and uses non-linear resource functions (but may still occur even with only linear resource functions [77]).

The difficulty with resource-polymorphic recursion is that typing a function itself requires a different specialization of that same function type, which in turn requires its own specialization, ad infinitum. To see this phenomenon in action, consider what happens to quadratic potential in the function `half` for halving the length of a list from n to $\lfloor n/2 \rfloor$, given in Figure 8.1. After, I will discuss how cost-free types solve the problem of resource-polymorphic recursion.

One can start by reasoning semantically about how the input's energy should be transformed by the function `half`. Initially, type the function `half` to take in a list `lst` : $L^{1,0}(\mathbb{Z})$ with one unit of quadratic energy, i.e., $\binom{n}{2}$ total energy. The code contains no "tick" expressions, so no energy is ever consumed to pay for costs. If no energy is lost, then a good AARA type should attempt to reallocate all the energy from the input to the output, and should thus re-express the amount in terms of $\lfloor n/2 \rfloor$ because the output list is half the length of the input. This goal is accomplished best when the output list is given the type $L^{4,1}(\mathbb{Z})$, having 4 units of quadratic energy and 1 unit of linear energy, because $\binom{n}{2} \geq 4\binom{\lfloor n/2 \rfloor}{2} + \binom{\lfloor n/2 \rfloor}{1}$, and this inequality is actually tight for even n. So one would like to show that `half` : $L^{1,0}(\mathbb{Z}) \to L^{4,1}(\mathbb{Z}) \sim L^{0,0}(\mathbb{Z})$, with no remainder necessary. (I therefore elide remainders for the rest of this section.)

Now compare this result to how potential energy is actually transformed by AARA's typing rules in lines 3 to 5 of Figure 8.1:

- From line 3 to the start of line 5, the input list `lst` gets broken up into two head elements (`x1` and `x2`) and a tail list `xs2`. To transfer all potential to the tail, AARA makes use of Pascal's identity twice: $\binom{m+2}{k+2} = \binom{m+1}{k+2} + \binom{m+1}{k+1} = \binom{m}{k+2} + 2\binom{m}{k+1} + \binom{m}{k}$. Reallocating potential according to this rule turns 1 unit of quadratic potential on `lst` into 1 unit of quadratic potential, 2 units of linear potential, and 1 unit of constant potential on `xs2`, because `xs2` has two fewer elements than `lst`. This allocation results in the type of `xs2`

[3]This issue could be characterized as a sort of frame problem like that solved by separation logic's frame rule [120].

being $L^{1,2}(\mathbb{Z})$ and 1 unit of free potential being leftover. This result makes sense because $\binom{m}{2} + 2\binom{m}{1} + 1 = \binom{m+2}{2}$.

- The next part of line 5 performs a recursive call on `xs2` and binds it to `tmp`. However, the function type assigned to `half` by the above semantic reasoning has an argument type of $L^{1,0}(\mathbb{Z})$, while `xs2` : $L^{1,2}(\mathbb{Z})$. AARA needs a different type for `half` to justify the current one! It turns out that the required new typing for `half` is $\langle L^{1,2}(\mathbb{Z});\, 1\rangle \to \langle L^{4,5}(\mathbb{Z});\, 1\rangle$. One can see that this is semantically valid by noting that $\binom{2m}{2} + 2\binom{2m}{1} + 1 = 4\binom{m}{2} + 5\binom{m}{1} + 1$.
- Finally, the rest of line 5 adds `x1` onto the front of the list. In terms of potential, this action corresponds to inverting Pascal's identity. This process leaves the output list typed $L^{4,1}(\mathbb{Z})$ as desired, because $4\binom{m}{2} + 5\binom{m}{1} + 1 = 4\binom{m+1}{2} + 1\binom{m+1}{1}$.

This typing therefore works out consistently with the given semantic reasoning, at least as long as there is that additional typing for `half` at the recursive call. However, it is justifying that additional typing `half` : $\langle L^{1,2}(\mathbb{Z});\, 1\rangle \to \langle L^{4,5}(\mathbb{Z});\, 1\rangle$ that is the problem. If one naively tries to justify this new type syntactically with AARA's typing rules, one would find that another new type is needed, `half` : $\langle L^{1,4}(\mathbb{Z});\, 6\rangle \to \langle L^{4,9}(\mathbb{Z});\, 6\rangle$, at the point of the recursive call. Justifying this type would then need yet another new type `half` : $\langle L^{1,6}(\mathbb{Z});\, 15\rangle \to \langle L^{4,13}(\mathbb{Z});\, 15\rangle$, which would need still another type `half` : $\langle L^{1,8}(\mathbb{Z});\, 28\rangle \to \langle L^{4,17}(\mathbb{Z});\, 28\rangle$, and so on.

Cost-free types can solve the problem of resource-polymorphic recursion. The pointwise difference between the desired type $\langle L^{1,0}(\mathbb{Z});\, 0\rangle \to \langle L^{4,1}(\mathbb{Z});\, 0\rangle$ and the first additional type $\langle L^{1,2}(\mathbb{Z});\, 1\rangle \to \langle L^{4,5}(\mathbb{Z});\, 1\rangle$ is $\langle L^{0,2}(\mathbb{Z});\, 1\rangle \to \langle L^{0,4}(\mathbb{Z});\, 1\rangle$. Thus, if this pointwise-difference type can be justified as a cost-free type, it can be added to the desired type to obtain the needed additional type. This cost-free addition avoids needing to derive the additional type directly, which is where resource-polymorphic recursion turns into infinite regress. And indeed, this cost-free type can be justified directly through typing without resource-polymorphic recursion, providing a finite means of deriving good AARA types for non-tail recursive functions like `half`.

Finally, note that resource-polymorphic recursion arises the same way when using exponential resource functions instead of polynomial. In the exponential setting, one can use the identity $\left\{{2n+1 \atop 2}\right\} = 6\left\{{n+1 \atop 4}\right\} + 6\left\{{n+1 \atop 3}\right\} + 3\left\{{n+1 \atop 2}\right\}$ to guess that `half` can be typed as $\langle L^{0,0,1}(\mathbb{Z});\, 0\rangle \to \langle L^{6,6,3}(\mathbb{Z});\, 0\rangle$. However, just like in the polynomial case, a different type for `half` is needed to justify the typing of the recursive call. Specifically, that type is $\langle L^{0,0,4}(\mathbb{Z});\, 3\rangle \to \langle L^{24,24,12}(\mathbb{Z});\, 3\rangle$, which in turn requires the type $\langle L^{0,0,16}(\mathbb{Z});\, 15\rangle \to \langle L^{96,96,48}(\mathbb{Z});\, 15\rangle$ to be justified, and so on.

8.2 The Problem: Costly, Specialized Inference

As explained in Section 8.1, cost-free types are critical for many purposes in AARA, especially for the purpose of sidestepping infinite regress in resource-polymorphic recursion. Thus, a mature AARA cost-analysis system is inclined to infer cost-free types. However, the existing approach to cost-free type inference is problematic in two ways: Firstly, the cost-free inference algorithm can be quite inefficient. Secondly, it is specialized to polynomial resource functions, and so does not apply to, e.g., the exponential resource functions of Chapter 6.

The existing method of inference with cost-free types stems from Hoffmann and Hofmann

[77]. However, the version given there is not general enough to handle many high-degree examples. Improvements were made in later work [80], but the full picture has never been published. Here I sketch the full picture to highlight the innovations of this chapter's new cost-free typing method.

The basic idea behind the existing inference method is to use brute force to infer a new cost-free type for the excess potential present at each and every function call site. Recall that annotations in AARA are initially symbolic during type inference and are only concretized *after* solving a linear program. Thus, this process of retyping amounts to assuming that the annotations at *every* function call are of the form $\vec{a} + \vec{b}$, where $\vec{a}$ matches the costful typing of the function argument, and $\vec{b}$ is some amount of excess. Then the function is retyped using the cost-free typing rules to accept an input annotated $\vec{b}$. Even if it would happen to be the trivial case that the excess $\vec{b}$ is the zero vector, AARA would not be able to take advantage of this fact until *after* the system of linear inequalities is generated and solved. Thus, the types of functions with trivial cost behavior can be equally expensive to infer as those with nontrivial resource-polymorphic recursion.

However, brute-force retyping alone is not sufficient for typing recursive functions. Retyping any function at its recursive call will cause the process to loop. To avoid looping indefinitely, the existing inference method introduces an additional assumption about the structure of potential annotations at recursive calls. Specifically, it is assumed that the highest-degree annotation in the excess vector $\vec{b}$ is 0. This assumption reduces the degree of potential represented by the annotation vectors at each iteration until the base case of linear excess potential. Under the given assumption, the linear case's recursive calls to a function are assumed to precisely match the assumed type of that function,[4] and therefore exhibit no resource-polymorphic recursion. This process mirrors how repeatedly taking the (discrete) derivative of a *polynomial* eventually yields the constant 0 function.

The function `half` from Section 8.1 is an example of a resource-polymorphic typing for which this algorithm works perfectly. At `half`'s recursive call, the needed cost-free type was $\langle L^{0,2}(\mathbb{Z});\ 1\rangle \to \langle L^{0,4}(\mathbb{Z});\ 1\rangle$, which indeed has a 0 in the first position. This type only uses linear energy and does not require any resource-polymorphic recursion to typecheck successfully. In this case, the key heuristic used in the existing inference method allows the algorithm to converge to the desired solution. Using higher-degree resource functions would require more iterations to converge.

Here is a more formal description of the inference algorithm. Let D_{max} be the greatest degree of polynomial resource functions being considered and $\vec{a}$ be the annotation of the type of some recursive function `f`. The linear program generated by the existing inference algorithm is constructed as follows:

1. If $D_{max} = 1$ so that $\vec{a}$ only annotates linear resource functions, assume that $\vec{a}$ is also used for `f`'s recursive call.
2. If $D_{max} > 1$ for the resource functions annotated by $\vec{a}$, assume a cost-free type annotated $\vec{b}$ of degree $D_{max} - 1$ is needed at the recursive call. Attempt to type `f` using the cost-free annotation $\vec{b}$, then use the annotation $\vec{a} + \vec{b}$ at `f`'s recursive call.

[4]That is, the type matches *modulo constant potential*. AARA can already freely pass extra constant potential through functions, so constant potential can be excluded from the concerns of resource-polymorphic recursion.

```
fun dbl lst = case lst of
  | [] -> []
  | x::xs -> x::x::(dbl xs)

fun round lst = case lst of
  | [] -> []
  | x::xs -> x::dbl(round(half xs))
```

Figure 8.2: Code for `dbl` and `round`

In step 2, the annotation at the call is divided into $\vec{a} + \vec{b}$ so that the "extra" potential energy annotated by $\vec{b}$ is retyped in a cost-free manner. Note how step 2 will be repeated for each internal recursive call to f and for each degree of polynomial energy being used to type f. Thus, if f has b recursive calls and potential energy is of degree d, each external call to f results in retyping f approximately b^d times. Every retyping of f entails retyping each helper function called in f, resulting in a combinatorially explosive cascade of retyping. This cost profile is revisited quantitatively in Section 8.9.

While this algorithm works for `half` and many other functions, it is somewhat lucky that it finds any solutions at all. This circumstance is because the key assumption that the algorithm relies on is not guaranteed to be true. Rather, the assumption is just *often* true when dealing with simple pattern matching and polynomial resource functions because Pascal's identity does not change the coefficient of the highest degree binomial coefficient. Thus, that coefficient gets cancelled out when expressing a cost-free type as the difference of types.

Using polynomial resource functions, a counterexample to the assumption can be found in `round` in Figure 8.2. This function computes a list of length equal to the largest number of the form $2^x - 1$ that is less than or equal to the input list's length. The code makes use of `half` in addition to pattern matching to shrink the size of the list before the recursive call, which increases the energy density. As a result, using only linear energy to type function `round` as $\langle L^a(\mathbb{Z}); 0\rangle \to \langle L^a(\mathbb{Z}); 0\rangle$ requires a type of $\langle L^{2a}(\mathbb{Z}); 1\rangle \to \langle L^{2a}(\mathbb{Z}); 1\rangle$ at the recursive call. The linear annotation increases by a factor of two with each round, violating the assumption needed for convergence.

While `round` is somewhat contrived, one does not need to look far to find counterexamples when using non-polynomial cost-bound templates. Unlike the annotation for the highest-degree polynomial resource function, the annotation for the highest-base exponential resource function rarely remains constant, so it cannot usually be cancelled out by taking differences. This variation is caused by the linear recurrence for Stirling numbers $\left\{ {n+1 \atop k+1} \right\} = (k+1)\left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\}$, which includes the scalar $k+1$, whereas Pascal's identity does not. When typing `half` with exponential energy, the different form of the recurrence causes a scaling factor of 4 between the desired type of $\langle L^{0,0,1}(\mathbb{Z}); 0\rangle \to \langle L^{6,6,3}(\mathbb{Z}); 0\rangle$ and the type needed at the recursive call of $\langle L^{0,0,4}(\mathbb{Z}); 3\rangle \to \langle L^{24,24,12}(\mathbb{Z}); 3\rangle$. Thus, the existing approach cannot find good exponential-cost bounds for `half`.

8.3 The Linear Idea: Linear Maps

To deal with the problems of of the inference method laid out in Section 8.2, this chapter develops an alternative cost-free typing approach based on linear maps, i.e., matrices. This method follows from the observation that all the needed cost-free types may often be summarized with a single linear map M. In this sense, these matrices are essentially *summary functions* [37, 38, 121] describing how a function transforms annotations.

In the case for `half` with polynomial or exponential potential, the desired cost-free matrices would be the following:

$$polynomial : \begin{pmatrix} 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad exponential : \begin{pmatrix} 505/12 & 206/3 & 6 & 0 \\ 10 & 22 & 6 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By associating $\langle L^{a,b}(\mathbb{Z});\ c\rangle$ to the vector $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$, one finds that, e.g., the polynomial map M is consistent with each of the polynomial cost-free types found for `half` in Section 8.1. For instance, the types $\langle L^{1,2}(\mathbb{Z});\ 1\rangle \to \langle L^{4,5}(\mathbb{Z});\ 1\rangle$ and $\langle L^{0,2}(\mathbb{Z});\ 1\rangle \to \langle L^{0,4}(\mathbb{Z});\ 1\rangle$ are encoded by the mappings

$$\begin{pmatrix} 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 1 \end{pmatrix}$$

Crucially, these matrices are invariant across recursive calls, which avoids infinite regress. And unlike the existing cost-free typing algorithm, inferring these matrices does not require additional typings of `half`, so type checking/inference with them can be made efficient (Section 8.7).

In Section 8.4, I define a declarative type system to justify matrices like these, and in Section 8.5 I prove that their use is sound. While there are many subtleties, the thrust of this approach is to treat annotation transformations algebraically and identify matrices that satisfy certain inequalities. For example, the polynomial matrix M for `half` satisfies the following inequalities, which correspond to the action that `half` takes on inputs of length 0, 1, or greater, respectively. To clarify notation, the symbol $*$ stands in for the angelic nondeterministic choice of an arbitrary number throughout this chapter, so that, e.g., $* \cdot 0 = 0$, $* + * = *$, and $n \leq *$ for any n.

$$M \leq \begin{pmatrix} * & * & * \\ * & * & * \\ 0 & 0 & 1 \end{pmatrix} \qquad M \leq \begin{pmatrix} * & * & * \\ * & * & * \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \qquad M \leq \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}^{-1} M \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}^{2}$$

Similarly, the exponential matrix satisfies the following inequalities:

$$M \leq \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M \leq \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 4 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad M \leq \begin{pmatrix} 4 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}^{-1} M \begin{pmatrix} 4 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}^{2}$$

To ensure soundness, the full system of Section 8.4 adds a few extra inequalities beyond these.

I also preemptively note that this matrix method of cost-free typing is not a strict improvement over the existing method. While type inference is usually faster with the matrix method, the matrix method cannot type all functions as well as the previous approach. For example, because the matrix method represents cost-free types with linear maps, it cannot express reallocation of potential energy based on a nonlinear function like `min`, nor can it express multiple

$$\tau, \sigma ::= \mathbb{1} \mid \tau \otimes \sigma \mid \tau \oplus \sigma \mid L(\tau) \mid \tau \xrightarrow{M} \sigma$$

Figure 8.3: Cost-free type grammar

choices of potential reallocation. I document all the various concessions that this system makes in Section 8.10, and a comparison between this system and the established cost-free approach is provided in Section 8.9.

Because each cost-free typing approach has its pros and cons, an implementation of cost-free typing should employ each inference approach tactically to achieve the best tradeoffs. A reasonable hybridization is to first attempt the more efficient matrix approach and then default to the previous approach should the matrix approach fail. This hybridization would allow many simpler functions to be handled quickly while still providing the robust coverage of the previous approach. In particular, if a helper function is amenable to the matrix new approach, then that function need not be retyped repeatedly when retyping its callers, cutting short the expensive cascade of retyping described in Section 8.2.

8.4 Type System

This section lays out this chapter's new matrix-based cost-free type system. To describe the new cost-free type system, this section first describes a collection of primitive linear maps in Section 8.4.2. These maps are used by the typing rules (Section 8.4.3) to construct collections of matrices that correspond to how annotations are manipulated along different paths through function bodies. A collections of such matrices can then be summarized with a single matrix to yield the new matrix-based cost-free function type (Section 8.4.1). Such cost-free function types are intended to be used alongside a usual costful AARA type system in the manner described in Section 8.6.

8.4.1 Types

The types supported for the matrix-based cost-free type system are given in Figure 8.3. These types include most of the AARA types that are present in the rest of this thesis, but do not include trees. The reasons for this exclusion are discussed in Section 8.10.

The cost-free function types displayed in Figure 8.3 have a different form than the function types in other chapters. Rather than have two annotation vectors for argument and return, a function is decorated with a matrix describing the transformation of argument annotation vectors into return (and remainder) annotation vectors. For the type $\tau \xrightarrow{M} \sigma$, this matrix M is indexed by pairs of indices i, j where $j \in \texttt{arg}.\mathit{Ind}(\tau) \cup \{\texttt{c}\}$ and $i \in \texttt{arg}.\mathit{Ind}(\tau) \cup \texttt{ret}.\mathit{Ind}(\sigma) \cup \{\texttt{c}\}$, where $\mathit{Ind}(-)$ is defined according the index system of Figure 3.5.

$$Mov^x_y = \begin{array}{c|cc} & x.\mathtt{d}_2 & x.\mathtt{d}_1 \\ \hline x.\mathtt{d}_2 & 0 & 0 \\ x.\mathtt{d}_1 & 0 & 0 \\ y.\mathtt{d}_2 & 1 & 0 \\ y.\mathtt{d}_1 & 0 & 1 \end{array} \qquad \overset{A}{\lhd}{}^x_y = \begin{array}{c|ccc} & x.\mathtt{d}_2 & x.\mathtt{d}_1 & \mathtt{c} \\ \hline x.\mathtt{d}_2 & 0 & 0 & 0 \\ x.\mathtt{d}_1 & 0 & 0 & 0 \\ y.\mathtt{d}_2 & 1 & 0 & 0 \\ y.\mathtt{d}_1 & 1 & 1 & 0 \\ \mathtt{c} & 0 & 1 & 1 \end{array}$$

$$\overset{A}{\rhd}{}^x_y = \begin{array}{c|ccc} & x.\mathtt{d}_2 & x.\mathtt{d}_1 & \mathtt{c} \\ \hline x.\mathtt{d}_2 & 0 & 0 & 0 \\ x.\mathtt{d}_1 & 0 & 0 & 0 \\ y.\mathtt{d}_2 & 1 & 0 & 0 \\ y.\mathtt{d}_1 & -1 & 1 & 0 \\ \mathtt{c} & 1 & -1 & 1 \end{array} \qquad \pi_{\neg x} = \begin{array}{c|ccccc} & x.\mathtt{d}_2 & x.\mathtt{d}_1 & y.\mathtt{d}_2 & y.\mathtt{d}_1 & \mathtt{c} \\ \hline x.\mathtt{d}_2 & 0 & 0 & 0 & 0 & 0 \\ x.\mathtt{d}_1 & 0 & 0 & 0 & 0 & 0 \\ y.\mathtt{d}_2 & 0 & 0 & 1 & 0 & 0 \\ y.\mathtt{d}_2 & 0 & 0 & 0 & 1 & 0 \\ \mathtt{c} & 0 & 0 & 0 & 0 & 1 \end{array}$$

$$Hav_x = \begin{array}{c|ccc} & x.\mathtt{d}_2 & x.\mathtt{d}_1 & \mathtt{c} \\ \hline x.\mathtt{d}_2 & * & * & * \\ x.\mathtt{d}_1 & * & * & * \\ \mathtt{c} & 0 & 0 & 1 \end{array}$$

Figure 8.4: Selected primitive maps with explicit indices for polynomial potential up to degree 2

8.4.2 Primitive Maps

To simplify the presentation of maps in the cost-free type system, more complex maps are built out of a primitive set of maps. Such maps are exemplified in Figure 8.4, where A is the matrix for Pascal's identity as described in Chapter 6. These primitive maps correspond to the annotation manipulations induced by basic evaluation steps and are often representable as simple numerical matrices. The matrix dimensions (and the matrices for "shift" and "unshift" in particular) are implicitly parameterized by D_{max}, the maximum resource function index. Furthermore, every matrix M is implicitly treated as the direct sum $M \oplus I$, so that every matrix acts as the identity on indices it would otherwise neglect. Each primitive map is explained in detail as follows:

- I — This map is just the identity map. It is used when the potential-carrying variables remain unchanged by a program action. (I is not shown in Figure 8.4.)
- Mov_y^x — This map moves all energy on the variable x to y. Each value at index $x.i$ ends up at index $y.i$ after application. It is assumed that this map is only used when y is an unused name or a variable with zero energy. By avoiding name collisions, this map acts as a form of relabelling.
- $\overset{A}{\lhd}\,{}^{x}_{y}$ — This "shift" map applies a transformation to the list indices of x and puts the results under the new name y, adjusting the constant amount as appropriate. This matrix typically appears at a list pattern-matching operation. The particular transformation applied is given by A, which is a shifting matrix from Chapter 6. Depending on the choice of A, one can obtain various resource functions, including polynomials and exponentials. As with Mov_y^x, it is assumed that the name y is unused. Notably, unlike other shift operations in this thesis, $\overset{A}{\lhd}\,{}^{x}_{y}$ does not handle the annotation indices of a list's elements (i.e., those annotation indices with a prefix like `e`).

  To be explicit, entry $(y.\mathtt{d}_i, x.\mathtt{d}_j)$ of $\overset{A}{\lhd}\,{}^{x}_{y}$ is $A_{i,j}$, where both $x.\mathtt{d}_0$ and $y.\mathtt{d}_0$ are identified with the free energy annotation index `c`.
- $\overset{A}{\rhd}\,{}^{x}_{y}$ — This "unshift" map acts as the inverse of $\overset{A}{\lhd}\,{}^{y}_{x}$. This map adjusts a list's annotations when an element is added to its front. To properly define this map, it is critical that A is chosen to be an invertible matrix. (Luckily, a sensible choice of A is typically invertible. Such invertible choices of A include those generating polynomials and exponentials.)

  To be explicit, entry $y.\mathtt{d}_i x.\mathtt{d}_j$ of $\overset{A}{\rhd}\,{}^{x}_{y}$ is $A^{-1}_{i,j}$, where both $x.\mathtt{d}_0$ and $y.\mathtt{d}_0$ are identified with the free energy annotation index `c`.
- Hav_x — This map is a special havoc operation used when pattern matching or instantiating the empty list or a variant. This havocking is represented by putting the symbol $*$ across each row $x.i$. In this work, $*$ stands for an arbitrary choice of number, so that, e.g., $*\cdot 0 = 0$ and $* + r = * \cdot * = *$.
- π_x — This map projects onto indices of the form $x.i$ if x is a variable name, or the index `c` if x is `c`. This projection is accomplished by zeroing all other indices. This notation is extended to sets so that $\pi_{\{x,y\}}$ projects onto both x and y.

  One writes $\pi_{\neg x}$ for the complement projection that projects *away* indices of the form $x.i$.

The map $\pi_{\neg x}$ is used when variable x leaves or (freshly) enters a scope.

Example 8.4.1. In Section 8.1, the last step of typing `half` creates the return value by attaching `x1` to the front of list `tmp`. The informal reasoning about this step transformed annotations using the identity $4\binom{m}{2} + 5\binom{m}{1} + 1 = 4\binom{m+1}{2} + 1\binom{m+1}{1}$. Using the unshift map $\overset{A}{\triangleright}\substack{\texttt{tmp}\\\texttt{ret}}$ where A is the matrix for Pascal's identity, this step is represented with primitive map composition as follows:

$$\overset{A}{\triangleright}\substack{\texttt{tmp}\\\texttt{ret}} \cdot \begin{array}{c}\texttt{tmp.d}_2\\\texttt{tmp.d}_1\\\texttt{c}\end{array}\begin{pmatrix}4\\5\\1\end{pmatrix} = \begin{array}{c|ccc} & \texttt{tmp.d}_2 & \texttt{tmp.d}_1 & \texttt{c}\\ \hline \texttt{tmp.d}_2 & 0 & 0 & 0\\ \texttt{tmp.d}_1 & 0 & 0 & 0\\ \texttt{ret.d}_2 & 1 & 0 & 0\\ \texttt{ret.d}_1 & -1 & 1 & 0\\ \texttt{c} & 1 & -1 & 1\end{array} \cdot \begin{array}{c}\texttt{tmp.d}_2\\\texttt{tmp.d}_1\\\texttt{c}\end{array}\begin{pmatrix}4\\5\\1\end{pmatrix} = \begin{array}{c}\texttt{tmp.d}_2\\\texttt{tmp.d}_1\\\texttt{ret.d}_2\\\texttt{ret.d}_1\\\texttt{c}\end{array}\begin{pmatrix}0\\0\\4\\1\\0\end{pmatrix}$$

8.4.3 Typing Rules

While the cost-free types of functions are the main goal, one must build up to those types compositionally by typing all subexpressions. This typing is formalized using the rules in Figures 8.5 and 8.6. To ease notation, these rules implicitly extend matrix operations pointwise over sets. In particular, $\mathcal{S} \geq M$ means $\forall S \in \mathcal{S}$, the matrix S is a pointwise upper bound on the matrix M. The rules also use the following typing judgement:

$$\Gamma \vdash_{cf} e : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$$

This typing judgment means:

- in typing context Γ, the expression e has type τ
- the evaluation of e might manipulate annotations according to any of the maps in $\mathcal{S}$ and $\mathfrak{C}$, detailed in this section as follows

The basic approach of the cost-free analysis is to use the type system to assign two sets of linear maps, $\mathcal{S}$ and $\mathfrak{C}$, to each program subexpression. (I refer to these sets as $\mathcal{S}$ and $\mathfrak{C}$ throughout this work.) The maps in $\mathcal{S}$ are used to generate the matrix inequalities exemplified in Section 8.3, and there is one map in $\mathcal{S}$ for each possible path through a function's source code to the return. Once a function's body is fully typed, these paths are approximated by the single matrix that annotates the cost-free function type.

The maps in $\mathfrak{C}$ are used to generate some additional matrix inequalities to ensure that energy is properly conserved. There is one map in $\mathfrak{C}$ for each possible path through the program's source to a function application or to the end of certain variables' scopes—those that get bound from let expressions or matched as list elements. The role of $\mathfrak{C}$ is to address the following problem: if a variable has negative annotations when it falls out of scope, then dropping that variable unsoundly gains energy. In other words, weakening is not generally allowable, as is typical for a linear type system. To salvage the ability to weaken, the annotation transformations in $\mathfrak{C}$ are used to check that certain annotations are nonnegative at key points. The role of $\mathfrak{C}$ is discussed more in Section 8.5.

One might wonder why a *set* of matrices $\mathcal{S}$ is tracked, rather than one single matrix. If a set of matrices can be approximated by a single matrix for typing a function, why can't the same be done to simplify down to a single matrix at every branch? The answer is that such a design makes

CF-LET

$$\frac{\Gamma \vdash_{cf} e_1 : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \qquad \Gamma, x : \sigma \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}}{\Gamma \vdash_{cf} \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \pi_{\neg x} \cdot \mathcal{T} \cdot \mathit{Mov}_x^{\texttt{ret}} \cdot \mathcal{S} \mid \mathfrak{C} \cup (\pi_x \cdot \mathcal{T} \cup \mathfrak{D}) \cdot \mathit{Mov}_x^{\texttt{ret}} \cdot \mathcal{S}}$$

CF-VAR

$$\frac{}{\Gamma, x : \tau \vdash_{cf} x : \tau \rightsquigarrow \{\mathit{Mov}_{\texttt{ret}}^{x}\} \mid \emptyset}$$

CF-TICK

$$\frac{}{\Gamma \vdash_{cf} \texttt{tick}\{0\} : \mathbb{1} \rightsquigarrow \{I\} \mid \emptyset}$$

CF-FUN

$$\frac{\begin{array}{cc} \Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma \vdash_{cf} e : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C} & \pi_{\neg\{y,\texttt{ret},\texttt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\texttt{dom}(\Gamma')} \geq 0 \\ \pi_{\{y,\texttt{ret},\texttt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\texttt{dom}(\Gamma')} \geq \pi_{\{y,\texttt{ret},\texttt{c}\}} \cdot \mathit{Mov}_y^{\texttt{arg}} \cdot M \cdot \mathit{Mov}_{\texttt{arg}}^{y} & \mathfrak{C} \cdot \pi_{\neg\texttt{dom}(\Gamma')} \geq 0 \end{array}}{\Gamma' \vdash_{cf} \texttt{fun}\ g\ y\ =\ e : \tau \xrightarrow{M} \sigma \rightsquigarrow \{I\} \mid \emptyset}$$

CF-APP

$$\frac{}{\Gamma, f : \tau \xrightarrow{M} \sigma, x : \tau \vdash_{cf} f\ x : \sigma \rightsquigarrow \{\mathit{Mov}_x^{\texttt{arg}} \cdot M \cdot \mathit{Mov}_{\texttt{arg}}^{x}\} \mid \{\pi_{\{x,\texttt{c}\}}\}}$$

CF-PAIR

$$\frac{}{\Gamma, x : \tau, y : \sigma \vdash_{cf} \langle x,\ y\rangle : \tau \otimes \sigma \rightsquigarrow \{\mathit{Mov}_{\texttt{ret}.2^{\texttt{nd}}}^{y} \cdot \mathit{Mov}_{\texttt{ret}.1^{\texttt{st}}}^{x}\} \mid \emptyset}$$

CF-CASEP

$$\frac{\begin{array}{c} \Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \vdash_{cf} e : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \\ \mathcal{T} = \mathit{Mov}_{x.1^{\texttt{st}}}^{y} \cdot \mathit{Mov}_{x.2^{\texttt{nd}}}^{z} \cdot \mathcal{S} \cdot \mathit{Mov}_z^{x.2^{\texttt{nd}}} \cdot \mathit{Mov}_y^{x.1^{\texttt{st}}} \qquad \mathfrak{D} = \mathfrak{C} \cdot \mathit{Mov}_z^{x.2^{\texttt{nd}}} \cdot \mathit{Mov}_y^{x.1^{\texttt{st}}} \end{array}}{\Gamma, x : \sigma \otimes \rho \vdash_{cf} \texttt{case}\ x\ \texttt{of}\ \langle y,\ z\rangle \rightarrow e : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}}$$

CF-SUML

$$\frac{}{\Gamma, x : \tau \vdash_{cf} \texttt{l}(x) : \tau \oplus \sigma \rightsquigarrow \{\mathit{Mov}_{\texttt{ret.l}}^{x} \cdot \mathit{Hav}_{\texttt{ret.r}}\} \mid \emptyset}$$

CF-SUMR

$$\frac{}{\Gamma, x : \sigma \vdash_{cf} \texttt{r}(x) : \tau \oplus \sigma \rightsquigarrow \{\mathit{Mov}_{\texttt{ret.r}}^{x} \cdot \mathit{Hav}_{\texttt{ret.l}}\} \mid \emptyset}$$

CF-CASES

$$\frac{\begin{array}{cc} \Gamma, x : \sigma \oplus \rho, y : \sigma \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} & \Gamma, x : \sigma \oplus \rho, z : \rho \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \\ \mathcal{U} = \mathit{Hav}_{x.\texttt{r}} \cdot \mathit{Mov}_{x.\texttt{l}}^{y} \cdot \mathcal{S} \cdot \mathit{Mov}_y^{x.\texttt{l}} \cdot \mathit{Hav}_{x.\texttt{r}} & \mathcal{V} = \mathit{Hav}_{x.\texttt{l}} \cdot \mathit{Mov}_{x.\texttt{r}}^{z} \cdot \mathcal{T} \cdot \mathit{Mov}_z^{x.\texttt{r}} \cdot \mathit{Hav}_{x.\texttt{l}} \\ \mathfrak{E} = \mathfrak{C} \cdot \mathit{Mov}_y^{x.\texttt{l}} \cdot \mathit{Hav}_{x.\texttt{r}} & \mathfrak{F} = \mathfrak{D} \cdot \mathit{Mov}_y^{x.\texttt{r}} \cdot \mathit{Hav}_{x.\texttt{l}} \end{array}}{\Gamma, x : \sigma \oplus \rho \vdash_{cf} \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \rightarrow e_1 \mid \texttt{r}(z) \rightarrow e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F}}$$

Figure 8.5: Linear-map-based, cost-free typing rules 1

CF-NIL

$$\frac{}{\Gamma \vdash_{cf} [\,] : L(\tau) \rightsquigarrow \{\pi_{\neg\texttt{ret.e}} \cdot Hav_{\texttt{ret}}\} \mid \emptyset}$$

CF-CONS

$$\frac{}{\Gamma, x : \tau, y : L(\tau) \vdash_{cf} x :: y : L(\tau) \rightsquigarrow \{\overset{A}{\rhd}{}^{t}_{\texttt{ret}} \cdot \pi_{\neg y.\texttt{e}}\} \mid \{\pi_{y.\texttt{e}}\}}$$

CF-CASEL

$$\frac{\begin{array}{c} \Gamma, x : L(\sigma) \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \qquad \mathcal{U} = \pi_{\neg x.e} \cdot Hav_x \cdot \mathcal{S} \cdot \pi_{\neg x.e} \cdot Hav_x \\ \mathcal{V} = \pi_{\neg y} \cdot \overset{A}{\rhd}{}^{z}_{x} \cdot \mathcal{T} \cdot \overset{A}{\lhd}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{E} = \mathfrak{C} \cdot \pi_{\neg x.e} \cdot Hav_x \\ \mathfrak{F} = \mathfrak{D} \cdot \overset{A}{\lhd}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{G} = \pi_y \cdot \overset{A}{\rhd}{}^{z}_{x} \cdot \mathcal{T} \cdot \overset{A}{\lhd}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{H} = \{\pi_{x.\texttt{e}}\} \end{array}}{\Gamma, x : L(\sigma) \vdash_{cf} \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F} \cup \mathfrak{G} \cup \mathfrak{H}}$$

Figure 8.6: Linear-map-based, cost-free typing rules 2

the system more difficult to automate. Approximating the matrices M and N by some $P \leq M, N$ introduces the unknown matrix P, which must be solved for. Having multiple such unknown matrices around can introduce nonlinear constraints, rendering automation impractical. This problem is minimized by not using such P and instead tracking *sets* of matrices in Figures 8.5 and 8.6, which reduces the number of unknown matrices during type inference. These effects are discussed more in Section 8.7.

One intentionally missing typing rule is the structural rule for weakening. This rule is missing for the same reasons discussed in the previous paragraph. As a result of this rule's exclusion, even the form of weakening typically allowed by AARA is not allowed here. That is, freely throwing away energy is not allowed. This lack of energy weakening is part of a larger pattern: the restrictions of linearity appear stronger in this system so that the usual ways of sidestepping them (annotation weakening and sharing) no longer work as well.

Another missing feature in this cost-free type system is sharing, AARA's specialized form of contraction. Usually AARA allows multiple choices for how energy is allocated via sharing, but the typing rules of this section codify one particular choice. In these cost-free rules, all energy is allocated to a variable's first use, leaving 0 potential for future additional uses. By fixing this arbitrary sharing decision, this system avoids full sharing while still remaining able to type many functions of interest. Supporting full sharing would yield nonlinear annotation relations and thus prevent the analysis from being captured with simple linear maps and from being automated via a linear program (see Section 8.10).

Nonetheless, remainder contexts do recover some feaures of sharing. For example, if a data structure is pattern-matched and its pieces are never used, the uncomputation of remainder contexts restores the pieces' energy to the original data structure at the end of the pattern match's scope. This restoration allows the data structure to then carry energy for additional uses. As a result, the problematic code patterns concerning sharing are those where a variable is used multiple times with overlapping scopes of use. Additionally, the uncomputation of remainder contexts lessens the amount of variables that must be tracked by $\mathfrak{C}$, as those that are uncomputed are guaranteed to have zeros for annotations.

With those structural features out of the way, the syntax-directed typing rules in Figures 8.5 and 8.6 are mostly direct encodings of the typing rules of the system of Chapter 6. Many rules merely transform $\mathcal{S}$ via the primitive map appropriate for the kind of expression being typed. For example, the typing rule *CF-Pair* merely remaps annotations to pair annotations. However, there are a few typing rules of interest which I go over in the following paragraphs.

The rules *CF-Let* makes nontrivial use of $\mathfrak{C}$. Specifically, *CF-Let* adds matrices to $\mathfrak{C}$ that give the annotations of x at the end of expression e_2. These matrices are used to ensure that x has nonnegative annotations at the end of its scope.

The rule *CF-CaseL* makes similar use of $\mathfrak{C}$ to ensure the binding of the list head is nonnegative. Further this rule adds constraints to $\mathfrak{C}$ to ensure that list element annotations are nonnegative, then adds a map to $\mathcal{S}$ to force the list element annotations to actually be zero. Alongside a similar setup in the rules *CF-Cons* and *CF-Nil*, it can be inductively ensured that list elements never carry energy. The reasons for forcing list element energy to be zero is discussed more in Section 8.10.

The rule *CF-App* also uses $\mathfrak{C}$ nontrivially. This rule includes the matrix $\pi_{\{x,\mathtt{c}\}}$ in $\mathfrak{C}$, which gives the annotation of x (and the free energy) before a function application. These matrices are used to ensure that x has nonnegative annotations before a function application.

Another typing rule of interest is *CF-Tick* because tick expressions define the cost model being used. Fundamentally, there are two ways one could treat tick expressions so as to force a cost-free cost model for cost-free typing. The first way is to simply treat the tick parameter as zero regardless of what it actually is. While this approach is often taken in implementations of cost-free type inference, this approach is inconsistent with the cost semantics provided here, and thus it would require additional tinkering to work with. Instead, the rule *CF-Tick* takes the alternative approach, which is to require that the tick parameter is zero. Because ticks express zero cost, the tick expression is just syntactic sugar for a unit expression $\langle\rangle$.

The function-typing rule *CF-Fun* contains the most interesting change from standard AARA. This rule makes use of three special matrix-inequality premisses that no other rule has, and these premisses are used to allow the sound approximation of sets of linear maps with a single matrix. The use of these inequalities can be seen in Section 8.5. Intuitively, these premisses encode the following additional information, assuming that annotations are initially non-negative:

- $\pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^y$ — The matrices in $\mathcal{S}$ yield a pointwise upper bound on the matrix M (modulo some index manipulation).
- $\pi_{\neg\{y,\mathtt{ret},\mathtt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0$ — Variables captured in the function's closure[5] do not have negative annotations when the function body finishes evaluating.
- $\mathfrak{C} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0$ — Arguments to functions and certain variables that fall out of scope in the course of evaluating the function body do not have negative annotations.

[5]That is, those variables bound outside the function but referred to in the function body. These variables should not contribute energy during the course of function evaluation.

Example 8.4.2. Consider typing `half` from Figure 8.1 as $L(\mathbb{Z}) \overset{M}{\to} L(\mathbb{Z})$ with polynomial potential. The typing rule *CF-Fun* can be used to justify that the following matrix from Section 8.1 works for M. To simplify this example, the entries and matrices involved in uncomputation are elided, as well as manipulations of list element annotations, as all matrix entries involved in these operations are zero here.

$$\begin{array}{c c} & \begin{array}{ccc} \texttt{arg.d}\mathit{2} & \texttt{arg.d}\mathit{1} & \texttt{c} \end{array} \\ \begin{array}{c} \texttt{ret.d}\mathit{2} \\ \texttt{ret.d}\mathit{1} \\ \texttt{c} \end{array} & \begin{pmatrix} 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

There are three paths through the body of `half`, so the typing rules will generate three matrices in the set $\mathcal{S}$. The first two of these paths return the empty list for inputs of length zero or one and correspond to the following matrix products[6] (where $*$ represents havocked matrix entries):

$$Hav_{\texttt{ret}} \cdot Hav_{\texttt{lst}} = \begin{array}{c c} & \begin{array}{ccc} \texttt{lst.d}\mathit{2} & \texttt{lst.d}\mathit{1} & \texttt{c} \end{array} \\ \begin{array}{c} \texttt{lst.d}\mathit{2} \\ \texttt{lst.d}\mathit{1} \\ \texttt{ret.d}\mathit{2} \\ \texttt{ret.d}\mathit{1} \\ \texttt{c} \end{array} & \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ 0 & 0 & 1 \end{pmatrix} \end{array} \qquad \pi_{\neg\{\texttt{xs1,x1}\}} \cdot Hav_{\texttt{ret}} \cdot Hav_{\texttt{xs1}} \cdot \overset{A}{\triangleleft}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{c c} & \begin{array}{ccc} \texttt{lst.d}\mathit{2} & \texttt{lst.d}\mathit{1} & \texttt{c} \end{array} \\ \begin{array}{c} \texttt{lst.d}\mathit{2} \\ \texttt{lst.d}\mathit{1} \\ \texttt{ret.d}\mathit{2} \\ \texttt{ret.d}\mathit{1} \\ \texttt{c} \end{array} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ * & * & * \\ * & * & * \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

The remaining path through `half` is that which makes a recursive call on inputs of length greater than one. Using the given choice of M, the typing rules generate the following matrix product:

$$\pi_{\neg\{\texttt{xs1,x1}\}} \cdot \pi_{\neg\{\texttt{xs2,x2}\}} \cdot \pi_{\neg\texttt{tmp}} \cdot \overset{A}{\triangleright}{}^{\texttt{tmp}}_{\texttt{ret}} \cdot Mov^{\texttt{ret}}_{\texttt{tmp}} \cdot M \cdot Mov^{\texttt{xs2}}_{\texttt{arg}} \cdot \overset{A}{\triangleleft}{}^{\texttt{xs1}}_{\texttt{xs2}} \cdot \pi_{\neg\texttt{x2}} \cdot \overset{A}{\triangleleft}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{c c} & \begin{array}{ccc} \texttt{lst.d}\mathit{2} & \texttt{lst.d}\mathit{1} & \texttt{c} \end{array} \\ \begin{array}{c} \texttt{lst.d}\mathit{2} \\ \texttt{lst.d}\mathit{1} \\ \texttt{ret.d}\mathit{2} \\ \texttt{ret.d}\mathit{1} \\ \texttt{ret} \end{array} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

The typing rule *CF-Fun* then requires that, for each of these $S \in \mathcal{S}$, two sets of inequalities hold.

The first set of inequalities specifies that, after identifying the labels `lst` and `arg`, each matrix S is a pointwise upper bound on M over the indices for `lst`, `ret`, and `c`, i.e., $\pi_{\{\texttt{lst,ret,c}\}} \cdot S \cdot \pi_{\neg\texttt{dom}(\Gamma)} \geq \pi_{\{\texttt{lst,ret,c}\}} \cdot Mov^{\texttt{arg}}_{\texttt{lst}} \cdot M \cdot Mov^{\texttt{lst}}_{\texttt{arg}}$. These inequalities simplify to those given in Section 8.3, which M satisfies.

The second set of inequalities specifies that each matrix $S \in \mathcal{S}$ avoids negative entries over indices for captured variables, i.e., $\pi_{\neg\{\texttt{lst,ret,c}\}} \cdot S \cdot \pi_{\neg\texttt{dom}(\emptyset)} \geq 0$. These second inequalities are trivial because there are no captured variables in `half`, so the inequalities only need to hold on a space of dimension 0.

[6]Recall that each matrix N is implicitly extended to $N \oplus I$, so matrices of mismatched dimensions can be multiplied.

The typing rules also generate the following matrices in $\mathfrak{C}$ for variables that leave scope.

$$\pi_{\{\texttt{xs1},\texttt{x1}\}} \cdot \mathit{Hav}_{\texttt{ret}} \cdot \mathit{Hav}_{\texttt{xs1}} \cdot \overset{A}{\lhd}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{r} \\ \texttt{x1}.\mathit{Ind}(\mathbb{Z}) \\ \texttt{xs1.d}_{\mathit{2}} \\ \texttt{xs1.d}_{\mathit{1}} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{lst.d}_{\mathit{2}} & \texttt{lst.d}_{\mathit{1}} & \texttt{c} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ * & * & * \\ * & * & * \end{pmatrix} \end{array}$$

$$\pi_{\{\texttt{xs1},\texttt{x1}\}} \cdot \pi_{\neg\{\texttt{xs2},\texttt{x2}\}} \cdot \pi_{\neg\texttt{tmp}} \cdot \overset{A}{\rhd}{}^{\texttt{tmp}}_{\texttt{ret}} \cdot \mathit{Mov}^{\texttt{ret}}_{\texttt{tmp}} \cdot M \cdot \mathit{Mov}^{\texttt{xs2}}_{\texttt{arg}} \cdot \overset{A}{\lhd}{}^{\texttt{xs1}}_{\texttt{xs2}} \cdot \pi_{\neg\texttt{x2}} \cdot \overset{A}{\lhd}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{r} \\ \texttt{x1}.\mathit{Ind}(\mathbb{Z}) \\ \texttt{xs1.d}_{\mathit{2}} \\ \texttt{xs1.d}_{\mathit{1}} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{lst.d}_{\mathit{2}} & \texttt{lst.d}_{\mathit{1}} & \texttt{c} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\pi_{\{\texttt{xs2},\texttt{x2}\}} \cdot \pi_{\neg\texttt{tmp}} \cdot \overset{A}{\rhd}{}^{\texttt{tmp}}_{\texttt{ret}} \cdot \mathit{Mov}^{\texttt{ret}}_{\texttt{tmp}} \cdot M \cdot \mathit{Mov}^{\texttt{xs2}}_{\texttt{arg}} \cdot \overset{A}{\lhd}{}^{\texttt{xs1}}_{\texttt{xs2}} \cdot \pi_{\neg\texttt{x2}} \cdot \overset{A}{\lhd}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{r} \\ \texttt{x2}.\mathit{Ind}(\mathbb{Z}) \\ \texttt{xs2.d}_{\mathit{2}} \\ \texttt{xs2.d}_{\mathit{1}} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{lst.d}_{\mathit{2}} & \texttt{lst.d}_{\mathit{1}} & \texttt{c} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\pi_{\texttt{tmp}} \cdot \overset{A}{\rhd}{}^{\texttt{tmp}}_{\texttt{ret}} \cdot \mathit{Mov}^{\texttt{ret}}_{\texttt{tmp}} \cdot M \cdot \mathit{Mov}^{\texttt{xs2}}_{\texttt{arg}} \cdot \overset{A}{\lhd}{}^{\texttt{xs1}}_{\texttt{xs2}} \cdot \pi_{\neg\texttt{x2}} \cdot \overset{A}{\lhd}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{r} \\ \texttt{tmp.d}_{\mathit{2}} \\ \texttt{tmp.d}_{\mathit{1}} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{lst.d}_{\mathit{2}} & \texttt{lst.d}_{\mathit{1}} & \texttt{c} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

And one last matrix is generated in $\mathfrak{C}$ for function arguments:

$$\pi_{\{\texttt{arg},\texttt{c}\}} \cdot \mathit{Mov}^{\texttt{xs2}}_{\texttt{arg}} \cdot \overset{A}{\lhd}{}^{\texttt{xs1}}_{\texttt{xs2}} \cdot \pi_{\neg\texttt{x2}} \cdot \overset{A}{\lhd}{}^{\texttt{lst}}_{\texttt{xs1}} \cdot \pi_{\neg\texttt{x1}} = \begin{array}{r} \\ \texttt{arg.d}_{\mathit{2}} \\ \texttt{arg.d}_{\mathit{1}} \\ \texttt{c} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{lst.d}_{\mathit{2}} & \texttt{lst.d}_{\mathit{1}} & \texttt{c} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix} \end{array}$$

The rule *CF-Fun* then requires that, for each $C \in \mathfrak{C}$, the matrix C is non-negative, i.e., $C \cdot \pi_{\neg\texttt{dom}(\emptyset)} \geq 0$. This condition holds for all of the matrices in $\mathfrak{C}$ (which were shown above), finishing the justification for typing `half` as $L(\mathbb{Z}) \xrightarrow{M} L(\mathbb{Z})$.

8.4.4 Well-Formed Values

The notion of well-formed values must be altered to account for this chapter's cost-free typing rules. Only one change needs to be made: the well-formedness of functions must account for the cost-free typing of the function. Morally speaking, however, nothing has changed about the well-formedness of values since Figure 3.8; function closures still are only well-formed if they admit a typing derivation. For completeness, I provide the changed rule as *V-FunCF* in Figure 9.11 alongside all the other unchanged rules.

These cost-free well-formedness rules are specialized to the cost-free system presented in this chapter. However, in a more general costful setting, the well-formedness rule for functions would need to be combined with its costful analogue. This combination is discussed further in Section 8.6.

8.5 Soundness

The notion of soundness for the cost-free system of this chapter differs somewhat from other chapters of this thesis because peak and net costs are (mostly) irrelevant in a cost-free setting. Additionally, the cost-free system does not deal directly with annotation vectors, so more indirect

$$\textsc{V-Unit}\ \frac{}{\langle\rangle : \mathbb{1}} \qquad \textsc{V-FunCF}\ \frac{V : \Gamma \qquad \Gamma \vdash_{cf} \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{M} \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C}}{\texttt{C}(V;\ f,\ x.\,e) : \tau \xrightarrow{M} \sigma} \qquad \textsc{V-Pair}\ \frac{v_1 : \tau \qquad v_2 : \sigma}{\langle v_1,\ v_2\rangle : \tau \otimes \sigma}$$

$$\textsc{V-SumL}\ \frac{v : \tau}{\texttt{l}(v) : \tau \oplus \sigma} \qquad \textsc{V-SumR}\ \frac{v : \sigma}{\texttt{r}(v) : \tau \oplus \sigma} \qquad \textsc{V-Nil}\ \frac{}{[\,] : L(\tau)} \qquad \textsc{V-Cons}\ \frac{v_1 : \tau \qquad v_2 : L(\tau)}{v_1 :: v_2 : L(\tau)} \qquad \textsc{V-Nont}\ \frac{}{\bullet : \tau}$$

$$\textsc{V-Context}\ \frac{\forall x \in \texttt{dom}(\Gamma).\, V(x) : \Gamma(x)}{V : \Gamma}$$

Figure 8.7: Cost-free value well-formedness rules

means must be employed to relate cost-free types back to potential energy. These indirect means take the form of additional conditions in the theorem concerning matrix inequalities.

The important soundness property here is that the annotation maps do not cause an increase in potential energy. This property is the correct notion of soundness because it ensures that, at best, the maps used can only move energy around. The effects of this notion of soundness are discussed more in Section 8.6, wherein it is made clear how this cost-free soundness interacts with costful AARA.

The soundness property is non-trivial due to the unavoidable presence of negative matrix entries, like in the primitive map $\overset{A}{\triangleright}{}^{x}_{y}$ for the polynomial system. Such entries can create negative annotations, posing two problems:

- Negative annotations on a variable could mean negative potential on that variable, so that if that variable were to fall out of scope potential would be gained.
- Linear maps that merely safely lose potential when applied to positive annotations will symmetrically gain potential when applied to negative annotations.

The former of these issues is not new to amortized-cost-analysis systems that allow negative potential energy, and has been handled previously by checking that energy is nonnegative before it is dropped [67, 96]. This chapter's cost-free system takes the same approach, but must do so indirectly because the system does not work directly with annotations. Instead this system uses the principle that if both the matrix $C \in \mathfrak{C}$ and annotation vector $\vec{a}$ are nonnegative, then so is $C \cdot \vec{a}$. The latter issue is new to the setting of linear maps, but can be handled similarly.

Forcing some matrices to be pointwise nonnegative might seem to be a heavy-handed way of ensuring nonnegative annotation vector entries, and one might wonder if some more relaxed conditions exist—maybe negative entries could be fine for certain vectors. However, it turns out to be a rather difficult to characterize which relations between matrices and vectors ensure repeated matrix multiplication never yields negative entries.[7] This problem is actually an instance

[7]Such repeated application arises when repeatedly applying a recursive function during evaluation.

of the *positivity problem*. Like its cousin the Skolem problem, the positivity problem is not known to be decidable [119]. Thus, pairing nonnegative matrices with nonnegative vectors is essentially the best available solution.

The soundness theorem is stated in Theorem 8.5.1. This theorem makes use of the potential function defined in Chapter 6. Because this setting allows negative annotations, these definitions can result in negative energy. This theorem also makes use of the evaluation rules of Chapter 2 (excluding those for trees). Notably, because all tick parameters are 0, it is guaranteed that $p = 0$ whenever it holds that $V \vdash e \Downarrow v \mid (p, q)$. Moreover, q is almost always 0 as well, with the exception being $q = \infty$ when nontermination applies (Lemma 2.4.2).

Theorem 8.5.1 (cost-free soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (0, q)$ *(the expression e evaluates to v in context V with q leftover energy)*
- $V : \Gamma$ *(V is well-formed with respect the type context Γ)*
- $\Gamma \vdash_{cf} e : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$ *(Γ types e cost-freely as τ with maps $\mathcal{S}$ and $\mathfrak{C}$)*
- $\mathfrak{C} \cdot \vec{a} \geq 0$ *($\mathfrak{C}$ only maps $\vec{a}$ to non-negative vectors)*
- *$\vec{a}$ annotates Γ*

then

- *$v : \tau$ (the return value is well-formed)*
- $\exists M \in \mathcal{S}. \Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid M \cdot \vec{a})$
  (for terminating evaluation, converting the initial annotation $\vec{a}$ to an annotation of the result by applying M does not gain potential energy)

Proof. This proof proceeds by induction on the evaluation judgment. First, however, the case where $q = \infty$ is covered in the *E-Nont* proof case so that all other cases may assume $q = 0$.

The most interesting proof case is that for the creation of a function, *E-App*, which is where the setup of these typing rules actually pays off. The rule *E-Let* is also somewhat interesting for how it uses $\mathfrak{C}$.

E-Nont Suppose that *E-Nont* is used anywhere in the evaluation judgment. Then Lemma 2.4.2 applies and $q = \infty$. Because ∞ is greater than or equal to anything and $\mathcal{S}$ is always nonempty, the energy bound is then satisfied.

To prove the well-formedness conclusion, there are two cases to consider:

Firstly, suppose the last rule applied for the evaluation judgment is *E-Nont*.

$$\textsc{E-Nont}\quad \frac{}{V \vdash e \Downarrow \bullet \mid (0, \infty)}$$

Then $v = \bullet$. Because $\bullet : \tau$ by *V-Nont*, the needed well-formedness judgment holds.

Secondly, suppose the last rule applied for the evaluation judgment is any other rule. Then well-formedness follows as in that rule's proof case.

E-Tick Suppose the last rule applied for the evaluation judgment is *E-Tick*.

$$\textsc{E-Tick}\quad \frac{}{V \vdash \texttt{tick}\{0\} \Downarrow \langle\rangle \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Tick}\quad \frac{}{\Gamma \vdash_{cf} \texttt{tick}\{0\} : \mathbb{1} \rightsquigarrow \{I\} \mid \emptyset}$$

Thus, $\mathcal{S} = \{I\}$.

Because $\langle\rangle : \mathbb{1}$ by *V-Unit*, the needed well-formedness judgment holds. Finally, the energy bound holds with the following identity at $M = I \in \mathcal{S}$ because units carry no energy and have no annotation indices.

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi(V, \texttt{ret} \mapsto \langle\rangle : \Gamma, \texttt{ret} : \mathbb{1} \mid M \cdot \vec{a})$$

E-Var Suppose the last rule applied for the evaluation judgment is *E-Var*

$$\textsc{E-Var}\quad \frac{}{V, x \mapsto v \vdash x \Downarrow v \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Var}\quad \frac{}{\Gamma, x : \tau \vdash_{cf} x : \tau \rightsquigarrow \{\mathit{Mov}^x_{\texttt{ret}}\} \mid \emptyset}$$

Then $(V, x \mapsto v) : (\Gamma, x : \tau)$. This well-formedness judgment must have been concluded from the well-formedness rule *V-Context*, which means $v : \tau$ must hold as a premiss via inversion. Thus the needed well-formedness judgment holds.

Finally, because the move map just moves the energy to a new label, the energy bound is also satisfied at $M = \mathit{Mov}^x_{\texttt{ret}} \in \mathcal{S}$ with the following identity:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \vec{a}) = \Phi((V, x \mapsto v, \texttt{ret} \mapsto v) : (\Gamma, x : \tau, \texttt{ret} : \tau) \mid M \cdot \vec{a})$$

E-Let Suppose the last rule applied for the evaluation judgment is *E-Let*.

$$\textsc{E-Let}\quad \frac{V \vdash e_1 \Downarrow v' \mid (0,0) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (0,0)}{V \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Let}\quad \frac{\Gamma \vdash_{cf} e_1 : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \qquad \Gamma, x : \sigma \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}}{\Gamma \vdash_{cf} \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \pi_{\neg x} \cdot \mathcal{T} \cdot \mathit{Mov}^{\texttt{ret}}_x \cdot \mathcal{S} \mid \mathfrak{C} \cup (\pi_x \cdot \mathcal{T} \cup \mathfrak{D}) \cdot \mathit{Mov}^{\texttt{ret}}_x \cdot \mathcal{S}}$$

The premisses of both of these rules hold by inversion.

Because $(\mathfrak{C} \cup (\pi_x \cdot \mathcal{T} \cup \mathfrak{D}) \cdot Mov_x^{\mathtt{ret}} \cdot \mathcal{S}) \cdot \vec{a} \geq 0$ by assumption, it true in particular that $\mathfrak{C} \cdot \vec{a} \geq 0$.

Each of the following judgments has now been found:

- $V \vdash e_1 \Downarrow v' \mid (0, 0)$
- $V : \Gamma$
- $\Gamma \vdash_{cf} e_1 : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$
- $\mathfrak{C} \cdot \vec{a} \geq 0$
- $\vec{a}$ annotates Γ

With these judgments, the inductive hypothesis can be applied to learn the following for some $M_1 \in \mathcal{S}$:

(1) $v' : \sigma$
(2) $\Phi(V : \Gamma \mid \vec{a}) \geq \Phi((V, \mathtt{ret} \mapsto v') : \Gamma, \mathtt{ret} : \sigma \mid M_1 \cdot \vec{a})$

Because $V : \Gamma$ by assumption and $v' : \sigma$ by (1), $(V, x \mapsto v') : (\Gamma, x : \sigma)$ follows by *V-Context*.

Next, via relabelling with $Mov_x^{\mathtt{ret}}$, the inequality of (2) can be extended to the following:

$$\Phi(V : \Gamma \mid \vec{a}) \geq \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a})$$

Now because $(\mathfrak{C} \cup (\pi_x \cdot \mathcal{T} \cup \mathfrak{D}) \cdot Mov_x^{\mathtt{ret}} \cdot \mathcal{S}) \cdot \vec{a} \geq 0$ by assumption and $M_1 \in \mathcal{S}$, it is the case that $\pi_x \cdot \mathcal{T} \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a} \geq 0$ and $\mathfrak{D} \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a} \geq 0$.

Each of the following judgments has now been found:

- $V, x \mapsto v' \vdash e_2 \Downarrow v \mid (0, 0)$
- $(V, x \mapsto v') : (\Gamma, x : \sigma)$
- $\Gamma, x : \sigma \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}$
- $\mathfrak{D} \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a} \geq 0$
- $Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a}$ annotates Γ

With these judgments, the inductive hypothesis can be applied to learn the following for some $M_2 \in \mathcal{T}$:

(3) $v : \tau$
(4) $\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a})$
$\geq \Phi((V, x \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma, \mathtt{ret} : \tau) \mid M_2 \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a})$

The well-formedness judgment (3) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven. This bound follows from the following inequalities, where the needed witness matrix is $\pi_{\neg x} \cdot M_2 \cdot Mov_x^{\mathtt{ret}} \cdot M_1$.

$$\begin{aligned}
&\Phi(V : \Gamma \mid \vec{a}) \\
&\geq \Phi((V, \mathtt{ret} \mapsto v') : \Gamma, \mathtt{ret} : \sigma \mid M_1 \cdot \vec{a}) && (2) \\
&= \Phi((V, x \mapsto v') : \Gamma, x : \sigma \mid Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a}) && \textit{relabelling} \\
&\geq \Phi((V, x \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma, \mathtt{ret} : \tau) \mid M_2 \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a}) && (4) \\
&\geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma, \mathtt{ret} : \tau) \mid \pi_{\neg x} \cdot M_2 \cdot Mov_x^{\mathtt{ret}} \cdot M_1 \cdot \vec{a})
\end{aligned}$$

The last inequality follows because the assumption $(\mathfrak{C} \cup (\pi_x \cdot \mathcal{T} \cup \mathfrak{D}) \cdot Mov_x^{\texttt{ret}} \cdot \mathcal{S}) \cdot \vec{a} \geq 0$ ensures $\pi_x \cdot M_2 \cdot Mov_x^{\texttt{ret}} \cdot M_1 \cdot \vec{a} \geq 0$ and energy is monotone (Lemma 3.4.5).

E-Fun Suppose the last rule applied for the evaluation judgment is *E-Fun*.

$$\textsc{E-Fun}\quad \frac{}{V \vdash \texttt{fun}\ f\ x\ =\ e \Downarrow \texttt{C}(V;\ f,\ x.\, e) \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Fun}\quad \frac{\begin{array}{c}\Gamma, f : \tau \xrightarrow{M} \sigma, x : \tau \vdash_{cf} e : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \qquad \pi_{\{x,\texttt{ret},\texttt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\texttt{dom}(\Gamma)} \geq \pi_{\{x,\texttt{ret},\texttt{c}\}} \cdot M \cdot Mov_{\texttt{arg}}^{x} \\ \pi_{\neg\{x,\texttt{ret},\texttt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\texttt{dom}(\Gamma)} \geq 0 \qquad \mathfrak{C} \cdot \pi_{\neg\texttt{dom}(\Gamma)} \geq 0\end{array}}{\Gamma \vdash_{cf} \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{M} \sigma \rightsquigarrow \{I\} \mid \emptyset}$$

Thus, $\mathcal{S} = \{I\}$.

Because $\texttt{C}(V;\ f,\ x.\, e) : \tau \xrightarrow{M} \sigma$ follows from *V-FunCF* and the assumed typing judgment, the needed well-formedness judgment holds. Finally, the energy bound is satisfied by the following identity at $M = I \in \mathcal{S}$ because functions carry no energy and have no annotation indices.

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \texttt{ret} \mapsto \texttt{C}(V;\ f,\ x.\, e)) : (\Gamma, \texttt{ret} : \tau \xrightarrow{M} \sigma) \mid I \cdot \vec{a})$$

E-App Suppose the last rule applied for the evaluation judgment is *E-App*.

$$\textsc{E-App}\quad \frac{V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (0,0)}{V, x \mapsto v', f \mapsto \texttt{C}(V';\ g,\ y.\, e) \vdash f\ x \Downarrow v \mid (0,0)}$$

Then this rule's premiss holds by inversion and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-App}\quad \frac{}{\Gamma, f : \tau \xrightarrow{M} \sigma, x : \tau \vdash_{cf} f\ x : \sigma \rightsquigarrow \{Mov_x^{\texttt{arg}} \cdot M \cdot Mov_{\texttt{arg}}^{x}\} \mid \{\pi_{\{x,\texttt{c}\}}\}}$$

Thus $\pi_{\{x,\texttt{c}\}} \cdot \vec{a} \geq 0$ by assumption.

Because $(V, x \mapsto v', f \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}} \sigma\vec{c})$ by assumption, the rule *V-Context* can be inverted to learn $\texttt{C}(V';\ g,\ y.\, e) : \tau \xrightarrow{\vec{b}} \sigma\vec{c}$. Then further, the rule *V-FunCF* can be inverted to learn that this function body can be typed in some context Γ' where $V' : \Gamma'$. Using *V-Context*, one can then use this well-formedness judgment to derive

$$(V', y \mapsto v', g \mapsto \texttt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma)$$

Now inspect the derivation of the type of the function closure's body. Only *CF-Fun* can conclude a typing derivation for a function. Thus the following rule applies:

CF-FUN

$$\frac{\begin{array}{ll}\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma \vdash_{cf} e : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C} & \pi_{\neg\{y,\mathtt{ret},\mathtt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0 \\ \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot \mathcal{S} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot \mathit{Mov}_y^{\mathtt{arg}} \cdot M \cdot \mathit{Mov}_{\mathtt{arg}}^{y} & \mathfrak{C} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0\end{array}}{\Gamma' \vdash_{cf} \mathtt{fun}\ g\ y\ =\ e : \tau \xrightarrow{M} \sigma \rightsquigarrow \{I\} \mid \emptyset}$$

This rule's premisses hold by inversion.

Now note the inequality resulting from the following chain of implications:

$$\begin{array}{lr} \quad\ \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0 & \mathit{assumption} \\ \Longrightarrow \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0 & \mathit{Mov}_y^x \geq 0 \\ \Longrightarrow \mathfrak{C} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0 & \mathfrak{C}' \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0 \end{array}$$

Each of the following judgments has now been found:

- $V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (0, 0)$
- $(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma)$
- $\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma \vdash_{cf} e : \sigma \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$
- $\mathfrak{C} \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0$
- $\pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}$ annotates $(\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma)$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{S}$:

(1) $v : \tau$

(2) $\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma) \mid \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a})$
$\geq \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma, \mathtt{ret} : \sigma) \mid N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a})$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven.

Then the needed bound can start to be derived from the following inequalities, where the needed witness matrix is $\mathit{Mov}_x^{\mathtt{arg}} \cdot M \cdot \mathit{Mov}_{\mathtt{arg}}^x$.

$$\begin{array}{lr}
\quad \Phi(V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e) : \Gamma, x : \tau, f : \tau \xrightarrow{M} \sigma \mid \vec{a}) & \\
= \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) & \\
\quad + \Phi((x \mapsto v') : (x : \tau) \mid \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) & \mathit{def} \\
= \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) & \\
\quad + \Phi((y \mapsto v') : (y : \tau) \mid \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) & \mathit{relabelling} \\
= \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) & \\
\quad + \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma) \mid \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) & \mathit{zeroed} \\
\geq \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) & \\
\quad + \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{M} \sigma, \mathtt{ret} : \sigma) \mid N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot \mathit{Mov}_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) & (2)
\end{array}$$

Now note that $N \in \mathcal{S}$, so a premiss of *CF-Fun* ensures that $\pi_{\neg\{y,\mathtt{ret},\mathtt{c}\}} \cdot N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq 0$. Moreover, $Mov_y^x \geq 0$ by construction, and $\pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0$ by assumption. The following inequality therefore follows:

$$\pi_{\neg\{y,\mathtt{ret},\mathtt{c}\}} \cdot N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot Mov_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq 0$$

Another premiss of *CF-Fun* yields $\pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \geq \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^y$. The following inequality therefore follows similarly:

$$\pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot Mov_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a} \geq \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^y \cdot Mov_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}$$

The chain of inequalities can then be continued as follows, starting using these new inequalities alongside the monotonicity of energy with respect to annotations:

$$\begin{aligned}
&\geq \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\,e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) \\
&\quad + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot N \cdot \pi_{\neg\mathtt{dom}(\Gamma')} \cdot Mov_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) && \textit{Lemma 3.4.5} \\
&\geq \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\,e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) \\
&\quad + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid \pi_{\{y,\mathtt{ret},\mathtt{c}\}} \cdot Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^y \cdot Mov_y^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) && \textit{Lemma 3.4.5} \\
&= \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\,e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) \\
&\quad + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) && \textit{simplification} \\
&= \Phi(V, f \mapsto \mathtt{C}(V';\ g,\ y.\,e) : \Gamma, f : \tau \xrightarrow{M} \sigma \mid Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^x \cdot \pi_{\neg\{x,\mathtt{c}\}} \cdot \vec{a}) \\
&\quad + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^x \cdot \pi_{\{x,\mathtt{c}\}} \cdot \vec{a}) && \textit{unused indices} \\
&= \Phi((V, y \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e), \mathtt{ret} \mapsto v) : (\Gamma, y : \tau, f : \tau \xrightarrow{M} \sigma, \mathtt{ret} : \sigma) \mid Mov_y^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^x \cdot \vec{a}) && \textit{algebra} \\
&= \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{M} \sigma, \mathtt{ret} : \sigma) \mid Mov_x^{\mathtt{arg}} \cdot M \cdot Mov_{\mathtt{arg}}^x \cdot \vec{a}) && \textit{relabelling}
\end{aligned}$$

E-Pair Suppose the last rule applied for the evaluation judgment is *E-Pair*.

$$\text{E-PAIR}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash \langle x,\ y\rangle \Downarrow \langle v_1,\ v_2\rangle \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\text{CF-PAIR}\quad \frac{}{\Gamma, x : \tau, y : \sigma \vdash_{cf} \langle x,\ y\rangle : \tau \otimes \sigma \rightsquigarrow \{Mov_{\mathtt{ret.2^{nd}}}^y \cdot Mov_{\mathtt{ret.1^{st}}}^x\} \mid \emptyset}$$

Because $\langle v_1,\ v_2\rangle : \tau \otimes \sigma$ follows from *V-Pair* and the assumed well-formedness judgment $(V, x_1 \mapsto v_1, x_2 \mapsto v_2) : (\Gamma, x : \tau, y : \sigma)$, the needed well-formedness judgment holds. Finally, the energy bound is satisfied by the following identity at $M = Mov_{\mathtt{ret.2^{nd}}}^y \cdot Mov_{\mathtt{ret.1^{st}}}^x \in \mathcal{S}$ because the energy of pair is the sum its parts'.

$$\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : \sigma) \mid \vec{a})$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto \langle v_1,\ v_2\rangle) : (\Gamma, x : \tau, y : \sigma, \mathtt{ret} : \tau \otimes \sigma) \mid M \cdot \vec{a})$$

E-CaseP Suppose the last rule applied for the evaluation judgment is *E-CaseP*.

$$\textsc{E-CaseP}\quad \frac{V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (0,0)}{V, x \mapsto \langle v_1,\, v_2\rangle \vdash \texttt{case}\ x\ \texttt{of}\ \langle y,\, z\rangle \to e \Downarrow v \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-CaseP}\quad \frac{\begin{array}{c}\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \vdash_{cf} e : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \\ \mathcal{T} = Mov^{y}_{x.1^{\texttt{st}}} \cdot Mov^{z}_{x.2^{\texttt{nd}}} \cdot \mathcal{S} \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \qquad \mathfrak{D} = \mathfrak{C} \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y}\end{array}}{\Gamma, x : \sigma \otimes \rho \vdash_{cf} \texttt{case}\ x\ \texttt{of}\ \langle y,\, z\rangle \to e : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho)$ by assumption, the rule *V-Context* can be inverted to learn $\langle v_1,\, v_2\rangle : \sigma \otimes \rho$. Then further, the rule *V-Pair* can be inverted to learn both $v_1 : \sigma$ and $v_2 : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$$

Each of the following judgments has now been found:

- $V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (0,0)$
- $(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$
- $\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \vdash_{cf} e : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$
- $\mathfrak{C} \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a} \geq 0$
- $Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a}$ annotates $(\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{S}$.

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a})$
$\geq \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid N \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a})$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven. Finally, the energy bound is satisfied by the following inequalities where the needed witness matrix is $M = Mov^{y}_{x.1^{\texttt{st}}} \cdot Mov^{z}_{x.2^{\texttt{nd}}} \cdot N \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \in \mathcal{T}$.

$$\begin{aligned}
&\Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho) \mid \vec{a}) \\
&= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a}) && \textit{def} \\
&\geq \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid N \cdot Mov^{x.2^{\texttt{nd}}}_{z} \cdot Mov^{x.1^{\texttt{st}}}_{y} \cdot \vec{a}) && (2) \\
&= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{def} \\
&= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{zeroed}
\end{aligned}$$

E-SumL Suppose the last rule applied for the evaluation judgment is *E-SumL*.

$$\dfrac{\text{E-S\small{UM}L}}{V, x \mapsto v \vdash \mathtt{l}(x) \Downarrow \mathtt{l}(v) \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\dfrac{\text{CF-S\small{UM}L}}{\Gamma, x : \tau \vdash_{cf} \mathtt{l}(x) : \tau \oplus \sigma \rightsquigarrow \{Mov^{x}_{\mathtt{ret.l}} \cdot Hav_{\mathtt{ret.r}}\} \mid \emptyset}$$

Because $\mathtt{l}(v) : \tau \oplus \sigma$ follows from *V-SumL* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \tau)$, the needed well-formedness judgment holds. Then because the potential energy of a variant is that of its tagged value, ignoring the other tag's energy annotations, the energy bound is also satisfied at $M = Mov^{x}_{\mathtt{ret.l}} \cdot Hav_{\mathtt{ret.r}} \in \mathcal{S}$ with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \vec{a}) = \Phi((V, x \mapsto v, \mathtt{ret} \mapsto \mathtt{l}(v)) : (\Gamma, x : \tau, \mathtt{ret} \mapsto \tau \oplus \sigma) \mid M \cdot \vec{a})$$

E-SumR Suppose the last rule applied for the evaluation judgment is *E-SumR*.

$$\dfrac{\text{E-S\small{UM}R}}{V, x \mapsto v \vdash \mathtt{r}(x) \Downarrow \mathtt{r}(v) \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\dfrac{\text{CF-S\small{UM}R}}{\Gamma, x : \sigma \vdash_{cf} \mathtt{r}(x) : \tau \oplus \sigma \rightsquigarrow \{Mov^{x}_{\mathtt{ret.r}} \cdot Hav_{\mathtt{ret.l}}\} \mid \emptyset}$$

Because $\mathtt{r}(v) : \tau \oplus \sigma$ follows from *V-SumR* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \sigma)$, the needed well-formedness judgment holds. Then because the potential energy of a variant is that of its tagged value, ignoring the other tag's energy annotations, the energy bound is also satisfied at $M = Mov^{x}_{\mathtt{ret.r}} \cdot Hav_{\mathtt{ret.l}} \in \mathcal{S}$ with the following equality:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \sigma) \mid \vec{a}) = \Phi((V, x \mapsto v, \mathtt{ret} \mapsto \mathtt{r}(v)) : (\Gamma, x : \tau, \mathtt{ret} \mapsto \tau \oplus \sigma) \mid M \cdot \vec{a})$$

E-CaseS-L Suppose the last rule applied for the evaluation judgment is *E-CaseS-L*.

$$\text{E-C\small{ASE}S-L}\quad \dfrac{V, x \mapsto \mathtt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (0,0)}{V, x \mapsto \mathtt{l}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

CF-CaseS

$$\frac{\begin{array}{cc}\Gamma, x : \sigma \oplus \rho, y : \sigma \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} & \Gamma, x : \sigma \oplus \rho, z : \rho \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \\ \mathcal{U} = \mathit{Hav}_{x.\mathtt{r}} \cdot \mathit{Mov}^{y}_{x.\mathtt{l}} \cdot \mathcal{S} \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} & \mathcal{V} = \mathit{Hav}_{x.\mathtt{l}} \cdot \mathit{Mov}^{z}_{x.\mathtt{r}} \cdot \mathcal{T} \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \\ \mathfrak{E} = \mathfrak{C} \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} & \mathfrak{F} = \mathfrak{D} \cdot \mathit{Mov}^{x.\mathtt{r}}_{y} \cdot \mathit{Hav}_{x.\mathtt{l}}\end{array}}{\Gamma, x : \sigma \oplus \rho \vdash_{cf} \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F}}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{l}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \sigma$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (0, 0)$
- $(V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$
- $\Gamma, x : \sigma \oplus \rho, y : \sigma \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$
- $\mathfrak{C} \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a} \geq 0$
- $\mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a}$ annotates $(\Gamma, x : \sigma \oplus \rho, y : \sigma)$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{S}$.

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a})$
$\geq \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid N \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a})$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven. Finally, the energy bound is satisfied by the following inequalities where the needed witness matrix is $M = \mathit{Hav}_{x.\mathtt{r}} \cdot \mathit{Mov}^{y}_{x.\mathtt{l}} \cdot N \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \in \mathcal{U}$.

$$\begin{aligned}
&\Phi((V, x \mapsto \mathtt{l}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \vec{a}) \\
&= \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a}) && \textit{def} \\
&\geq \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid N \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} \cdot \vec{a}) && (2) \\
&= \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{def} \\
&= \Phi((V, x \mapsto \mathtt{l}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{zeroed}
\end{aligned}$$

E-CaseS-R Suppose the last rule applied for the evaluation judgment is *E-CaseS-L*.

E-CaseS-R

$$\frac{V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_2 \Downarrow v \mid (0, 0)}{V, x_s \mapsto \mathtt{r}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Cases}\quad \frac{\begin{array}{cc}\Gamma, x : \sigma \oplus \rho, y : \sigma \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} & \Gamma, x : \sigma \oplus \rho, z : \rho \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \\ \mathcal{U} = \mathit{Hav}_{x.\mathtt{r}} \cdot \mathit{Mov}^{y}_{x.\mathtt{l}} \cdot \mathcal{S} \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} & \mathcal{V} = \mathit{Hav}_{x.\mathtt{l}} \cdot \mathit{Mov}^{z}_{x.\mathtt{r}} \cdot \mathcal{T} \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \\ \mathfrak{E} = \mathfrak{C} \cdot \mathit{Mov}^{x.\mathtt{l}}_{y} \cdot \mathit{Hav}_{x.\mathtt{r}} & \mathfrak{F} = \mathfrak{D} \cdot \mathit{Mov}^{x.\mathtt{r}}_{y} \cdot \mathit{Hav}_{x.\mathtt{l}}\end{array}}{\Gamma, x : \sigma \oplus \rho \vdash_{cf} \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F}}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumR* can be inverted to learn $v' : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{l}(v'), z \mapsto v' \vdash e_1 \Downarrow v \mid (0, 0)$
- $(V, x \mapsto \mathtt{l}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$
- $\Gamma, x : \sigma \oplus \rho, z : \rho \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}$
- $\mathfrak{D} \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a} \geq 0$
- $\mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a}$ annotates $(\Gamma, x : \sigma \oplus \rho, z : \rho)$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{T}$.

(1) $v : \tau$

(2) $\Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a})$
$\geq \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid N \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a})$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven. Finally, the energy bound is satisfied by the following inequalities where the needed witness matrix is $M = \mathit{Hav}_{x.\mathtt{l}} \cdot \mathit{Mov}^{z}_{x.\mathtt{r}} \cdot N \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \in \mathcal{V}$.

$$\begin{aligned}
&\Phi((V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid \vec{a}) \\
&= \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a}) && \textit{def} \\
&\geq \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid N \cdot \mathit{Mov}^{x.\mathtt{r}}_{z} \cdot \mathit{Hav}_{x.\mathtt{l}} \cdot \vec{a}) && (2) \\
&= \Phi((V, x \mapsto \mathtt{r}(v'), z \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{def} \\
&= \Phi((V, x \mapsto \mathtt{r}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \mathtt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{zeroed}
\end{aligned}$$

E-Nil Suppose the last rule applied for the evaluation judgment is *E-Nil*.

$$\textsc{E-Nil}\quad \frac{}{V \vdash [\,] \Downarrow [\,] \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Nil}\quad \frac{}{\Gamma \vdash_{cf} [\,] : L(\tau) \rightsquigarrow \{\pi_{\neg\mathtt{ret.e}} \cdot Hav_{\mathtt{ret}}\} \mid \emptyset}$$

Because $[\,] : L(\tau)$ follows from *V-Nil*, the needed well-formedness judgment holds. Finally, because the initial and remainder annotations are identical except for the empty list annotations and empty lists carry no energy regardless of annotation, the energy bound is also satisfied with the following equality at $M = \pi_{\neg\mathtt{ret.e}} \cdot Hav_{\mathtt{ret}} \in \mathcal{S}$:

$$\Phi(V : \Gamma \mid \vec{a}) = \Phi((V, \mathtt{ret} \mapsto [\,]) : (\Gamma, \mathtt{ret} : L(\tau)) \mid M \cdot \vec{a})$$

E-Cons Suppose the last rule applied for the evaluation judgment is *E-Cons*.

$$\textsc{E-Cons}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-Cons}\quad \frac{}{\Gamma, x : \tau, y : L(\tau) \vdash_{cf} x :: y : L(\tau) \rightsquigarrow \{\overset{A}{\triangleright}{}^{t}_{\mathtt{ret}} \cdot \pi_{\neg y.\mathtt{e}}\} \mid \{\pi_{y.\mathtt{e}}\}}$$

Because $v_1 :: v_2 : L(\tau)$ follows from *V-Cons* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau))$, the needed well-formedness judgment holds. Finally, the energy bound is also satisfied with the following inequalities where $M = \overset{A}{\triangleright}{}^{y}_{\mathtt{ret}} \cdot \pi_{\neg y.\mathtt{e}} \in \mathcal{S}$:

$$\begin{aligned}
&\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid \vec{a}) \\
\geq\ &\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid \pi_{\neg y.\mathtt{e}} \cdot \vec{a}) && \textit{Lemma } 3.4.5 \\
=\ &\Phi((V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto v_1 :: v_2) : (\Gamma, x : \tau, y : L(\tau), \mathtt{ret} : L(\tau)) \mid M \cdot \vec{a}) && \textit{Lemma } 6.4.1
\end{aligned}$$

The first inequality makes use of the assumption that $\pi_{y.\mathtt{e}} \cdot \vec{a} \geq 0$, and the last equality makes use of the fact that A is invertible.

E-CaseL-Nil Suppose the last rule applied for the evaluation judgment is *E-CaseL-Nil*.

$$\textsc{E-CaseL-Nil}\quad \frac{V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (0,0)}{V, x \mapsto [\,] \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (0,0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{CF-CaseL}\quad \frac{\begin{array}{c}
\Gamma, x : L(\sigma) \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \\
\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \qquad \mathcal{U} = \pi_{\neg x.e} \cdot Hav_x \cdot \mathcal{S} \cdot \pi_{\neg x.e} \cdot Hav_x \\
\mathcal{V} = \pi_{\neg y} \cdot \overset{A}{\triangleright}{}^{z}_{x} \cdot \mathcal{T} \cdot \overset{A}{\triangleleft}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{E} = \mathfrak{C} \cdot \pi_{\neg x.e} \cdot Hav_x \\
\mathfrak{F} = \mathfrak{D} \cdot \overset{A}{\triangleleft}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{G} = \pi_y \cdot \overset{A}{\triangleright}{}^{z}_{x} \cdot \mathcal{T} \cdot \overset{A}{\triangleleft}{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \qquad \mathfrak{H} = \{\pi_{x.\mathtt{e}}\}
\end{array}}{\Gamma, x : L(\sigma) \vdash_{cf} \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F} \cup \mathfrak{G} \cup \mathfrak{H}}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (0, 0)$
- $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$
- $\Gamma, x : L(\sigma) \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C}$
- $\mathfrak{C} \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a} \geq 0$
- $\pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a}$ annotates $(\Gamma, x : L(\sigma))$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{S}$.

(1) $v : \tau$
(2) $\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a})$
$\geq \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid N \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a})$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven. Finally, because empty lists carry no potential energy regardless of annotation, the energy bound follows at $M = \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot N \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \in \mathcal{U}$ using the following inequalities:

$$\begin{aligned}
&\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \vec{a}) && \\
&= \Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a}) && \mathit{def} \\
&\geq \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid N \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \vec{a}) && (2) \\
&= \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid M \cdot \vec{a}) && \mathit{def}
\end{aligned}$$

E-CaseL-Cons Suppose the last rule applied for the evaluation judgment is *E-CaseL-Cons*.

E-CaseL-Cons
$$\frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (0, 0)}{V, x \mapsto v_1 :: v_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

CF-CaseL
$$\frac{\begin{array}{c}\Gamma, x : L(\sigma) \vdash_{cf} e_1 : \tau \rightsquigarrow \mathcal{S} \mid \mathfrak{C} \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D} \qquad \mathcal{U} = \pi_{\neg x.e} \cdot \mathit{Hav}_x \cdot \mathcal{S} \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \\ \mathcal{V} = \pi_{\neg y} \cdot \overset{A}{\triangleright}\substack{z \\ x} \cdot \mathcal{T} \cdot \overset{A}{\triangleleft}\substack{x \\ z} \cdot \pi_{\neg\{y, x.e\}} \qquad \mathfrak{E} = \mathfrak{C} \cdot \pi_{\neg x.e} \cdot \mathit{Hav}_x \\ \mathfrak{F} = \mathfrak{D} \cdot \overset{A}{\triangleleft}\substack{x \\ z} \cdot \pi_{\neg\{y, x.e\}} \qquad \mathfrak{G} = \pi_y \cdot \overset{A}{\triangleright}\substack{z \\ x} \cdot \mathcal{T} \cdot \overset{A}{\triangleleft}\substack{x \\ z} \cdot \pi_{\neg\{y, x.e\}} \qquad \mathfrak{H} = \{\pi_{x.\mathtt{e}}\}\end{array}}{\Gamma, x : L(\sigma) \vdash_{cf} \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \rightsquigarrow \mathcal{U} \cup \mathcal{V} \mid \mathfrak{E} \cup \mathfrak{F} \cup \mathfrak{G} \cup \mathfrak{H}}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 : L(\sigma)$. Then further, the rule *V-Cons* can be inverted to learn both $v_1 : \sigma$ and $v_2 : L(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$$

Each of the following judgments has now been found:

- $V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (0, 0)$
- $(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$
- $\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \vdash_{cf} e_2 : \tau \rightsquigarrow \mathcal{T} \mid \mathfrak{D}$
- $\mathfrak{D} \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a} \geq 0$
- $\overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a}$ annotates $(\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$

With these judgments, the inductive hypothesis can be applied to learn the following for some $N \in \mathcal{T}$.

(1) $v : \tau$

(2) $\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a})$
$\geq \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid N \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a})$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's energy bound remains to be proven.

Now let $M = \pi_{\neg y} \cdot \overset{A}{\rhd}\,{}^{z}_{x} \cdot N \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \in \mathcal{V}$. Observe that all rules that manipulate list element annotations inductively ensure that the element annotations are zero, so the same must be true of the element annotations of x in the vector $M \cdot \vec{a}$. Because these annotations are zero, they can be ignored during shifting and unshifting. Finally, because shifting conserves the potential energy of a list (Lemma 6.4.1), the energy bound follows using the following inequalities:

$$\begin{aligned}
& \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid \vec{a}) \\
\geq\ & \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid \pi_{\neg x.\mathtt{e}} \cdot \vec{a}) && \textit{Lemma } 3.4.5 \\
=\ & \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a}) && \textit{Lemma } 6.4.1 \\
\geq\ & \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid N \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a}) && (2) \\
=\ & \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\rhd}\,{}^{z}_{x} \cdot N \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a}) && \textit{Lemma } 6.4.1 \\
=\ & \Phi((V, x \mapsto v_1 :: v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid M \cdot \vec{a}) && \textit{Lemma } 3.4.5
\end{aligned}$$

The first inequality makes use of the assumption that $\pi_{x.\mathtt{e}} \cdot \vec{a} \geq 0$, the penultimate equality makes use of the fact that A is invertible, and the last equality makes use of the assumption that $\pi_{y} \cdot \overset{A}{\rhd}\,{}^{z}_{x} \cdot N \cdot \overset{A}{\lhd}\,{}^{x}_{z} \cdot \pi_{\neg\{y,x.e\}} \cdot \vec{a} \geq 0$.

□

8.6 Costful Integration

This section explains more formally how to make use of this chapter's cost-free type system in the costful setting. All the changes needed to integrate the two systems concern functions.

I-FUN

$$\frac{\begin{array}{c}\Gamma \vdash_{cf} \texttt{fun}\ f\ x\ =\ [\texttt{tick}\{0\}/\texttt{tick}\{r\}]e : \tau \overset{\vec{c}|\vec{d}|M}{\to} \sigma \rightsquigarrow \{I\} \mid \emptyset \\ \Gamma, x:\tau, f:\tau \overset{\vec{c}|\vec{d}|M}{\to} \sigma \mid 0\cdot\vec{a}, [x/\texttt{arg}]\vec{c} \vdash e : \sigma \mid 0\cdot\vec{a}, [x/\texttt{arg}]\vec{d}\end{array}}{\Gamma \mid \vec{a}, \vec{b} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \overset{\vec{c}|\vec{d}|M}{\to} \sigma \mid \vec{a}, \vec{b}}$$

I-APP

$$\frac{\vec{a} \geq 0 \qquad \vec{d} \geq 0}{\Gamma, x:\tau, f:\tau \overset{\vec{b}|\vec{c}|M}{\to} \sigma \mid \Upsilon_x^{x,\texttt{arg}}(\vec{a}, \vec{b}) + \vec{d} \vdash f\ x : \sigma \mid \Upsilon_x^{x,\texttt{arg}}(\vec{a}, \vec{c}) + Mov_x^{\texttt{arg}} \cdot M \cdot Mov_{\texttt{arg}}^x \cdot \vec{d}}$$

V-FUNI

$$\frac{V : \Gamma \qquad \Gamma \mid \vec{a} \vdash \texttt{fun}\ f\ x\ =\ e : \tau \overset{\vec{a}|\vec{b}|M}{\to} \sigma \mid \vec{b}}{\texttt{C}(V;\ f,\ x.\,e) : \tau \overset{\vec{a}|\vec{b}|M}{\to} \sigma}$$

Figure 8.8: Integrated costful function rules

Firstly, the types of functions themselves must be made into an amalgam of both costful and cost-free types. Such a function type looks like the following, where the argument and remainder annotation vectors $\vec{a}$ and $\vec{b}$ play the same role that they do in the costful system, and the cost-free matrix M plays the same role it does in the cost-free system.

$$\tau \overset{\vec{a}|\vec{b}|M}{\to} \sigma$$

These new function types are then governed by the rules in Figure 8.8, which replace previous function and application rules. The cost-free typing judgments in these rules should also be interpreted to ignore the new annotation vectors on functions. The idea of these new rules is to use both the cosftul and cost-free system to type functions, where the costful type may depend (recursively) upon the cost-free type. Both the rules *I-Fun* and *V-FunI* accomplish this goal just by ensuring that functions are typed under both systems. (The rule *I-Fun* additionally imposes the cost-free cost model for cost-free typing via substitution.) It is the application rule *I-App* where the real magic happens, which I explain in the following paragraph.

The rule *I-App* accomplishes the main goal of cost-free typing by reallocating excess potential energy. In the rule, that excess is given by the nonnegative annotation vectors $\vec{a}$ and $\vec{d}$. The vector $\vec{a}$ plays the same role that it does in earlier chapters: this vector indicates excess potential energy that should be left with the input and not taken in by the function. In contrast, the excess energy indicated by $\vec{d}$ is passed through the function. This behaviour is shown in the type rule by transforming $\vec{d}$ according to the function's cost-free matrix M and incorporating the result in the remainder via adding it to $\Upsilon_x^{x,\texttt{arg}}(\vec{a}, \vec{c})$. Apart from this new option to transform $\vec{d}$'s excess energy, the typing rule is exactly the same as in earlier chapters.

To understand why these rules are sound, one should consider Corollary 8.6.0.1, which is a corollary to Theorem 8.5.1. This corollary just specializes the theorem to functions. As before,

$q = 0$ in the corollary statement if the evaluation is terminating, and $q = \infty$ if the evaluation does not terminate (Lemma 2.4.2).

Corollary 8.6.0.1 (cost-free function soundness). *In a cost-free cost model, if*

- $V \vdash f\ x \Downarrow v \mid (0, q)$ *(applying f to x yields v in context V)*
- $V : \Gamma$ *(V is well-formed w.r.t. Γ)*
- $\Gamma(f) = \tau \overset{M}{\to} \sigma$ *and* $\Gamma(x) = \tau$ *(Γ gives f and x these cost-free types)*
- $\vec{a} \geq 0$ *annotates x*

then

$$\Phi(V(x) : \tau \mid \vec{a}) + q \geq \Phi(v : \sigma \mid Mov_x^{\texttt{arg}} \cdot M \cdot Mov_{\texttt{arg}}^x \cdot \vec{a})$$

the argument energy bounds the remainder energy when the annotations are transformed via M

With this corollary in hand, one can then show the soundness of the full costful system that integrates cost-freedom via Theorem 8.6.1.

Theorem 8.6.1 (integrated costful soundness). *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid \vec{a} \vdash e : \tau \mid \vec{b}$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid \vec{a}) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid \vec{a}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid \vec{b}) + p$ *(diff. bounds net)*

Proof. This property is proven almost exactly as in Theorems 5.4.1 and 6.4.3. There are only two new typing rules to consider: *I-Fun* and *I-App*. There is also only one new well-formedness rule, *V-FunI*, which only holds when both *V-Fun* and *V-FunCf* would hold in their separate settings.

I-Fun If *E-Fun* concludes the evaluation judgment and *I-Fun* concludes the typing judgment, then the case follows just as in Theorems 5.4.1 and 6.4.3. The new premiss may simply be ignored, and functions still carry no energy.

I-App If *E-App* concludes the evaluation judgment and *I-App* concludes the typing judgment, then the peak and net cost bounds hold as follows:

The peak cost bound follows from Theorem 6.4.3 by monotonicity (Lemma 3.4.5) since $\vec{d} \geq 0$. New premisses in the integrated costful system can simply ignored.

To obtain the net cost bound, first note that if $V \vdash e \Downarrow v \mid (p, q)$ holds in a costful setting, then setting the tick parameters to zero allows the derivation of $V \vdash e \Downarrow v \mid (0, q')$ in a cost-free cost model. Then there are two cases to consider depending upon whether nontermination (*E-Nont*) is used in the evaluation judgment.

If *E-Nont* is used, then $q = q' = \infty$ by Lemma 2.4.2. In this case, the net cost bound holds because ∞ is greater than or equal to anything.

Alternatively if *E-Nont* is not used, then $q' = 0$. Finally, the following inequalities hold, where $W = (V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e))$ and $\Delta = \Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}|M}{\to} \sigma$.

$$\begin{aligned}
&\Phi(W : \Delta \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a},\vec{b}) + \vec{d}) + q \\
&= \Phi(W : \Delta \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a},\vec{b})) + q + \Phi(W : \Delta \mid \vec{d}) && \textit{Lemma } 3.4.6 \\
&\geq \Phi((W, \mathtt{ret} \mapsto v) : (\Delta, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a},\vec{c})) + p + \Phi(W : \Delta \mid \vec{d}) && \textit{Theorem } 6.4.3 \\
&\geq \Phi((W, \mathtt{ret} \mapsto v) : (\Delta, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a},\vec{c})) + p \\
&\quad + \Phi((W, \mathtt{ret} \mapsto v) : (\Delta, \mathtt{ret} : \sigma) \mid \mathit{Mov}_x^{\mathtt{arg}} \cdot M \cdot \mathit{Mov}_{\mathtt{arg}}^x \cdot \vec{d}) && \textit{Corollary } 8.6.0.1 \\
&= \Phi((W, \mathtt{ret} \mapsto v) : (\Delta, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(\vec{a},\vec{c}) + \mathit{Mov}_x^{\mathtt{arg}} \cdot M \cdot \mathit{Mov}_{\mathtt{arg}}^x \cdot \vec{d}) + p && \textit{Lemma } 3.4.6
\end{aligned}$$

□

8.7 Automation

This section first explains how to automate type checking in the matrix-based cost-free type system, and then it explains how to automate type inference. In particular, this section shows how type checking is always efficiently reducible to simple linear arithmetic and how type inference can *usually* be reduced to linear programming. Obstacles to further automation are discussed in Section 8.10.

Checking Types can be checked directly in this type system. The un-annotated base types can be checked via standard techniques, and then all that is left is to confirm the annotations of functions. However, once each function is annotated with a concrete matrix, confirming the correctness of those matrices amounts to checking the three matrix inequalities over the sets $\mathcal{S}$ and $\mathfrak{C}$ in the rule *CF-Fun*, which is mostly a matter of linear arithmetic. In detail:

1. All the maps of $\mathcal{S}$ and $\mathfrak{C}$ are generated through the typing rules' matrix multiplications.
2. Pointwise inequalities between matrices are set up according to *CF-Fun*
3. Inequalities over havocked elements $*$ from Hav_x are filtered out, as they are trivially met.
4. The remaining inequalities are checked.

Thus, the efficiency of type checking depends only on how quickly the matrices of the sets $\mathcal{S}$ and $\mathfrak{C}$ can be generated, and how quickly those equalities can be checked.

Let the longest execution path through a function's body be comprised of n subexpressions, and let there be up to k lists present in scope at a time. Note the following asymptotic bounds:

- $\Theta(n)$ matrix multiplications are used to obtain each $M \in \mathcal{S} \cup \mathfrak{C}$
- the dimension of all matrices involved is $O(k \cdot d)$
- $|\mathcal{S}| + |\mathfrak{C}| = O(2^n)$

These bounds can be combined to provide a bound on how quickly the matrix inequalities of *CF-Fun* can be generated. Letting $\omega \in [2,3]$ be the exponent associate with matrix multiplication, that bound is $O(2^n \cdot n \cdot (k \cdot d)^\omega)$. Then there are only $O((k \cdot d)^2)$ matrix elements to check inequalities over, so $O(2^n \cdot n \cdot (k \cdot d)^\omega)$ gives an upper bound on the time complexity of the entire process.

Note that the only super-polynomial part of this complexity comes from the number of paths through the function body being $O(2^n)$ in the worst case. In practice, people do not usually write code in a way that hits this upper bound. In particular, if no let-bound expression involves branching, then the number of paths is $O(n)$, yielding a time complexity of $O(n^2 \cdot (k \cdot d)^\omega)$.

Inference The only inherent difference between type inference and checking is that the matrices of $\mathcal{S}$ and $\mathfrak{C}$ now include unknown variables from the function annotation being inferred. As a result, the matrix inequalities that must hold for the rule *M-Fun* are no longer simple arithmetic checks; instead they require solving systems of inequalities over polynomials. However, to minimize the number of unknowns involved, one should also impose the following ordering on type inference:

1. Topologically sort functions such that $f \leq g$ if f is called in the body of g.
2. Infer types in an ascending order.

By inferring types in this order, the type of a function f is only inferred after all the functions f depends on have had their types inferred. Thus, the polynomial inequalities only involve unknowns from the matrix annotating the type of f itself (for non-mutually-recursive f).

Such inequalities can be solved optimally in general using quadratically-constrained quadratic programming, which is known to be NP-hard. However, it is common that such polynomials are actually linear, allowing an optimal solution to be found in polynomial time via linear programming. For example, consider the inequalities from the example typing `half` in Example 8.4.2—a linear program that appropriately maximizes the higher-degree entries of M yields the exact desired matrix for `half`. In such a case, the effect on time complexity is that the $O((k \cdot d)^2)$ inequality checks can be replaced with $O((k \cdot d)^r)$, where $r \geq \omega$ is the exponent associated with linear program solving.

One can tell when linear programming is insufficient simply by checking the generated constraint set. In such a case, one could either default to nonlinear constraint solving or go back to the existing cost-free approach, depending on the tradeoffs one wishes to make. However, many cases are amenable to linear programming in practice, as validated in Section 8.8.2.

To see why linear programming commonly suffices, consider what needs to happen for nonlinear constraints to arise. Nonlinear terms can only arise in the course of multiplication between two (possibly identical) matrices M and N both containing unknown variables. Such multiplication can only occur if, along a single path through a function's body, two function calls are made to functions whose types have not yet been inferred. This circumstance can occur when calling higher-order functions or when inferring the type of recursive functions with non-linear recursion schemes. Much code, including every code example given so far in this work, uses a linear recursion scheme, and thus admits efficient type inference. (See Section 8.10 for higher-order function discussion.)

Further, while nonlinear recursion is *necessary* for nonlinear constraints in a first-order setting, it is not *sufficient*. If $M = M' \oplus I$ and $N = I \oplus N'$, then $M \cdot N = M' \oplus N'$, which has only linear entries if M and N do. This pattern arises if two recursive calls are made to independent arguments. Thus, nonlinear constraints require something like the chaining of function applications to define the very function being applied. In other words, the typical example of the problem case is recursively defining the function `f` by using `f (f x)` somewhere in its body. This pattern arises in the Ackermann function, but this pattern almost never arises in real code.

8.8 Experiments

This section presents experiment with an implementation of the new matrix-based cost-free type inference algorithm. First, the algorithm's efficiency is compared to the existing approach on parameterized synthetic code, which demonstrates the relative exponential improvement in how the new algorithm scales. Then the new algorithm is run on a collection of realistic list functions to obtain an absolute sense of its performance. The latter experiment also validates that the new algorithm usually deals with only linear constraints. All experiments were implemented in OCaml 4.12.0, run on a Mac with a 2.3 GHz Dual-Core Intel Core i5 processor, and use the Gurobi version 9.5.1 solver [70].

8.8.1 Efficiency Comparison

To compare the matrix-based cost-free type-inference technique with the existing approach [77, 80] empirically, the efficiency of implementations of both techniques—the existing cost-free type-inference algorithm and the new algorithm from Section 8.7—were measured. (A theoretical comparison of the two algorithms can be found in in Section 8.9.)

These algorithms were run on synthetic code to measure their efficiency. This code exhibits relevant patterns that both algorithms are capable of analyzing. Because these algorithms' runtimes are highly dependent on the code patterns used, the code is parameterized based on two salient features: how many calls c are made to a function after the function is defined, and the length ℓ of the chain of a function's dependencies on other functions. Both c and ℓ not only measure features salient for efficiency of the algorithms, but also naturally scale with the use of helper functions in real code. Experiments were also performed with varying degrees d of polynomial potential energy, which is another performance-relevant parameter of interest.

The specific code pattern generated for the experiments is exemplified in Figure 8.9, where ℓ linearly recursive functions are defined, each of which calls the previously defined function c times. This code is functionally equivalent to the identity function on lists.

This experiment was designed to answer the following questions:

```
    let g2 =
        let g1 = fun f0 x0 = x0 in
        fun f1 x1 = case x1 of
            | [] -> []
            | h1::t1 -> h1::(g1 (g1 (g1 (f1 t1))))
    in fun f2 x2 = case x2 of
        | [] -> []
        | h2::t2 -> h2::(g2 (g2 (g2 (f2 x2))))
```

Figure 8.9: Synthetic code pattern example, $c = 3, \ell = 2$

Q1. How well does the existing algorithm scale as a function of c, the number of calls to a helper function; ℓ, the length of the dependency chain; and d, the degree?

Q2. How well does the new algorithm scale as a function of c, ℓ, and d?

Q3. Does the overhead of the new algorithm make it less efficient than the existing algorithm on simple programs without challenging patterns; if so, to what extent?

The experimental findings can be summarized as follows:

A1. The existing algorithm scales poorly both in terms of the number constraints and time. Its efficiency seems to scale exponentially in each of c, ℓ, and d. On the largest cases in the testing range, this algorithm times out after days.

A2. In comparison, the new algorithm scales well. Its efficiency does not seem to scale exponentially in any of c, ℓ, and d. The new algorithm always generates fewer constraints than the existing algorithm and in many cases is multiple orders of magnitude more efficient in terms of both time taken and constraints generated.

A3. Despite generating fewer constraints than the existing algorithm, the new algorithm takes more time to solve those constraints for the smallest cases in the testing range. However, the difference is never more than a fraction of a second.

To keep the experiments fair, a fresh implementation of the existing approach was built. There is an available implementation of AARA called Resource Aware ML (RaML) [81], which uses the existing approach. However, RaML implements many special features, including multivariate resource functions,[8] which would unfairly slow down RaML's analysis. Simultaneously, RaML includes many optimizations to minimize memory usage and constraint generation that the prototype does not use, unfairly *advantaging* RaML's analysis. Finally, RaML does not use remainder contexts, whereas both the implementations used in the experiment do. Thus, a fresh implementation gives a more direct comparison.

Each of the implementations was run over the same code representations on the same com-

[8]Multivariate resource functions express potential energy not merely as sums of basic functions, but also as products. This concept is discussed in Chapter 7.

puter with the same interface to the same linear-program solver, creating a level testing ground. These experiments measured the number of linear constraints generated, the time taken to generate the constraints, and the time taken to solve the constraints for $1 \leq d \leq 6$ and $0 \leq c, \ell \leq 5$.

The complete experimental data can be found in Section 8.12. Graphs of some selected features can be found in Figure 8.10. Each surface plot graph shows the scaling effects of two of d, c, and ℓ while the third parameter is fixed at 3. Note that the data points occur at mesh vertices, and that the vertical axis of each graph is on a log scale.

Even in this small domain, the experimental results show that the new algorithm scales significantly better. While both algorithms had similar runtimes and generated similar numbers of constraints in the smallest cases, the exponential growth of the existing algorithm took over quickly. The implementation of the existing algorithm eventually timed out after taking days on the largest cases, where $d + c + \ell \geq 15$, while the new algorithm could complete the analysis of each of those cases in well under half a second. One of the largest tests that both of the implementations completed, with $d = \ell = 5$ and $c = 4$, saw the new algorithm take 0.13s to create 631 constraints, which were solved in 0.19s—whereas the existing algorithm took 36s to generate 27 million constraints, which were solved in about 150000 seconds (43 hours).

8.8.2 Realistic Examples

To give an absolute sense of the kind of inference performance that could be expected on real code, as well as to validate that nonlinear constraints do not usually arise, the inference algorithm was run on various common list functions. This set of functions includes some higher-order functions, `map` and `filter`, where the inference modification outlined in Section 8.10 was taken.

This experiment was designed to address the following questions about the inference algorithm for the matrix-based cost-free type system:

Q1. How often does the algorithm avoid nonlinear constraints on realistic code?

Q2. How quickly can the algorithm handle realistic code?

Q3. How often does the map inferred by the algorithm optimally reallocate potential from input to output?

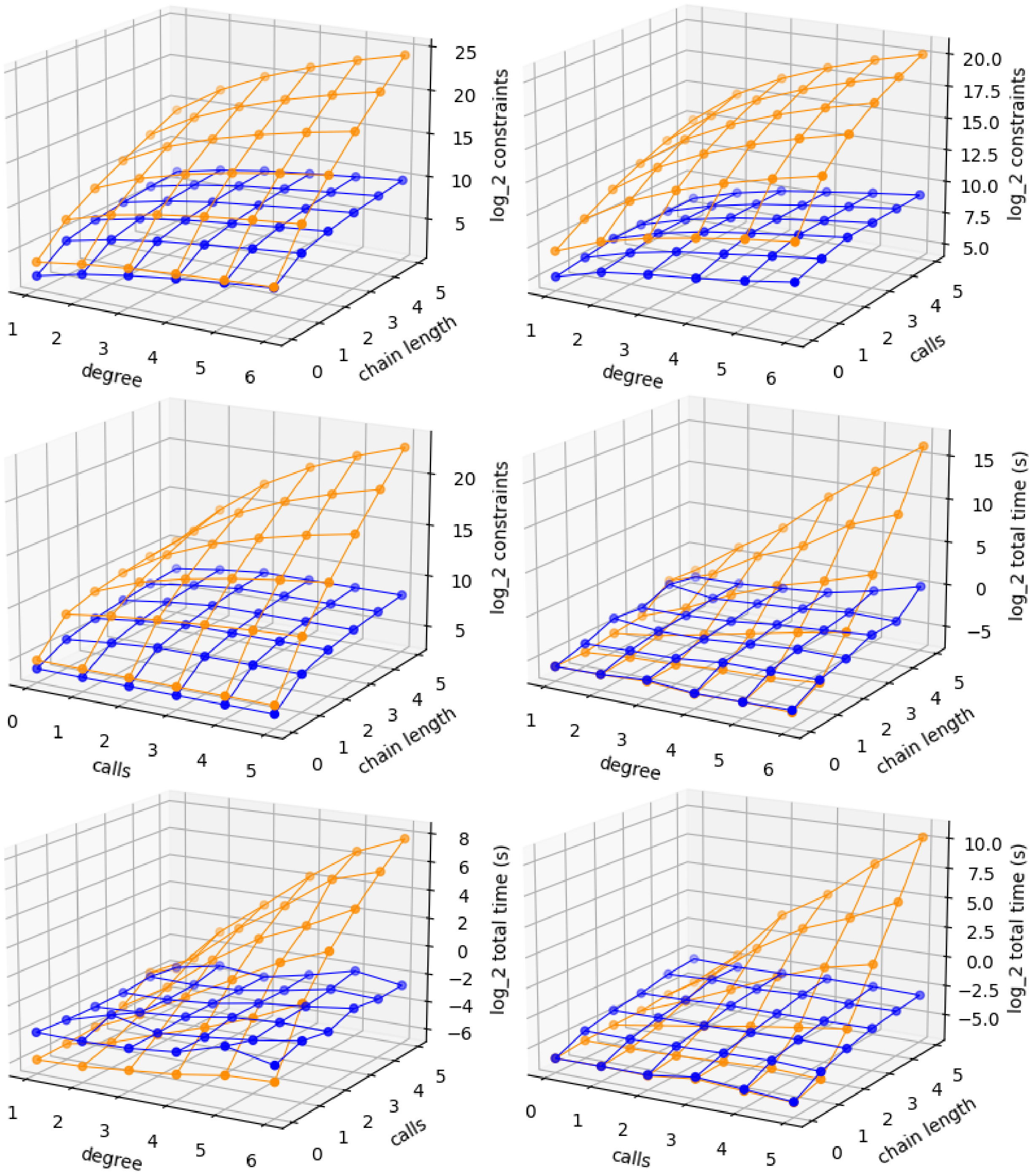
The findings can be summarized as follows:

A1. None of the code examples generated nonlinear constraints.

A2. Even while working with a high degree of potential (10), most code could be analyzed in under a second.

A3. Non-optimal reallocation usually occurs in code that manipulates multiple lists.

The data from the experiments can be found in Table 8.1. The table data includes the time spent to generate constraints from the source code, the time spent to solve those constraints,



Key: **orange** = existing approach, **blue** = new approach

Figure 8.10: Surface plots of data with one of d, c, ℓ fixed at 3. (Lower is better.)

function	constr secs	solve secs	total secs	constr count	linear	infer success	optimal realloc.
cons	0.011	0.032	0.043	122	✓	✓	✓
uncons	0.019	0.029	0.048	122	✓	✓	✓
map	0.036	0.068	0.104	158	✓	✓	✓
filter	0.078	0.049	0.127	278	✓	✓	✓
zip	0.338	0.082	0.420	359	✓	✓	✓
unzip	0.266	0.081	0.347	255	✓	✓	✓
insert	0.096	0.034	0.130	278	✓	✓	✓
remove	0.050	0.040	0.090	267	✓	✓	✓
insertion sort	0.117	0.070	0.186	544	✓	✓	✓
split	0.047	0.097	0.144	266	✓	✓	✓
merge	0.750	0.091	0.841	1009	✓	✓	X
merge sort	2.307	0.329	2.636	1703	✓	✓	X

Table 8.1: Performance on list functions at max polynomial degree 10

the total time, the total number of constraints generated, whether inference successfully yields a type, and whether the inferred linear map optimally reallocates potential.

The results of this experiment show some benefits and drawbacks of this chapter's approach to cost-free type inference. First, the matrix-based approach is quite efficient, even when working with high degrees of polynomial resource functions. Second, while quadratic constraints could theoretically arise out of the matrix-based approach, this experiment supports the claim that the mitigating factors discussed in Section 8.7 really do keep constraints linear in practice. Even in the case of *merge sort*, which does not use a linear recursion scheme, only linear constraints arose because the recursive calls are not nested in the pathological way described in Section 8.7. Finally, however, the analysis of *merge* (and thus also *merge sort*) shows that working with multiple lists can be a source of non-optimal reallocation of potential energy, which results in looser cost bounds. Specifically, the inferred cost-free matrix in these cases is only able to reallocate constant amounts of energy, losing all higher-degree potential energy. Section 8.9 dives deeper into why this reallocation fails. Note, however, that the maps for *zip* and *unzip* both optimally reallocate potential despite the fact that they each manipulate multiple lists.

While this experiment was only run on library code rather than large complicated algorithms, the results are still supportive of the efficiency gains that could be expected from using the matrix-based approach to cost-freedom in a fully-automated setting. Many real-world applications make heavy use of such library functions [50], so even just being able to handle these functions efficiently should help to cut short the expensive cascade of retypings that would otherwise be performed by the preexisting cost-free approach.

8.9 Further Comparison

To further compare the matrix-based cost-free type inference algorithm with the existing algorithm, this section examines the theoretical behaviour of both in terms of asymptotic scaling and cost-bound tightness.

Asymptotics Here bounds are given on the number of linear constraints generated in terms of d, c, and ℓ for both the existing inference algorithm and the new algorithm. These bounds can be compared to the empirical data obtained in Section 8.8. Theoretical bounds on runtimes can be obtained from these bounds on constraint counts by composing the bounds with the complexity of linear program solving. As the choice of linear program solver is not essential to the type system, runtime is not further considered in this section.

To analyze the existing algorithm, note the following recurrence for $T(d, c, \ell)$ which bounds the worst-case number of constraints that the algorithm generates:

$$\begin{aligned}
&T(d, c, 0) = O(d) \\
&T(1, c, \ell) = c \cdot T(1, c, \ell - 1) + O(c) \\
&T(d, c, \ell) = c \cdot T(d, c, \ell - 1) + T(d - 1, c, \ell) + O(c \cdot d)
\end{aligned}$$

This recurrence arises because:

1. If $\ell = 0$ the code generated is simply the non-recursive identity function. The number of constraints generated is therefore proportional the size of the annotation vector $O(d)$.
2. If $d = 1$ then the algorithm needs to (i) find specialized types for each of the c calls to helper functions (with one less ℓ), as well as (ii) type its own body. In typing the body, in the worst case, the algorithm generates constraints proportional to the body length $O(c) = O(c \cdot d)$.
3. In larger cases, the algorithm needs to do all that was discussed above, in addition to retyping the function body with one less degree of potential.

While I know no closed form for this recurrence, one can count the number of occurrences of $T(d - i, c, \ell - j)$ as $\binom{i+j}{i} \cdot c^j \leq \binom{i+j}{i} \cdot c^\ell$ and overapproximate the constraints generated per occurrence as $O(c \cdot d)$ (assuming $c \geq 1$). Then $T(d, c, \ell) \leq dc^{\ell+1} \sum_{i=0}^{d-1} \sum_{j=0}^{\ell} \binom{i+j}{i} = d \cdot c^{\ell+1} \cdot (\binom{d+\ell+1}{d} - 1) = O(d \cdot c^{\ell+1} \cdot \binom{d+\ell+1}{d})$ gives an upper bound on the number of constraints generated. While this is only an upper bound, its poor scaling in all three considered parameters c, d, ℓ is qualitatively consistent with the experimental findings from Section 8.8.

Analyzing the new algorithm's constraint generation on the same patterns is more straightforward. Because each function is analyzed only once, the worst-case constraints generated are proportional to the number of functions to analyze $O(\ell)$. In each function, the number of non-negativity constraints is at most proportional to the number of calls $O(c)$ and the size of the symbolic matrix $O(d^2)$. Thus, in the worst-case, the number of constraints generated is $O(d^2 \cdot c \cdot \ell)$.

It is also relevant to consider the behavior as c and ℓ get *small* to compare performance on less extreme cases. Such a comparison is made here by analyzing the case where $c = \ell = 0$, which corresponds to a linearly recursive function calling no helper functions. In such a case, the two algorithms both generate $\Theta(d^2)$ constraints. For the existing algorithm, the constraint count is $\Theta(d^2)$ because the constraints are generated in proportion to the size of the annotation vector being constrained, which is $\Theta(d)$ in a single typing pass, and $\Theta(d)$ passes are made. For the new algorithm, the constraint count is $\Theta(d^2)$ because the constraints generated are proportional to the size of the matrix being constrained, which is $\Theta(d^2)$.

Bound Tightness Ideally, a cost-free function type would tightly express the cost of 0 between its argument and return types, because no energy should be spent. However, AARA cannot always perfectly reallocate potential energy across different data structures, so some energy might be lost instead. The less potential energy that is lost, the tighter the cost bound that AARA can infer.

The existing approach for inferring cost-free types (when it succeeds) yields the tightest bounds that AARA can get. This optimality occurs due to brute force: the existing approach infers a new cost-free type wherever such a type could better specialize to the present potential. Thus, as long as the approach's key assumption is met and inference succeeds, the type found allocates potential as losslessly as AARA types can represent.

In contrast, the new matrix-based approach may yield looser cost bounds. While this approach does well for code that manipulates only one list, code that manipulates multiple lists may introduce more loss (as in Table 8.1). This loss occurs because it is easy to use multiple lists to make certain type annotations dependent upon the minimum of the annotations across the lists, but minimum is not a linear function. The following projection function `proj` provides an example of such loss because that function would be best typed to reallocate the most energy possible to the return with the type $\mathbb{B} \otimes L^a(\tau) \otimes L^b(\tau) \to L^{min(a,b)}(\tau) \sim \mathbb{B} \otimes L^0(\tau) \otimes L^0(\tau)$. However, the best possible linear transformation is the trivial type $\mathbb{B} \otimes L^a(\tau) \otimes L^b(\tau) \to L^0(\tau) \sim \mathbb{B} \otimes L^a(\tau) \otimes L^b(\tau)$, where no energy is meaningfully reallocated, because the constant zero function is the best linear function under-approximating the minimum. I hope such nonlinear functions can be addressed in future work.

```
fun proj (b, lst1, lst2) = if b then lst1 else lst2
```

8.10 Limitations

To keep cost-free types inferable via linear programming, this chapter's type system lacks certain features. Some of these have already been discussed in other sections, such as nonlinear annotation transformations. However, other unsupported features have not yet been fully discussed. Such features include list elements, trees, full sharing, and higher-order functions. This section discusses the extent of these remaining limitations, as well as some possible routes for generalizing beyond them.

List Elements and Trees This chapter's type system forces list element annotations to be zero, as described in Section 8.4.3. The reason for forcing such annotations to be zero is that the type system cannot support nonlinear annotation transformations. If list elements could carry energy, then properly formalizing unshifting would require minima. Specifically, adding an element of type $L^a(\mathbb{B})$ to the front of a list of type $L^0(L^b(\mathbb{B}))$ would have the resulting list best typed as $L^0(L^{\min(a,b)}(\mathbb{B}))$. To avoid this problem, the system forces such element annotations to be zero. This restriction is not actually much of a detriment, as it is difficult to have resource-polymorphic recursion affect such annotations—costful typing usually needs no help from cost-free types to properly handle list element annotations.

Trees have the same problem as lists, except the issue plagues *all* of their annotations. Combining two subtrees typed using linear energy as $T^a(\mathbb{B})$ and $T^b(\mathbb{B})$ would have the resulting tree best typed as $T^{\min(a,b)}(\mathbb{B})$. Resolving this issue in the same way as lists would have trees only annotated by zeros. This solution is valid but trivial, so this chapter simply ignores trees instead.

Higher-Order Functions During type inference, higher-order function arguments have both an unknown type annotation and an unknown body. As a result, it is unclear what annotation is appropriate to infer. However, this problem is not new in the setting of cost analysis. The cost of running a higher-order function is usually dependent on the cost of the function it takes as input, and therefore would require a symbolic cost bound parameterized by the input's cost behaviour. AARA cannot currently express such a bound.

To deal with this problem, previous AARA literature treats higher-order functions with unknown arguments as if their argument functions have zero cost. This treatment has the effect of only reporting the cost behaviour of the higher-order function itself. Then, when given a concretely-typed argument function, AARA macro-expands and retype the higher-order function to take the known cost behaviour into account. This chapter's system can handle higher-order functions in the same way without raising any additional conceptual issues. This approach is taken in this chapter's experiments to obtain the results of Section 8.8.2.

Full Sharing A variable's energy must be split between all its uses. Thus, if costful AARA would annotate x with $\vec{a}$, then $\vec{b} \geq 0$ could go to one use and $\vec{c} \geq 0$ to another, where $\vec{a} = \vec{b} + \vec{c}$. This splitting is called "sharing."

While such sharing constraints are linear, the corresponding *map* acting on $\vec{a}$ introduces nonlinearity. Such a sharing map looks like $p \cdot Mov_y^x + (1-p) \cdot Mov_z^x$, where y and z stand in for two uses of x and $p \in [0,1]$. This map involves the unknown p, which means that composing with other matrices involving unknowns (like those for recursive calls) can result in nonlinear terms.

To avoid this problem, this chapter's type system codifies the essentially-arbitrary choice that all potential energy is allocated to a variable's first use. This choice is sufficient for typing many functions of interest. In addition, remainder contexts can recover unused potential in certain circumstances, regaining some (but not all) of the benefits of full sharing.

Nonlinear Constraints Given how often nonlinear constraints have come up, it is natural to ask whether nonlinear constraint solvers are efficient enough to reasonably automate this system's type inference. Indeed, quadratically-constrained quadratic programs (QCQPs) were explored during the development of this system for the purposes of such automation. QCQPs are like a linear program extended to allow quadratic terms, and this power is sufficient to handle general polynomial constraints as well. However, unlike linear programs, QCQPs are NP-hard to solve outside of special cases, most notably semidefinite programs [136]. Unfortunately, this system's constraints are not semidefinite, so the efficiency of QCQP solving is a real issue. The experiments with a general purpose QCQP solver yielded little success. On simple cases, a prototype implementation ran for several days without finding a solution until it timed out. Nonetheless, the experiments of Section 8.8 suggest that this level of efficiency might be competitive with the preexisting cost-free approach, so this avenue is something to consider in the future, especially as QCQP solvers improve. Until then, maximizing the use of linear constraints is the key to efficient inference.

A second way to handle nonlinear constraints is an iterative approach like Kleene iteration [37] or Newton's method [54]. Conceptually, such an algorithm obtains a solution for a set of nonlinear constraints by creating a set of recursive equations, fixing an initial guess at a fixed point, and iterating. Each iteration should yield a "better" approximation to the solution, as defined by some partial order. However, attempts to devise such an iterative strategy for this type system were unsuccessful because the system must support matrices with negative entries, like the unshift map $\overset{A}{\triangleright}\,{}^{x}_{y}$. When composed, such maps transform positive entries of matrices into negative ones, and vice versa. As a result, any partial ordering on approximate solutions that respects iteration does not properly respect pointwise ordering. Iteration uses the former order, but this system's use of linear programs use the latter, so they cannot be combined.

8.11 Related Work

This section discusses related work along two major axes: How other cost analyses might use cost-freedom, and how other approaches might deal with with inferring similar linear maps to this chapter's cost-free matrices.

Cost-Freedom in Other Cost Analyses Cost-freedom is only relevant in a rather restricted set of circumstances. The main purpose of cost-freedom is to soundly describe how excess resources should be allocated when composing cost analyses. However, it is always just as sound to simply drop excess resources or reanalyze the composed program, so less automated or less mature systems may not have found cost-freedom necessary. Furthermore, the stickier problem solved by cost-freedom, resource-polymorphic recursion, only occurs when a variety of features come together: some form of resource credits (like potential energy), nonlinear cost bounds, and non-tail recursion. Notably, loop programs are always tail recursive, thus they never encounter resource-polymorphic recursion.

Because resource credits are a key factor for the use of cost-freedom, amortized cost analyses are good target for its application. There are many program analysis systems aside from AARA that deal with amortized cost bounds. These systems include program logics augmented with cost credits [10, 66], other type systems [125], recurrence-relation solving [40], term rewriting [87, 107], and more [15, 56, 58, 104, 113, 131]. However, it appears that only AARA-based systems currently exploit the ideas of cost-freedom. Some AARA-based approaches like the program logic of Atkey [10] do not currently use cost-freedom, but they would likely need it if extended to support nonlinear resource functions. Other AARA-based approaches like the term-rewriting system of the work of Moser and Schneckenreither [107] do use cost-freedom, and would likely benefit from this chapter's matrix-based cost-free types.[9]

[9]Indeed, Moser and Schneckenreither explicitly observe that cost-freedom can make their constraint sets blow up in size when using the preexisting method.

As an alternative to using cost-free types, other resource-credit-based approaches seem to favor using ideas from separation logic [120]. For example, consider the work of [30], which, like many alternative cost-analysis techniques, uses a counter to track credits rather than structured resource functions. Cost relations concerning this counter are then imported into a proof assistant where a user (non-automatically) works out the needed invariants. Rather than employ cost-free types, this system reallocates excess credits through a function call via applying separation logic's frame rule to the number of excess credits tracked by the counter. The effect of this frame-rule application is that some quantity of credits r is temporarily ignored, so that if applying some function converts p credits to q, then it also would convert $p+r$ to $q+r$. This reasoning principle is not as complicated as cost-free types because the quantities of credits may be manipulated directly rather than indirectly via resource functions. The tradeoff is that it would be difficult to fully automate such an approach because the quantities of credits typically depend on data structure sizes, which are not generally computable. AARA's indirect manipulation via resource functions avoids most consideration of size by the type system.

Linear Fixed Points This chapter's inference system uses linear programming to solve recursive constraints over matrices, and this method constrains certain features of the system (Section 8.10). However, there are many other approaches in program analysis that solve similar linear, fixpoint-like problems, and it is worth considering how they might be applied to solve the problems of this chapter. Finding such fixed points has been studied since at least the 70s when Cousot and Halbwachs introduced the polyhedral domain for abstract interpretation, which finds linear invariants of programs [38]. Abstract-interpretation work since then has focused more specifically on inferring linear or affine transformations [111, 129], although nothing seems quite sufficient for encoding the transformations in which I are interested. Some other relevant work includes that of de Oliveira et al., which represents polynomials with a linear basis much like AARA, and uses eigenvectors of linear maps to find loop invariants [49]. Unfortunately, extending this idea to a domain like this chapter's where inequalities matter quickly runs into issues like the positivity problem [119]. As part of this work's experiments with iterative methods, discussed in Section 8.10, the work of Reps et al. [127] was also considered. This work uses a tensor-product operation to transform linear context-free equations into regular-language equations, allowing each Newton iteration of Newtonian program analysis [54] to be solved efficiently. In this chapter's setting, however, tensors do not change the linear programs that would be solved.

8.12 Experimental Data

Tables 8.2 to 8.4 contain the data recorded from the experiments outlined in Section 8.8.

	params	—		new		—		old		
d	c	ℓ	constr secs	solve secs	total secs	constrs	constr secs	solve sec	total secs	constrs
1	0	0	0.000042	0.016539	0.016581	4	0.000026	0.015233	0.015259	12
1	0	1	0.000099	0.029961	0.030060	14	0.000041	0.015706	0.015747	53
1	0	2	0.000170	0.045621	0.045791	24	0.000074	0.015939	0.016013	94
1	0	3	0.000284	0.062229	0.062513	34	0.000094	0.016423	0.016517	135
1	0	4	0.000335	0.086318	0.086653	44	0.000464	0.021997	0.022461	176
1	0	5	0.000380	0.089821	0.090201	54	0.000146	0.016866	0.017012	217
1	1	0	0.000025	0.014744	0.014769	4	0.000014	0.015327	0.015341	12
1	1	1	0.000107	0.032644	0.032751	17	0.000081	0.018657	0.018738	72
1	1	2	0.000209	0.045724	0.045933	30	0.000131	0.016680	0.016811	186
1	1	3	0.000268	0.057673	0.057941	43	0.000238	0.017584	0.017822	352
1	1	4	0.000389	0.087940	0.088329	56	0.000644	0.024758	0.025402	570
1	1	5	0.001474	0.114994	0.116468	69	0.000856	0.024653	0.025509	840
1	2	0	0.000019	0.015084	0.015103	4	0.000017	0.015241	0.015258	12
1	2	1	0.000119	0.030124	0.030243	21	0.000088	0.016134	0.016222	95
1	2	2	0.000212	0.042785	0.042997	38	0.000219	0.016977	0.017196	321
1	2	3	0.000308	0.056932	0.057240	55	0.000611	0.019885	0.020496	831
1	2	4	0.000543	0.083748	0.084291	72	0.001632	0.030908	0.032540	1912
1	2	5	0.000629	0.104308	0.104937	89	0.005318	0.057943	0.063261	4132
1	3	0	0.000018	0.016538	0.016556	4	0.000017	0.016756	0.016773	12
1	3	1	0.000149	0.031353	0.031502	21	0.000084	0.015582	0.015666	115
1	3	2	0.000415	0.077578	0.077993	38	0.000374	0.022728	0.023102	491
1	3	3	0.000365	0.060048	0.060413	55	0.001375	0.029278	0.030653	1691
1	3	4	0.000849	0.144139	0.144988	72	0.010170	0.184897	0.195067	5351
1	3	5	0.000605	0.091483	0.092088	89	0.016646	0.147977	0.164623	16400
1	4	0	0.000017	0.015032	0.015049	4	0.000015	0.016025	0.016040	12
1	4	1	0.000216	0.029888	0.030104	21	0.000132	0.017826	0.017958	135
1	4	2	0.000361	0.047575	0.047936	38	0.000521	0.022471	0.022992	702
1	4	3	0.000438	0.071367	0.071805	55	0.004783	0.084018	0.088801	3048
1	4	4	0.000559	0.076119	0.076678	72	0.012303	0.111350	0.123653	12510
1	4	5	0.000818	0.087661	0.088479	89	0.056572	0.602399	0.658971	50434
1	5	0	0.000019	0.015040	0.015059	4	0.000015	0.015895	0.015910	12
1	5	1	0.000225	0.036546	0.036771	21	0.000101	0.018815	0.018916	155
1	5	2	0.000290	0.043415	0.043705	38	0.000833	0.022671	0.023504	953
1	5	3	0.000583	0.061408	0.061991	55	0.006523	0.051278	0.057801	5031
1	5	4	0.003645	0.085588	0.089233	72	0.025974	0.278086	0.304060	25502
1	5	5	0.001311	0.086741	0.088052	89	0.142253	1.285613	1.427866	127944
2	0	0	0.000028	0.015328	0.015356	9	0.000029	0.015573	0.015602	20
2	0	1	0.000289	0.033261	0.033550	29	0.000131	0.019285	0.019416	129
2	0	2	0.000502	0.047067	0.047569	49	0.000160	0.017428	0.017588	238
2	0	3	0.000729	0.059772	0.060501	69	0.000223	0.017761	0.017984	346
2	0	4	0.001074	0.091373	0.092447	89	0.000325	0.022354	0.022679	454
2	0	5	0.001716	0.114943	0.116659	109	0.000558	0.024100	0.024658	562
2	1	0	0.000028	0.015250	0.015278	9	0.000017	0.015819	0.015836	20
2	1	1	0.000360	0.031529	0.031889	35	0.000136	0.017132	0.017268	193
2	1	2	0.000686	0.046000	0.046686	61	0.000500	0.021283	0.021783	623
2	1	3	0.001164	0.109110	0.110274	87	0.001144	0.027268	0.028412	1410
2	1	4	0.001434	0.076398	0.077832	113	0.003020	0.044966	0.047986	2660
2	1	5	0.001770	0.112141	0.113911	139	0.005409	0.059942	0.065351	4476
2	2	0	0.000025	0.015468	0.015493	9	0.000018	0.015634	0.015652	20
2	2	1	0.000429	0.031337	0.031766	44	0.000165	0.017853	0.018018	254
2	2	2	0.000807	0.046065	0.046872	79	0.000981	0.024991	0.025972	1204
2	2	3	0.001801	0.062295	0.064096	114	0.003758	0.052979	0.056737	4150
2	2	4	0.002516	0.080507	0.083023	149	0.013180	0.177627	0.190807	12238
2	2	5	0.002774	0.095361	0.098134	184	0.037644	0.463047	0.500691	32885
2	3	0	0.000028	0.017961	0.017989	9	0.000018	0.017703	0.017721	20
2	3	1	0.000589	0.034955	0.035544	44	0.000203	0.020655	0.020858	316
2	3	2	0.001131	0.052838	0.053969	79	0.001976	0.035812	0.037788	1989
2	3	3	0.001467	0.060390	0.061857	114	0.009366	0.106147	0.115513	9425
2	3	4	0.004764	0.078818	0.083582	149	0.047127	0.589632	0.636759	39081
2	3	5	0.002945	0.094591	0.097536	184	0.175089	1.930193	2.105282	150189
2	4	0	0.000023	0.015724	0.015747	9	0.000015	0.015552	0.015567	20
2	4	1	0.000628	0.030262	0.030890	44	0.000276	0.018271	0.018547	377
2	4	2	0.001415	0.053228	0.054643	79	0.002895	0.048787	0.051682	2970
2	4	3	0.001849	0.066815	0.068664	114	0.020787	0.248645	0.269432	18060
2	4	4	0.003051	0.079288	0.082339	149	0.121902	2.613878	2.735780	97380
2	4	5	0.003527	0.095953	0.099480	184	0.603298	10.648155	11.251453	490539
2	5	0	0.000026	0.016016	0.016042	9	0.000020	0.016125	0.016145	20
2	5	1	0.000678	0.033611	0.034289	44	0.000302	0.019871	0.020173	438
2	5	2	0.001414	0.045110	0.046524	79	0.003294	0.052815	0.056109	4154
2	5	3	0.003176	0.093567	0.096743	114	0.055972	0.501996	0.557968	30940
2	5	4	0.003303	0.081001	0.084304	149	0.240742	3.001745	3.242487	205845
2	5	5	0.006122	0.093704	0.099826	184	1.503743	17.240749	18.744492	1285286

Table 8.2: Experimental data for $d = 1, 2$

params		—		new		—			old	
d	c	ℓ	constr secs	solve secs	total secs	constrs	constr secs	solve sec	total secs	constrs
3	0	0	0.000041	0.015634	0.015675	16	0.000026	0.015668	0.015694	28
3	0	1	0.001046	0.032759	0.033805	43	0.000186	0.018984	0.019170	236
3	0	2	0.001450	0.049265	0.050715	70	0.000379	0.020762	0.021141	443
3	0	3	0.001887	0.062912	0.064799	97	0.000490	0.021987	0.022477	649
3	0	4	0.002996	0.092591	0.095587	124	0.000699	0.026594	0.027293	856
3	0	5	0.003764	0.113372	0.117136	151	0.001065	0.025972	0.027037	1063
3	1	0	0.000036	0.015988	0.016024	16	0.000020	0.016161	0.016181	28
3	1	1	0.001061	0.032295	0.033356	59	0.000331	0.018910	0.019241	357
3	1	2	0.002349	0.049491	0.051840	102	0.001315	0.031187	0.032502	1395
3	1	3	0.003493	0.063860	0.067353	145	0.003248	0.059259	0.062507	3758
3	1	4	0.008033	0.098151	0.106184	188	0.009770	0.126782	0.136552	8266
3	1	5	0.006617	0.100908	0.107525	231	0.020647	0.202641	0.223288	15948
3	2	0	0.000034	0.016045	0.016079	16	0.000017	0.015359	0.015376	28
3	2	1	0.001706	0.033962	0.035668	75	0.000358	0.020168	0.020526	479
3	2	2	0.003464	0.048022	0.051486	134	0.003168	0.046793	0.049961	2913
3	2	3	0.005143	0.069853	0.074996	193	0.014636	0.182602	0.197238	12759
3	2	4	0.008598	0.083686	0.092284	252	0.053517	1.009344	1.062861	46559
3	2	5	0.010271	0.100567	0.110838	311	0.199687	3.691999	3.891686	151105
3	3	0	0.000045	0.023406	0.023451	16	0.000021	0.021076	0.021097	28
3	3	1	0.001888	0.036658	0.038546	75	0.000878	0.022383	0.023261	601
3	3	2	0.004736	0.050563	0.055299	134	0.004270	0.074808	0.079078	5013
3	3	3	0.006328	0.068159	0.074487	193	0.037331	0.611558	0.648889	30823
3	3	4	0.018882	0.084396	0.103278	252	0.187965	5.052531	5.240496	160364
3	3	5	0.022301	0.101693	0.123994	311	0.931370	18.897349	19.828719	749203
3	4	0	0.000039	0.019391	0.019430	16	0.000026	0.017685	0.017711	28
3	4	1	0.002620	0.038494	0.041114	75	0.000599	0.025804	0.026403	721
3	4	2	0.008145	0.050352	0.058497	134	0.007200	0.110314	0.117514	7654
3	4	3	0.007776	0.068316	0.076092	193	0.055860	1.653075	1.708935	60979
3	4	4	0.008763	0.083909	0.092672	252	0.440956	14.162719	14.603675	414116
3	4	5	0.014517	0.105401	0.119918	311	3.195450	171.840712	175.036162	2537242
3	5	0	0.000036	0.015974	0.016010	16	0.000017	0.015511	0.015528	28
3	5	1	0.002837	0.033307	0.036144	75	0.000654	0.022925	0.023579	842
3	5	2	0.006123	0.050831	0.056954	134	0.019261	0.154263	0.173524	10872
3	5	3	0.011121	0.067044	0.078165	193	0.121545	2.997829	3.119374	106554
3	5	4	0.011861	0.084976	0.096837	252	1.122581	53.309085	54.431666	893955
3	5	5	0.014272	0.097983	0.112255	311	9.312985	1179.742946	1189.055931	6785087
4	0	0	0.000069	0.019897	0.019966	25	0.000044	0.016803	0.016847	36
4	0	1	0.001729	0.033189	0.034918	59	0.000337	0.019493	0.019830	365
4	0	2	0.003536	0.056949	0.060485	93	0.000665	0.022510	0.023175	694
4	0	3	0.005667	0.076041	0.081708	127	0.000871	0.027398	0.028269	1024
4	0	4	0.015344	0.128815	0.144159	161	0.001213	0.032686	0.033899	1353
4	0	5	0.010312	0.107239	0.117551	195	0.001545	0.032817	0.034362	1685
4	1	0	0.000059	0.018750	0.018809	25	0.000025	0.018959	0.018984	36
4	1	1	0.005002	0.047072	0.052074	84	0.000478	0.059413	0.059891	568
4	1	2	0.009083	0.055694	0.064777	143	0.002287	0.049415	0.051702	2606
4	1	3	0.013858	0.068480	0.082338	202	0.009116	0.121333	0.130449	8179
4	1	4	0.019216	0.092336	0.111552	261	0.025725	0.343922	0.369647	20762
4	1	5	0.029570	0.150571	0.180141	320	0.090916	1.158684	1.249600	45671
4	2	0	0.000059	0.026470	0.026529	25	0.000033	0.024135	0.024168	36
4	2	1	0.006506	0.036330	0.042836	109	0.000667	0.024205	0.024872	771
4	2	2	0.013719	0.053968	0.067687	193	0.006065	0.090156	0.096221	5731
4	2	3	0.027344	0.080500	0.107844	277	0.029282	0.640377	0.669659	30523
4	2	4	0.028854	0.091598	0.120452	361	0.134407	4.517918	4.652325	133754
4	2	5	0.037787	0.112178	0.149965	445	0.543178	24.347409	24.890587	512519
4	3	0	0.000053	0.017022	0.017075	25	0.000025	0.016221	0.016246	36
4	3	1	0.009729	0.045220	0.054949	109	0.001295	0.030739	0.032034	972
4	3	2	0.018507	0.054518	0.073025	193	0.009415	0.160940	0.170355	10074
4	3	3	0.034511	0.075975	0.110486	277	0.093669	2.533223	2.626892	76585
4	3	4	0.042081	0.087974	0.130055	361	0.612588	21.951089	22.563677	483261
4	3	5	0.051126	0.114073	0.165199	445	4.248675	448.071415	452.320090	2680916
4	4	0	0.000049	0.017285	0.017334	25	0.000021	0.016384	0.016405	36
4	4	1	0.011328	0.035795	0.047123	109	0.001075	0.028545	0.029620	1173
4	4	2	0.034361	0.053880	0.088241	193	0.026270	0.292175	0.318445	15596
4	4	3	0.038971	0.070234	0.109205	277	0.146051	7.262634	7.408685	154511
4	4	4	0.044262	0.091807	0.136069	361	1.697200	167.918893	169.616093	1276812
4	4	5	0.056506	0.141851	0.198357	445	12.724109	4700.986052	4713.710161	9301888
4	5	0	0.000057	0.017507	0.017564	25	0.000025	0.016347	0.016372	36
4	5	1	0.012953	0.034416	0.047369	109	0.001228	0.030184	0.031412	1373
4	5	2	0.028344	0.053067	0.081411	193	0.019104	0.401896	0.421000	22350
4	5	3	0.044963	0.080593	0.125556	277	0.302339	17.910969	18.213308	273266
4	5	4	0.065743	0.087002	0.152745	361	3.626365	588.597753	592.224118	2793847
4	5	5	0.069873	0.114544	0.184417	445	36.485721	70570.439041	70606.924762	25225105

Table 8.3: Experimental data for $d = 3, 4$

	params		—	new			—		old	
d	c	ℓ	constr secs	solve secs	total secs	constrs	constr secs	solve sec	total secs	constrs
5	0	0	0.000077	0.017486	0.017563	36	0.000037	0.016726	0.016763	44
5	0	1	0.003436	0.039762	0.043198	83	0.000409	0.027827	0.028236	532
5	0	2	0.006980	0.061981	0.068961	130	0.000961	0.032137	0.033098	1020
5	0	3	0.024448	0.137288	0.161736	177	0.001332	0.039969	0.041301	1507
5	0	4	0.013967	0.097683	0.111650	224	0.001894	0.037573	0.039467	1995
5	0	5	0.015675	0.114982	0.130657	271	0.002366	0.041321	0.043687	2482
5	1	0	0.000066	0.017811	0.017877	36	0.000029	0.016461	0.016490	44
5	1	1	0.009268	0.036315	0.045583	119	0.000722	0.025874	0.026596	826
5	1	2	0.018078	0.056897	0.074975	202	0.004226	0.076642	0.080868	4345
5	1	3	0.029984	0.076846	0.106830	285	0.020346	0.285027	0.305373	15644
5	1	4	0.036824	0.102572	0.139396	368	0.058999	1.694275	1.753274	45265
5	1	5	0.054419	0.113217	0.167636	451	0.135150	5.011172	5.146322	112556
5	2	0	0.000076	0.017852	0.017928	36	0.000027	0.016072	0.016099	44
5	2	1	0.016984	0.044732	0.061716	155	0.000843	0.032951	0.033794	1128
5	2	2	0.029933	0.058477	0.088410	274	0.010980	0.171963	0.182943	9913
5	2	3	0.040894	0.083943	0.124837	393	0.078169	2.127854	2.206023	62382
5	2	4	0.053890	0.095180	0.149070	512	0.379000	16.143426	16.522426	320332
5	2	5	0.080353	0.129780	0.210133	631	2.058113	213.112385	215.170498	1421648
5	3	0	0.000063	0.018092	0.018155	36	0.000025	0.018186	0.018211	44
5	3	1	0.019366	0.038255	0.057621	155	0.001215	0.033221	0.034436	1432
5	3	2	0.038226	0.058702	0.096928	274	0.017728	0.349406	0.367134	17703
5	3	3	0.055996	0.082414	0.138410	393	0.189165	6.806947	6.996112	160536
5	3	4	0.081476	0.104385	0.185861	512	1.654466	225.653820	227.308286	1195445
5	3	5	0.109081	0.249445	0.358526	631	10.321564	6194.622692	6204.944256	7718071
5	4	0	0.000143	0.017999	0.018142	36	0.000036	0.016671	0.016707	44
5	4	1	0.027136	0.039636	0.066772	155	0.001423	0.034885	0.036308	1727
5	4	2	0.054744	0.057714	0.112458	274	0.034404	0.634923	0.669327	27643
5	4	3	0.073931	0.094613	0.168544	393	0.645488	36.678300	37.323788	328136
5	4	4	0.171072	0.199687	0.370759	512	5.518761	1182.287350	1187.806111	3208306
5	4	5	0.131733	0.186035	0.317768	631	35.674655	154367.904023	154403.578678	27243494
5	5	0	0.000143	0.018394	0.018537	36	0.000029	0.016771	0.016800	44
5	5	1	0.028720	0.038908	0.067628	155	0.001854	0.039140	0.040994	2029
5	5	2	0.066871	0.056814	0.123685	274	0.042691	1.075947	1.118638	39857
5	5	3	0.088594	0.132821	0.221415	393	0.581227	84.127568	84.708795	585037
5	5	4	0.435554	0.603455	1.039009	512	36.992054	5770.545060	5807.537114	7086341
5	5	5	0.155609	0.146074	0.301683	631	timeout	timeout	timeout	timeout
6	0	0	0.000085	0.019476	0.019561	49	0.000030	0.016957	0.016987	52
6	0	1	0.006856	0.053433	0.060289	121	0.000972	0.032932	0.033904	718
6	0	2	0.010274	0.062646	0.072920	193	0.001603	0.034992	0.036595	1385
6	0	3	0.015783	0.083983	0.099766	265	0.001778	0.044023	0.045801	2052
6	0	4	0.022587	0.103950	0.126537	337	0.002568	0.049037	0.051605	2719
6	0	5	0.023988	0.124171	0.148159	409	0.003221	0.053733	0.056954	3386
6	1	0	0.000084	0.019479	0.019563	49	0.000030	0.018554	0.018584	52
6	1	1	0.010019	0.043066	0.053085	163	0.001360	0.032638	0.033998	1130
6	1	2	0.018473	0.059746	0.078219	277	0.006969	0.124090	0.131059	6710
6	1	3	0.031780	0.121696	0.153476	391	0.052798	0.826922	0.879720	27306
6	1	4	0.047615	0.106880	0.154495	505	0.150234	3.723004	3.873238	89024
6	1	5	0.042664	0.126581	0.169245	619	0.433976	9.306156	9.740132	247862
6	2	0	0.000089	0.019588	0.019677	49	0.000030	0.018670	0.018700	52
6	2	1	0.012102	0.043387	0.055489	212	0.002004	0.037488	0.039492	1550
6	2	2	0.027652	0.062923	0.090575	375	0.017866	0.324983	0.342849	15722
6	2	3	0.033779	0.084825	0.118604	538	0.133347	5.242922	5.376269	114325
6	2	4	0.056717	0.108987	0.165704	701	0.908094	67.925591	68.833685	674646
6	2	5	0.064283	0.204992	0.269275	864	5.001324	1296.569909	1301.571233	3412730
6	3	0	0.000668	0.019511	0.020179	49	0.000032	0.016984	0.017016	52
6	3	1	0.017178	0.042157	0.059335	212	0.001928	0.045486	0.047414	1969
6	3	2	0.028368	0.063418	0.091786	375	0.027076	0.725203	0.752279	28390
6	3	3	0.070527	0.114887	0.185414	538	0.438994	21.245849	21.684843	299542
6	3	4	0.058867	0.114669	0.173536	701	3.070279	906.614686	909.684965	2576633
6	3	5	0.306451	0.675611	0.982062	864	71.489372	79469.399906	79540.889278	19035507
6	4	0	0.000104	0.023031	0.023135	49	0.000032	0.018054	0.018086	52
6	4	1	0.018192	0.041655	0.059847	212	0.002206	0.046417	0.048623	2384
6	4	2	0.035369	0.063444	0.098813	375	0.040629	1.421071	1.461700	44632
6	4	3	0.054724	0.095273	0.149997	538	0.744372	73.169898	73.914270	618038
6	4	4	0.098837	0.155009	0.253846	701	8.922452	6845.493612	6854.416064	6994063
6	4	5	0.103610	0.150520	0.254130	864	timeout	timeout	timeout	timeout
6	5	0	0.000111	0.026519	0.026630	49	0.000045	0.018480	0.018525	52
6	5	1	0.023738	0.042042	0.065780	212	0.004236	0.050720	0.054956	2804
6	5	2	0.041496	0.065930	0.107426	375	0.063495	2.458531	2.522026	64600
6	5	3	0.063459	0.091847	0.155306	538	1.080118	212.582821	213.662939	1108168
6	5	4	0.083420	0.110104	0.193524	701	timeout	timeout	timeout	timeout
6	5	5	0.110426	0.139084	0.249510	864	timeout	timeout	timeout	timeout

Table 8.4: Experimental data for $d = 5, 6$

Chapter 9

Quantum Physicist's Method

This chapter introduces a refinement of the physicist's method [133] which I call the *quantum physicist's method*. This chapter then adapts this refinement into AARA to allow the system to more flexibly allocate energy across data structures. This flexibility is enabled by considering multiple simultaneous typings that I call *worldviews* and using a special bookkeeping method for borrowing energy that I call *resource tunneling*.

Worldviews are this system's version of superposition and behave like intersection types or the additive products of linear logic. Worldviews allow choices about the allocation of energy to be delayed, enabling more flexibility. Because each worldview behaves (mostly) like a usual AARA typing, their inference can be automated via linear programming. Further, worldviews enable *resource tunneling*.

Resource tunneling is this system's version of quantum tunneling and allows energy to be passed through potential energy barriers. Such a potential energy barrier might take the form of a function typed $L^1(\tau) \to \mathbb{1} \sim L^1(\tau)$ which has a peak cost higher than its net cost, just like a barrier's height is higher than the ground on either of its sides. Such function types arise often in the presence of reusable resources like memory.

The quantum physicist's method in AARA really shines when its comes to analyzing trees. Worldviews enable resource functions to depend on the height of trees, rather than (loosely) on their number of nodes. Resource tunneling is critical for analyzing tree traversals in AARA.

The quantum physicist's method would not be possible without the remainder contexts of Chapter 5, but I build this chapter upon Chapter 6 for the extra value its generality provides.

9.1 The Problem: Nonlocal Resource Allocation

Sometimes AARA can derive poor cost bounds for the simple reason that its method of bookkeeping potential energy is not flexible enough to allocate energy where it is needed. Because AARA cannot effectively use the energy it has, it demands more energy to cover costs, inflating the cost bounds it finds. This problem occurs because the AARA type system developed so far only allows for "local" energy manipulation—energy is only assigned to data structures directly when they are created. However, sometimes code may demand "nonlocal" energy allocation. The main offending code witnessing this problem can be boiled down to two prototypical code

```
fun branchId lst = let lst2 = lst in
  if f lst then lst else lst2
```

Figure 9.1: Code for branching problem of energy allocation

```
fun seqApp lst = let lst2 = lst in
  let _  = f lst2 in f lst
```

Figure 9.2: Code for sequencing problem code of energy allocation

patterns which are exemplified in Figures 9.1 and 9.2, where f : $L^1(\mathbb{1}) \to \mathbb{B} \sim L^1(\mathbb{1})$. Later I show how these same code patterns arise in, e.g., tree traversals.

The "branching problem" of energy allocation is witnessed by the function `branchId` in Figure 9.1. Intuitively, this function is the indentity on lists and accrues no net cost. Thus, it should be able to be given a type which conserves all energy like $L^2(\mathbb{1}) \to L^2(\mathbb{1}) \sim L^0(\mathbb{1})$. However, AARA can only find a type like $L^2(\mathbb{1}) \to L^1(\mathbb{1}) \sim L^0(\mathbb{1})$, which loses half of the energy. This loss occurs because the body of `branchId` may return `lst` through one of two different aliases, either `lst` or `lst2`, depending on the (unknown) return of `f lst`. Because AARA can only assign energy locally in the let expression prior to the branch, AARA assigns energy to *each* alias to be ready for whichever branch is taken. Whichever alias is unused therefore wastes energy.

To summarize the branching problem at a high level, the problem is that different branches of code may use different existing data structures. Thus, because it is not generally knowable which branch of code will be executed, energy must be set up on the data structures for *all* branches, which wastes the energy on the data structures of branches not taken. It would be preferable if there was some way to delay assigning energy to data structures when creating them, so that the correct amount of energy could be assigned when the relevant code branch is reached. That is, it would be preferable if energy could be assigned *nonlocally*.

The "sequencing problem" of energy allocation is witnessed by the function `seqApp` in Figure 9.2. Intuitively, this function just acts as `f` and accrues no net cost. Thus, it should be able to given a type which conserves all energy like $L^2(\mathbb{1}) \to \mathbb{B} \sim L^2(\mathbb{1})$. However, AARA can only find a type like $L^2(\mathbb{1}) \to \mathbb{B} \sim L^1(\mathbb{1})$, which loses half of the energy. This loss occurs because the body of `seqApp` aliases its argument `lst` as `lst2` and calls `f` on each alias. The peak cost of `f` is nonzero, so the first call to `lst2` requires `lst2` to carry some amount of energy. Then applying `f` to `lst2` incurs no net cost, so the leftover energy of `lst2` should be able to be reassigned to the identical value `lst` for the next call to `f`. However, AARA cannot recover the leftover energy of `lst2` because `lst2` is never used in the code again. Energy is only able to be manipulated locally, so energy can only possibly be moved between `lst` and `lst2` when some subexpression uses both variables. Thus the energy leftover on `lst2` is lost, wasting energy.

To summarize the sequencing problem at a high level, the problem is that sequential portions of code may use different data structures. Thus, any reusable resources allocated to the first portion of code should be able to reallocated to the second portion of code. However, this transfer of energy can only happen if expressions exist between the two code portions that use both

```
    fun mem (x,tr) = case tr of
      | Leaf -> false
      | Node (t1,y,t2) ->
        if y = x
        then true
        else if y < x
        then mem (x,t1)
        else mem (x,t2)
```

Figure 9.3: Code for binary tree search

```
    fun size tr = case tr of
      | Leaf -> 0
      | (t1,y,t2) ->
        let t1size = size t1 in
        let t2size = size t2 in
        t1size + t2size + 1
```

Figure 9.4: Code for binary tree size

portions' data structures. Without such expressions, the leftover energy is stranded. It would be preferable if there was some way to more freely transfer energy from the first to second code portion. That is, it would be preferable if energy could be transferred *nonlocally*.

Both of these examples are rather contrived and only lose a constant factor of energy. Thus, they do not change the asymptotic complexity, and one might think that the problems they exemplify are not particularly detrimental. However, these are just minimal examples, and the problems run much deeper. For one thing, these examples can be iterated, pumping the constant factor loss arbitrarily low. For another, the same code patterns arise naturally in common code like tree manipulations. The branching problem arises during tree search (Figure 9.3), and the sequencing problem arises during tree traversal (Figure 9.4). In both cases, code has a worst-case peak cost equal to the depth of the trees involved, but AARA finds bounds in terms of the number of nodes because it must keep energy on each subtree. For example, AARA types `size` as $T^1(\mathbb{Z}) \to \mathbb{Z} \sim T^1(\mathbb{Z})$, where each annotation is for the linear resource function. Such cost bounds can be exponentially bad for balanced trees.

9.2 The Linear Ideas: Quantum Physics and Superposition

To address the problem described in Section 9.1, I make a connection between the domain of cost analysis and the domain of quantum physics. This connection yields what I call the *quantum physicist's method*, which is a refinement of the physicist's method of amortized cost analysis [133]. While this might seem to be a surprising connection, it turns out to be rather natural when considering both domains' underpinnings in linear logic.

Like the AARA system developed so far, classical physics only allows for local manipula-

&R
$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P\&Q}$$

&L1
$$\frac{\Gamma, P \vdash R}{\Gamma, P\&Q \vdash R}$$

&L2
$$\frac{\Gamma, P \vdash R}{\Gamma, Q\&P \vdash R}$$

Figure 9.5: Linear logic rules for additive product

tions of energy. However, actual physical phenomena are *nonlocal*, as shown theoretically via Bell's theorem [16] and experimentally via Bell tests (e.g., [60]). Instead of classical physics, quantum physics has been adopted (at least in part) to properly describe such nonlocality. This section shows that a similar quantum leap can be made in AARA's cost analysis to properly handle the desired nonlocality of energy described in Section 9.1.

First, it is useful to note that quantum physics has long been viewed as governed by the same linear logic that underlies AARA's cost analysis [2, 53, 63, 122]. The clearest hint that such a connection between the two domains exists are the *no cloning* and *no deletion* theorems, which state that arbitrary quantum states can neither be duplicated nor thrown away, respectively. (Some states, like classical states, can still be duplicated or thrown away; it is just impossible for arbitrary states.) These theorems mirror linear logic's lack of contraction and weakening, respectively.

Of course, quantum physics is more famous for its *quantum superposition*. Quantum states can be described as being in quantum superposition, which acts as a special kind of probability distribution over classical, everyday states. This notion of probability is not ordinary probability, but rather a special sort that makes the idea of negative probability coherent. These signed probabilities can cancel each other out, allowing quantum superposition to support more interesting correlations than a normal probability distribution. Sampling from this probability distribution occurs via *observing* the state, which causes the probability distribution to *collapse* into a classical state.[1]

While quantum superposition might seem exotic, it turns out to fall squarely in the domain of linearity. Quantum superposition is simply a specialized version of the linear principle of superposition: for a linear operator F, if $F(x) = x'$ and $F(y) = y'$, then $F(x + y) = x' + y'$. This principle is essentially just the algebraic notion that linear operators distribute over addition, but arranged to be suggestive of how linear relations may be "overlaid" on one another through some notion of addition. Quantum superposition arises when different solutions to Schrödinger's equation are "added" to one another.

Linear algebra obviously admits the superposition principle as well. So does linear logic: if $\Gamma, P \vdash Q$ and $\Gamma, R \vdash S$, then $\Gamma, P\&R \vdash Q\&S$. In this sense, syntactic entailment $\vdash$ is a linear operator, and the notion of addition is the *additive product* (i.e., "with" or "&"), given by the rules of Figure 9.5. It is this linear logical superposition that I will later adapt into AARA to mimic some of the behaviour enabled by quantum superposition.

Quantum superposition enables some of the unusual nonlocal effects of quantum physics through *entanglement*. To understand entanglement, it is helpful to understand a bit of quantum

[1]This description is a simplified account of the role of measurement in quantum mechanics, but is sufficient for my purposes here.

notation. The quantum superposition of two states ϕ and ψ is typically written as $a \cdot |\phi\rangle + b \cdot |\psi\rangle$ for some scalars a, b.[2] A state with multiple components ϕ and ψ may be written as the tensor product of each component, $|\phi\rangle \otimes |\psi\rangle$. Two quantum states $|\phi\rangle \otimes |\phi'\rangle$ and $|\psi\rangle \otimes |\psi'\rangle$ are entangled when their sum $a \cdot |\phi\rangle \otimes |\phi'\rangle + b \cdot |\psi\rangle \otimes |\psi'\rangle$ cannot be decomposed into a product of the sums for each state component $(c \cdot |\phi\rangle + d \cdot |\psi\rangle) \otimes (c' \cdot |\phi'\rangle + d' \cdot |\psi'\rangle)$. As a result of this failure to decompose, an entangled quantum state $a \cdot |\phi\rangle \otimes |\phi'\rangle + b \cdot |\psi\rangle \otimes |\psi'\rangle$ can permit behaviour that cannot be expressed locally in terms of its parts.

Cost analysis also has a form of entanglement that I organize with what I call "worldviews", which are hypothetical allocations of resources. This entanglement can be seen at a high level by considering Alice and Bob paying a \$20 bill at a restaurant when each has \$20. The corresponding notion of physical state is the contents of Alice and Bob's wallets, A and B dollars respectively, which can be written as $|\$A\rangle \otimes |\$B\rangle$. In one worldview Alice pays the bill, resulting in state $|\$0\rangle \otimes |\$20\rangle$. In another worldview Bob pays the bill, resulting in state $|\$20\rangle \otimes |\$0\rangle$. Thus, via superposition, the state induced by paying the bill could be described as $|\$0\rangle \otimes |\$20\rangle + |\$20\rangle \otimes |\$0\rangle$. If this state was decomposable as $(a \cdot |\$0\rangle + b \cdot |\$20\rangle) \otimes (c \cdot |\$0\rangle + d \cdot |\$20\rangle)$, then there would be a worldview $|\$20\rangle \otimes |\$20\rangle$ where no one pays the bill,[3] which should not be possible. Thus the amount of money left in each person's wallet is entangled.

In this way, entanglement captures the idea that certain parts of state can depend upon others. Determining one part of a state may then have a nonlocal effect on another part. For example, after Alice and Bob pay the bill, one can observe the contents of Alice's wallet. This observation determines the contents of Bob's wallet without directly involving Bob.

One of the key nonlocal quantum phenomena enabled by quantum superposition is *quantum tunneling*. Quantum tunneling occurs when a particle with passes through a potential barrier that seems to require a higher amount of energy than the particle has. In classical physics, such a scenario might be described using a ball and a hill. If the ball is in a valley adjacent to the hill, a certain amount of energy would be needed to push it over the hill to the next valley. Quantum tunneling would be where there is not enough energy to go over the hill, and yet the ball still ends up in the next valley as if it "tunneled" through the hill.

Quantum tunneling can occur in quantum physics because there is a rule that one cannot perfectly know a particle's location and energy (Heisenburg's uncertainty principle). This uncertainty can be expressed via a superposition over various locations and energy amounts, where a tighter distribution over one quantity results in a looser distribution over the other. In this way, if a particle is well-known to be prior to a potential barrier, it actually can be ascribed a wide range of energy values. With some small probability, it might actually have enough energy to pass the barrier. But interestingly, passing such a barrier does *not* collapse the particle's superposition; if observed after the barrier the particle might be found to have lower energy than the barrier would have required. In some sense, this situation can be viewed as the particle temporarily borrowing energy from its environment. The state with high particle energy indicates that the particle can borrow enough energy to pass the barrier, and a low-energy particle past the barrier has simply returned this energy to wherever it came from in the environment.

[2]For my purposes, it is not necessary to go into what the scalars a, b are.

[3]One could also consider letting one of b or d be zero, but doing so would zero out the state where either Alice or Bob pay nothing, respectively. Because there clearly are such states where the other pays the whole bill, this circumstance can be ruled out.

Cost analysis also exhibits quantum tunneling in a form that I call "resource tunneling". Resource tunneling is a way of bookkeeping the temporary borrowing of resources to overcome potential barriers. The principle of resource tunneling is the following: as long *some* worldview covers the peak cost, then *all* worldviews may pay only the net cost.

To see resource tunneling in action, consider Alice and Bob going to buy some chocolate and sharing \$5 between themselves. Alice wants a \$3 pack of chocolates from a vending machine, and Bob wants a \$2 chocolate bar from a store shelf, so it would seem that the appropriate worldview to use is where Alice has \$3 and Bob \$2. However, if the vending machine requires \$5 bills (and thus acts as a potential barrier) this split of money does not allow both people to buy their chocolate. Nonetheless, a different worldview witnesses that Alice can use this vending machine: the one where she holds all the money. As a result, resource tunneling concludes that both people can buy their chocolate, as they can cover the net cost. Intuitively, if Alice were to take all the money to use the vending machine, she would be left with \$2 change that she can give to Bob so that he can buy his chocolate.

An explicit bookkeping of the resource tunneling example is represented in Table 9.1, wherein each line is a different worldview and A/B represents Alice's money A and Bob's money B in a given state. Even though this accounting includes negative amounts of money, each column contains a worldview with all nonnegative amounts of money, so the peak cost is always covered by at least one worldview. The point where Alice gives Bob the change is where this peak-covering worldview changes. Switching between worldviews in this way therefore implicity encodes how resources get borrowed.

initial	Alice buys	change	Bob buys
5/0	0/0	2/0	2/-2
3/2	-2/2	0/2	0/0

Table 9.1: Resource tunneling bookkeeping

9.3 Refining the Physicist's Method

This section describes the quantum physicist's method in a context divorced from the AARA type system. The quantum phyicist's method is not an AARA-specific technique, but rather a refinement of the usual physicist's method.

Classical Physicist's Method

First it is useful to recall how the classical physicist's method works in Sleator and Tarjan's set up of amortized analysis. This setup considers an abstract program state space being manipulated by a sequence of different kinds of operations such that operation o_i transforms state s_i into s_{i+1}. If each operation o_i has cost c_i in state s_i, then the net cost of the sequence operations is $\sum_i c_i$.

The hope is that the worst-case costs of each kind of operation can be combined to get a tight bound on the worst-case net cost of the sequence of operations. However, a bound found by adding worst-case costs is often quite loose. Suppose that some operation is performed n

times in sequence, and that it is free in all states but one, where it costs c. Then the sequence's actual cost is c, and the worst-case cost of the operation is (at least) c. However, bounding the sequence's cost by adding the worst case cost n times yields $n \cdot c$, which is much worse than the ground truth.

The key observation of amortized analysis is that there is a different notion of worst-case cost that yields tighter cost bounds using the above reasoning. That new notion is the *amortized* cost, as defined in Definition 9.3.1. I use the word "feasible" in this definition to allow for restricting consideration to certain sequences, but often such restriction is not considered so that operations might come in any order.

Definition 9.3.1 (amortized cost)**.** *Let S be the set of all the program states reachable by some feasible sequence of operations. Let Φ be some function from S to nonnegative real numbers. Then a valid amortized cost of the operation o with respect to Φ is given by some a such that, for every feasible sequence of operations,*

$$\forall o_i = o, s_i \in S.\ c_i - a \leq \Phi(s_i) - \Phi(s_{i+1})$$

That is, the difference between the true and amortized cost is always bounded by the difference between each of the program state's potential energy.

This definition of amortized cost can then be used to bound the true net cost of sequences of operations as follows in Theorem 9.3.1. This bound makes use of s_0, the initial state, and s_{n+1}, the final state after all n operations in the sequence.

Theorem 9.3.1 (amortized net cost bound [133])**.** *Letting a_i be the amortized cost of operation o_i,*

$$\sum_{i=0}^{n} c_i \leq \Phi(s_0) - \Phi(s_{n+1}) + \sum_{i=0}^{n} a_i$$

Proof. This inequality holds directly as follows:

$$\begin{aligned} \sum_{i=0}^{n} c_i &\leq \sum_{i=0}^{n} (\Phi(s_i) - \Phi(s_{i+1}) + a_i) && \textit{def} \\ &= \Phi(s_0) - \Phi(s_{n+1}) + \sum_{i=0}^{n} a_i && \textit{telescopic cancellation} \end{aligned}$$

□

Furthermore, while it in many contexts it is not relevant that the potential function yields nonnegative numbers, it is important when dealing with reusable resources. In this setting, some costs may be negative to represent the return of resources, and the peak cost $\max_{m \in [-1,n]} \sum_{i=0}^{m} c_i$ can be more important to know than the net cost. For example, space costs are best described with peak cost because net space costs are zero. Luckily, peak-cost bounds may expressed with amortized costs just as nicely as net-cost bounds, as given in Theorem 9.3.2.

Theorem 9.3.2 (amortized peak cost)**.** *Letting a_i be the amortized cost of operation o_i,*

$$\max_{m\in[-1,n]} \sum_{i=0}^{m} c_i \leq \Phi(s_0) + \max_{m\in[-1,n]} \sum_{i=0}^{m} a_i$$

Proof. This inequality is proven by induction over the sequence length.

n=0 For empty sequences of operations, the true peak cost is 0 and the amortized peak cost is also 0. Thus the nonnegativity of the potential function completes this case.

n+1 Assuming the inequality holds for sequences of length n, it can then be shown that it holds for sequences of length $n + 1$.

$$\begin{aligned}
\max_{m\in[-1,n+1]} \sum_{i=0}^{m} c_i &= \max\Big(\max_{m\in[-1,n]} \sum_{i=0}^{m} c_i, \sum_{i=0}^{n+1} c_i\Big) && \textit{def} \\
&\leq \max\Big(\Phi(s_0) + \max_{m\in[-1,n]} \sum_{i=0}^{m} a_i, \sum_{i=0}^{n+1} c_i\Big) && \textit{IH} \\
&\leq \max\Big(\Phi(s_0) + \max_{m\in[-1,n]} \sum_{i=0}^{m} a_i, \Phi(s_0) - \Phi(s_{n+2}) + \sum_{i=0}^{n+1} a_i\Big) && \textit{Theorem } 9.3.1 \\
&\leq \max\Big(\Phi(s_0) + \max_{m\in[-1,n]} \sum_{i=0}^{m} a_i, \Phi(s_0) + \sum_{i=0}^{n+1} a_i\Big) && \Phi(s_{n+2}) \geq 0 \\
&= \Phi(s_0) + \max_{m\in[-1,n+1]} \sum_{i=0}^{m} a_i && \textit{algebra}
\end{aligned}$$

□

In the classical amortized-cost framework, better choices of potential function allow for lower amortized costs to be assigned, yielding tighter implied cost bounds for sequences of operations. At the ideal limit, each amortized cost is zero, giving the tightest bounds possible. In this ideal setting, Theorem 9.3.1 says that the net cost is bounded by the difference between the initial and final states' energies, $\Phi(s_{n+1}) - \Phi(s_0)$. Similarly, Theorem 9.3.2 says that the peak cost is bounded by the initial state's energy, $\Phi(s_0)$. Conveniently, these bounds require no consideration of intermediate program states. AARA exists in this setting, where true costs are bound using only potential functions of initial and final program states.

Quantum Physicist's Method

While the classical physicist's method is all well and good, it leaves the construction of a good potential function Φ as a complete black box. In practice, however, there seems to be a favored method for constructing Φ: *local* construction. By "local", I mean that energy is treated as if it is stored locally with different parts of the program state, so that the energy of the whole state is just the sum of the energies of its parts. This local definition of the potential function is the approach taken by AARA in each other chapter of this thesis.

There are many benefits of locally constructed potential functions. The key is their locality: manipulating one part of the program state in some way yields the same change in potential energy no matter the rest of the program state. Thus, operations like AARA's shift $\lhd$ can be sensibly defined, as the energy of a list in no way depends on the energy of other data structures that may be present. Locality makes for ergonomic reasoning.

However, locally constructed potential functions are prone to the problems described in Section 9.1. Sometimes it more desireable to *not* tie energy down to particular parts of the program state, as it might not be clear which parts of the program state will need the energy.

To get the best of both local and nonlocal potential functions, I provide the quantum physicist's method as a framework for defining potential functions. This framework uses local manipulations of energy augmented by the worldviews and resource tunneling described in Section 9.2. Together, these features allow local reasoning to describe nonlocal potential functions. In some sense, what this framework does is allow a nondeterministic choice between different potential functions so that the best one may be used at any given point.

To describe the quantum physicist's method, it is first necessary to divide the program state into parts. This division can be made by treating the state as a (labelled) set of parts. A potential function over such a state s is local when it can be written as $\Phi(s) = \sum_{t \in s} \Phi(t)$.

It is then necessary to describe the behaviour of operations o in more detail. Before, such an operation merely indicated a transition between states. In this new setting, this notion is refined so that o not only transitions between states, but also are parameterized by the labels of some particular part(s) of the state on which they act. The action of such an operation is dependent only upon those labels on which they act. In principle, such an action can result in the creation of new parts and the deletion of existing parts. Thus, when o_i brings s_i to s_{i+1}, the state s_{i+1} may have different parts than s_i. To ensure that operations are well-defined, feasible operation sequences only consider those operations o_i acting on labels that exist in state s_i.

Now consider a collection of functions Φ_w indexed by some w— call these w *worldviews*. Moreover, consider a collection of *quantum costs* b, one for each kind of operation o, with respect to that collection of Φ_w. These Φ_w are local potential functions except that they may take on negative values. Additionally, for all feasible sequences of operations, the functions Φ_w and numbers b must satisfy the following two conditions, where b_i is the quantum cost of o_i:

(1) For all i indexing the sequence and all $t \in s$, for some worldview w,

$$\Phi_w(t) \geq 0$$

(2) For all i indexing the sequence and all worldviews w,

$$c_i - b_i \leq \Phi_w(s_i) - \Phi_w(s_{i+1})$$

These conditions are the conditions of resource tunneling described in Section 9.2. When these conditions hold, Theorem 9.3.3 follows:

Theorem 9.3.3 (quantum physicist's method)**.** *If Φ is defined such that*

$$\Phi(s) = \max_w \Phi_w(s)$$

then the quantum cost b with respect to the collection Φ_w is the amortized cost with respect to the potential function Φ.

Proof. To show the validity of this choice of potential function and amortized cost, it only needs to be checked against Definition 9.3.1. This definition amounts to checking that $\Phi(s_i)$ is always nonnegative and that $c_i - b_i \leq \Phi(s_i) - \Phi(s_{i+1})$

nonnegativity The nonnegativity of Φ holds as follows, letting the worldview ensured by resource tunneling condition (1) be w.

$$\begin{aligned}
\Phi(s_i) &= \max_u \Phi_u(s_i) && \textit{def} \\
&\geq \Phi_w(s_i) && \textit{algebra} \\
&= \sum_{t \in s_i} \Phi_w(t) && \textit{local} \\
&\geq 0 && (1)
\end{aligned}$$

cost inequality The amortized cost inequality holds as follows using resource tunneling condition (2), letting the worldview w be the worldview maximizing $\Phi(s_{i+1})$.

$$\begin{aligned}
\Phi(s_i) - \Phi(s_{i+1}) &= \max_u \Phi_u(s_i) - \max_u \Phi_u(s_{i+1}) && \textit{def} \\
&= \max_u \Phi_u(s_i) - \Phi_w(s_{i+1}) && \textit{def} \\
&\geq \Phi_w(s_i) - \Phi_w(s_{i+1}) && \textit{algebra} \\
&\geq c_i - b_i && (2)
\end{aligned}$$

□

As a result of Theorem 9.3.3, the potential functions generated by the quantum physicist's method slot directly into the classical physicist's method, and corollaries to Theorems 9.3.1 and 9.3.2 follows. However, unlike the classical physicist's method, the quantum physicist's method describes a concrete way of constructing potential functions. This construction conservatively extends the local potential functions that are commonly used, and also allows the creation of nonlocal potential functions via nonlocal maxima. By constructing potential functions in this way and using resource tunneling, the resulting nonlocal potential functions can still largely be manipulated locally, allowing similar ergonomic usage as local potential functions. As this chapter will show, such ergonomic usage allows automation in AARA.

9.4 Quantum Physicist's Method System

This section details how to adapt AARA to use the quantum physicist's method. The key is to represent multiple simultaneous typings by using multidimensional maps for annotations, rather than the "classical" annotation maps of previous chapters. These multidimensional maps are organized to use a special namespace of *worldviews*, where each worldview represents a different typing. Thus, these multidimensional maps go from annotation indices and worldviews to annotation values.

Notationally, for these multidimensional maps, I use subscripts for worldview arguments and function application syntax for annotation index arguments. With this notation P_w is a classical annotation map, and $P(i)$ is a map from worldviews to annotations for index i. I also use $\mathtt{wv}(P)$ to pick out the worldviews over which such a map P is defined. Finally, let operations normally defined over classical annotation maps, like shifting and sharing, be distributed over worldviews so that, e.g., $\Upsilon_z^{x,y}(P) = \lambda w.\ \Upsilon_z^{x,y}\,(P_w)$.

After replacing classical annotation maps with these new multidimensional maps, all that remains is to handle the additional dynamics of worldviews and resource tunneling. This section details these features are handled.

9.4.1 Typing Rules

The quantum physicist's method typing rules are given across Figures 9.6 to 9.8. These new typing rules make use of a similar typing judgment to those used in Chapters 5 and 6, except that each typing judgment uses multidimensional maps for its annotations. (In particular, I continue Chapter 6's parameterization on a matrix A for shifting some family of resource functions $R_k(n)$.)

The new typing judgment is:

$$\Gamma \mid P \vdash e : \tau \mid Q$$

This typing judgment means that, in type context Γ with initial energy annotated by P, the expression e is typed τ, with remainder energy annotated by Q. To be explicit, P_w is indexed by $\mathit{Ind}(\Gamma)$ for every worldview $w \in \mathtt{wv}(P)$, and Q_w is indexed by $\mathit{Ind}(\Gamma, \mathtt{ret} : \tau)$ for every worldview $w \in \mathtt{wv}(Q)$. Further, call a classical annotation map *classically valid* when it is nonnegative, and call a multidimensional map R *quantumly valid* when R_w is classically valid for some $w \in \mathtt{wv}(R)$; the typing judgment requires that both P and Q are quantumly valid. Note that this typing judgment allows negative annotations and does not require that P and Q be defined over the same worldviews.

The main difference between this typing judgment and those of Chapters 5 and 6 is the use of multidimensional maps for annotations. These multidimensional maps act similarly previous chapters' annotations at each worldview, and therefore they behave as if they organize multiple simultaneous typings. A secondary difference is the allowance of negative annotations. These negative annotations allow more flexibility in the dynamics of resource allocation.

Only a couple of rules actually change or need to be added after upgrading annotation maps to multidimensional maps. These rules are given in Figure 9.6 and are explained more in the following subsections. Otherwise, every rule in Figures 9.7 and 9.8 matches the rules in Chapters 5

Q-SuperposeL
$$\frac{\Gamma \mid P, u \mapsto P_w \vdash e : \tau \mid Q}{\Gamma \mid P \vdash e : \tau \mid Q}$$

Q-SuperposeR
$$\frac{\Gamma \mid P \vdash e : \tau \mid Q}{\Gamma \mid P \vdash e : \tau \mid Q, u \mapsto Q_w}$$

Q-CollapseL
$$\frac{\Gamma \mid P \vdash e : \tau \mid Q}{\Gamma \mid P, u \mapsto \vec{a} \vdash e : \tau \mid Q}$$

Q-CollapseR
$$\frac{\Gamma \mid P \vdash e : \tau \mid Q, u \mapsto \vec{a}}{\Gamma \mid P \vdash e : \tau \mid Q}$$

Q-App
$$\frac{P_u \geq 0}{\Gamma, x : \tau, f : \tau \overset{\vec{b}|\vec{c}}{\to} \sigma \mid \Upsilon_x^{x,\texttt{arg}}(P, \lambda w.\, \vec{b}) \vdash f\ x : \sigma \mid \Upsilon_x^{x,\texttt{arg}}(P, \lambda w.\, \vec{c})}$$

Q-Fun
$$\frac{\Gamma, x : \tau, f : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma \mid \lambda w.\, (0 \cdot P_w, [x/\texttt{arg}]\vec{c}) \vdash e : \sigma \mid \lambda w.\, (0 \cdot P_w, [x/\texttt{arg}]\vec{d})}{\Gamma \mid P, Q \vdash \texttt{fun}\ f\ x \ =\ e : \tau \overset{\vec{c}|\vec{d}}{\to} \sigma \mid P, Q}$$

Q-Let
$$\frac{\Gamma \mid P \vdash e_1 : \sigma \mid R \qquad \Gamma, x : \sigma \mid [x/\texttt{ret}]R \vdash e_2 : \tau \mid Q, S \qquad S \geq 0}{\Gamma \mid P \vdash \texttt{let}\ x \ =\ e_1\ \texttt{in}\ e_2 : \tau \mid Q}$$

Figure 9.6: Key quantum physicist's method typing rules

and 6.

The first four typing rules of note are the structural rules of *Q-SuperposeL, Q-SuperposeR, Q-CollapseL*, and *Q-CollapseR* which all relate to worldview manipulation. These rules cover each combination of two different axes: whether they affect the initial or remainder contexts, and whether they are a form of contraction (superposition) or weakening (collapsing). The rules' relations to contraction or weakening are reversed depending on whether they apply to the initial or remainder context. (This reversing intuitively occurs because the remainder contexts "reverse" computation.) Each superposing rule intuitively duplicates an existing worldview for use in the typing derivation. These distinct worldviews can then evolve differently to satisfy typing rules that might be derived in distinct ways. Dually, each collapsing rule throws away an existing worldview. (As a side condition, such collapsing requires the resulting context is still quantumly valid.) More discussion on worldviews is present in Section 9.8.

The rule *Q-App* is the rule responsible for reifying resource tunneling (which otherwise is present due to worldviews and negative annotations). This rule requires that, in some worldview u, the argument of the function is typed with enough energy to run. It then allows *all* worldviews to pay only the net cost of running the function.

The rule *Q-Fun* is of interest because this system forces function types to be the same across all worldviews. This restriction matches how function types have been presented throughout

$$\textsc{Q-Sub}\quad \frac{\Gamma \mid P' \vdash e : \tau \mid Q' \qquad P \geq P' \qquad Q \leq Q'}{\Gamma \mid P \vdash e : \tau \mid Q}$$

$$\textsc{Q-Var}\quad \frac{}{\Gamma, x : \tau \mid \curlyvee_{x}^{x,\texttt{ret}}(P) \vdash x : \tau \mid P}$$

$$\textsc{Q-Tick}\quad \frac{P(\mathtt{c}) = Q(\mathtt{c}) + \lambda w.\, r \qquad \forall i \neq \mathtt{c}.\, P(i) = Q(i)}{\Gamma \mid P \vdash \texttt{tick}\{r\} : \mathbb{1} \mid Q}$$

$$\textsc{Q-Pair}\quad \frac{}{\Gamma, x : \tau, y : \sigma \mid \curlyvee_{x}^{x,\texttt{ret.1}^{\texttt{st}}}(\curlyvee_{y}^{y,\texttt{ret.2}^{\texttt{nd}}}(P)) \vdash \langle x,\, y\rangle : \tau \otimes \sigma \mid P}$$

$$\textsc{Q-CaseP}\quad \frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid P \vdash e : \tau \mid Q}{\Gamma, x : \sigma \otimes \rho \mid \curlyvee_{x.1^{\texttt{st}}}^{y,x.1^{\texttt{st}}}(\curlyvee_{x.2^{\texttt{nd}}}^{z,x.2^{\texttt{nd}}}(P)) \vdash \texttt{case}\ x\ \texttt{of}\ \langle y,\, z\rangle \to e : \tau \mid \curlyvee_{x.1^{\texttt{st}}}^{y,x.1^{\texttt{st}}}(\curlyvee_{x.2^{\texttt{nd}}}^{z,x.2^{\texttt{nd}}}(Q))}$$

$$\textsc{Q-SumL}\quad \frac{}{\Gamma, x : \tau \mid \curlyvee_{x}^{x,\texttt{ret.l}}(P) \vdash \texttt{l}(x) : \tau \oplus \sigma \mid P, Q}$$

$$\textsc{Q-SumR}\quad \frac{}{\Gamma, x : \sigma \mid \curlyvee_{x}^{x,\texttt{ret.r}}(P) \vdash \texttt{r}(x) : \tau \oplus \sigma \mid P, Q}$$

$$\textsc{Q-CaseS}\quad \frac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid P, Q, R' \vdash e_1 : \tau \mid S, T, U' \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid P, Q', R \vdash e_2 : \tau \mid S, T', U\end{array}}{\Gamma, x : \sigma \oplus \rho \mid P, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(Q), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(R) \vdash \texttt{case}\ x\ \texttt{of}\ \texttt{l}(y) \to e_1 \mid \texttt{r}(z) \to e_2 : \tau \mid S, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(T), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(U)}$$

Figure 9.7: Other quantum physicists method typing rules 1

Q-Nil

$$\frac{}{\Gamma \mid P \vdash [\,] : L(\tau) \mid\ P, Q}$$

Q-Cons

$$\frac{}{\Gamma, x : \tau, y : L(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(P))) \vdash x :: y : L(\tau) \mid\ P}$$

Q-CaseL

$$\frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(R, S)\end{array}}{\Gamma, x : L(\sigma) \mid P, \curlyvee_x^{x,x'}(Q) \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid\ R, \curlyvee_x^{x,x'}(S)}$$

Q-Leaf

$$\frac{}{\Gamma \mid P \vdash \mathtt{leaf} : T(\tau) \mid\ P, Q}$$

Q-Node

$$\frac{}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y',z'}(P)))) \vdash \mathtt{node}(x,\ y,\ z) : T(\tau) \mid\ P}$$

Q-CaseT

$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(R, S)\end{array}}{\Gamma, t : T(\sigma) \mid P, \curlyvee_t^{t,t'}(Q) \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{leaf} \to e_1 \mid \mathtt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ R, \curlyvee_t^{t,t'}(S)}$$

Figure 9.8: Other quantum physicists method typing rules 2

this thesis, with classical annotations baked in. It might be possible to extend function types to themselves make use of worldviews, where such worldviews would allow functions to be given multiple typings similarly to cost-free typings (Chapter 8)—this direction is left for future work.

The rule *Q-Let* is special because it revokes the ability for the annotations of let-bound variables to be negative at the end of their scopes. This revocation is necessary because removing a variable from scope with negative annotations is akin to erasing debts and unsoundly gains energy.[4] Thus, the conditon is left that S, the remainder annotation of the let-bound variable, must be nonnegative.

The observations about the rule *Q-Let* are actually somewhat far reaching. Variables with negative annotations cannot be removed from scope willy-nilly. As a result, this type system does not admit weakening for variables. Weakening for variables was not *needed* since Chapter 5, but has usually been *admissable*—it no longer is. Now, with the quantum physicist's method, typed variables cannot be contracted/duplicated (because positive energy cannot be duplicated) nor weakened/discarded (because negative energy cannot be discarded).[5] Moreover, worldviews behave like the additive product, as described in Section 9.8. Thus, this type system shows more aspects of logical linearity than any before it in this thesis.

Example 9.4.1. To demonstrate this type system, consider the scenario in which Alice and Bob buy candy in Section 9.2. This scenario can be modeled with polynomial resource functions via the functions in Figures 9.9 and 9.10 where a list type $L^a(\mathbb{1})$ represents a monetary allocation ("wallet") of a dollars. Thus, the function `aliceBuy` simulates Alice's use of the vending machine by taking \$5 from Alice and giving her back \$2 in change, corresponding to the type $L^5(\mathbb{1}) \to \mathbb{1} \sim L^2(\mathbb{1})$. Similarly, Bob's buying of \$2 of chocolate corresponds to `bobBuy`'s type of $L^2(\mathbb{1}) \to \mathbb{1} \sim L^0(\mathbb{1})$.

As described in Section 9.2, it should be possible for Alice and Bob to buy their chocolate with only \$5, leaving no money leftover. This cost corresponds to the function type $L^5(\mathbb{1}) \to \mathbb{1} \sim L^0(\mathbb{1})$. Each person buying their chocolate is modeled by the function `buyChoco` in Figure 9.10, and indeed, the function `buyChoco` can be given the desired type. Such a typing is witnessed by the energy comments in Figure 9.10 which match the bookkeeping of Table 9.1. Notationally, these comments use a slash to separate annotations in different worldviews. The comments also use `a` to indicate Alice's wallet contents (i.e., the linear energy on list `aliceWallet`), `b` to indicate Bob's wallet contents in the same way, and `w` to indicate an initial placeholder wallet before Alice and Bob actually carry the money. Thus, `a : x/y` means that Alice has x dollars (the list `aliceWallet` has x units of linear energy) in the first worldview and y in the second. Finally, the comments elide remainder annotations because there is no remainder.

The power of resource tunneling is exhibitted in Figure 9.10 in lines 4 and 5. Line 4 is where the first worldview justifies that Alice has enough money to buy from the vending machine, i.e., that the list `aliceWallet` has enough linear energy for the function `aliceBuy`. Then in line 5 the second worldview is the one where everyone is left with a sensible nonnegative amount of money, i.e., where the worldview is classically valid. Switching between the two worldviews

[4]In the demotion of Chapter 6, negative annotations are not an issue because they are guaranteed to be offset by other annotations present on the same variable.

[5]Of course, some forms of weakening and contraction are still present, like *Q-Sub* discarding energy and remainder contexts keeping the bindings of used variables. These are enough to capture the nonlinear nature of real programs, but somehow they do not suffocate the linearity of the overall typing discipline.

```
fun aliceBuy wallet = case wallet of
  | [] -> []
  | x::xs ->
    let _ = tick{5} in
    let tmp = vend xs in
    let _ = tick{-2} in
    tmp

fun bobBuy wallet = case wallet of
  | [] -> []
  | x::xs ->
    let _ = tick{2} in
    bob xs
```

Figure 9.9: Alice and Bob buying models

```
fun buyChoco wallet =                (* w:5/5 *)
  let aliceWallet = wallet in        (* a:5/3,  w:0/2 *)
  let bobWallet = wallet in          (* a:5/3,  b:0/2 *)
  let _ = aliceBuy aliceWallet in    (* a:2/0,  b:0/2 *)
  bobBuy bobWallet                   (* a:2/0,  b:-2/0 *)
```

Figure 9.10: Buying chocolate with energy annotations

$$\textsc{V-Unit}\ \frac{}{\langle\rangle : \mathbb{1}} \qquad \textsc{V-Fun'}\ \frac{V : \Gamma \qquad \Gamma \mid P \vdash \texttt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{a}|\vec{b}} \sigma \mid Q}{\texttt{C}(V;\ f,\ x.\,e) : \tau \xrightarrow{\vec{a}|\vec{b}} \sigma} \qquad \textsc{V-Pair}\ \frac{v_1 : \tau \qquad v_2 : \sigma}{\langle v_1,\ v_2\rangle : \tau \otimes \sigma}$$

$$\textsc{V-SumL}\ \frac{v : \tau}{\texttt{l}(v) : \tau \oplus \sigma} \qquad \textsc{V-SumR}\ \frac{v : \sigma}{\texttt{r}(v) : \tau \oplus \sigma} \qquad \textsc{V-Nil}\ \frac{}{[\,] : L(\tau)} \qquad \textsc{V-Cons}\ \frac{v_1 : \tau \qquad v_2 : L(\tau)}{v_1 :: v_2 : L(\tau)} \qquad \textsc{V-Leaf}\ \frac{}{\texttt{leaf} : T(\tau)}$$

$$\textsc{V-Node}\ \frac{v_1 : T(\tau) \qquad v_2 : \tau \qquad v_3 : T(\tau)}{\texttt{node}(v_1,\ v_2,\ v_3) : T(\tau)} \qquad \textsc{V-Context}\ \frac{\forall x \in \texttt{dom}(\Gamma).\, V(x) : \Gamma(x)}{V : \Gamma}$$

Figure 9.11: Quantum physicist's method value well-formedness rules

between lines 4 and 5 corresponds to reallocating the money from Alice to Bob, i.e., reallocating energy between the two lists.

This example also witnesses some of the trickiness of reasoning about reusable resources: If the order of the functions `aliceBuy` and `bobBuy` is reversed, the function is no longer be typable as $L^5(\mathbb{1}) \to \mathbb{1} \sim L^0(\mathbb{1})$ because the peak cost goes up to 7 units of linear energy. The consumption of partially reusable resources is not generally commutative, unlike the consumption of non-reusable resources like time.

9.4.2 Well-Formed Values

Only one minor change need be made with respect to the definition of well-formed values in AARA with the quantum physicit's method. That change is that function closures must admit typing derivations in the new type system, which now replaces classical annotations with multidimensional maps. Because I use a slightly different syntax for these multidimensional maps, I include a new rule for determining its well-formedness. Morally speaking, however, nothing has changed about the well-formedness of values since Figure 3.8; function closures still are only well-formed if they admit a typing derivation. For completeness, I provide the changed rule as *V-Fun'* in Figure 9.11 alongside all the other unchanged rules.

9.4.3 Potential Energy

The potential energy of a collection of worldviews is given the maximum energy across all worldviews. This definition is formalized in Figure 9.12, which extends the definition of the potential function to multidimensional annotations by building upon the definition of potential energy for classical annotations (Figure 3.9,Figure 6.6).

A key result of this definition of energy is that energy of a typing context with multidimensional anntoations can no longer be decomposed as the sum of the energies of each variable.

$$\Phi(V : \Gamma \mid P) = \max_{w \in \mathtt{wv}(P)} \Phi(V : \Gamma \mid P_w)$$

Figure 9.12: Energy across worldviews

This circumstance occurs because summation and maxima do not commute, $\sum_i \max_j a_{i,j} \neq \max_j \sum_i a_{i,j}$. In more concrete terms, one might distribute one unit of linear energy accross two copies of the same list as either $x : L^1(\tau)$, $y : L^0(\tau)$ or $x : L^0(\tau)$, $y : L^1(\tau)$, putting all energy on copy x or copy y, respectively. These differing energy allocations can be captured accross differing worldviews, wherein the maximum energy is always one unit of linear energy. However, the maximum energy for x across all worldviews plus the maximum energy for y is *two* units of linear energy, one from each list, which is double the intended amount. This same phenomenon occurs in quantum physics as *entanglement*, wherein a quantum state might not be decomposable into the states of its parts.

9.4.4 Soundness

The soundness of AARA with the quantum physicist's method is given in two parts as Lemma 9.4.1 and then Theorem 9.4.2. The reason for providing Lemma 9.4.1 first is to provide deeper insight into the behaviour of the system. In particular, this lemma shows that the principle of resource tunneling fundamentally underlies the entire system—as long as some worldview covers the peak cost, all can pay only the net cost. The soundness statement Theorem 9.4.2 then follows essentially as a corollary to Lemma 9.4.1.

Otherwise, the soundness statement itself is fundamentally no different than in previous chapters, except that it must adapt to the new presence of multidimensional annotation maps. It is still the case that the initial potential energy of the context bounds the peak cost of evalution, and the difference between initial and final energies bounds the net cost.

Lemma 9.4.1 (resource tunneling soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid P \vdash e : \tau \mid Q$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return value is well-formed)*
- $\exists w \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid P_w) \geq p$ *(initial energy bounds peak cost in some worldview)*
- $\forall u \in \mathtt{wv}(Q).\, \exists w \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid P_w) + q \geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q_u) + p$
  (every rem. worldview has some init. worldview s.t. difference in energy bounds net cost)

Proof. The soundness proof proceeds by lexicographic induction over the derivation of the evaluation judgment followed by the typing judgment.

This proof follows much the same way as that for remainder contexts' soundness, Theorem 5.4.1. Only a few cases here are of interest, which are those for the interesting typing rules pointed out in Section 9.4.1. In particular, the key behaviours of this type system are shown in the superposition/collapse rules and the application rule *Q-App*.

Now each case in the induction is given in more detail:

Q-SuperposeL This case deals with the structural typing rule *Q-SuperposeL* so that future considerations the typing judgment derivation need not consider the case that the derivation ends with application of *Q-SuperposeL*.

Suppose the last rule applied for the typing judgment is *Q-SuperposeL*.

$$\textsc{Q-SuperposeL}\quad \frac{\Gamma \mid P, u \mapsto P_w \vdash e : \tau \mid Q}{\Gamma \mid P \vdash e : \tau \mid Q}$$

Then the premiss of this rule holds by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid P, u \mapsto P_w \vdash e : \tau \mid Q$ to learn:

(1) $v : \tau$

(2) $\exists w' \in \mathtt{wv}(P, u \mapsto P_w).\, \Phi(V : \Gamma \mid (P, u \mapsto P_w)_{w'}) \geq p$

(3) $\forall u' \in \mathtt{wv}(Q).\, \exists w' \in \mathtt{wv}(P, u \mapsto P_w).$
$\Phi(V : \Gamma \mid (P, u \mapsto P_w)_{w'}) + q \geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q_{u'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from inequalities (2) and (3) by observing that if ever the quantification $\exists w' \in \mathtt{wv}(P, u \mapsto P_w)$ avoids picking a worldview from $\mathtt{wv}(P)$, then it picks worldview u, in which case it could equally well have picked $w \in \mathtt{wv}(P)$ because $(P, u \mapsto P_w)_u = P_w$. Thus, it suffices to pick $w' \in \mathtt{wv}(P)$.

Q-SuperposeR This case deals with the structural typing rule *Q-SuperposeR* so that future considerations the typing judgment derivation need not consider the case that the derivation ends with application of *Q-SuperposeR*.

Suppose the last rule applied for the typing judgment is *Q-SuperposeR*.

$$\textsc{Q-SuperposeR}\quad \frac{\Gamma \mid P \vdash e : \tau \mid Q}{\Gamma \mid P \vdash e : \tau \mid Q, u \mapsto Q_w}$$

Then the premiss of this rule holds by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid P \vdash e : \tau \mid Q$ to learn:

(1) $v : \tau$

(2) $\exists w' \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid (P)_{w'}) \geq p$

(3) $\forall u' \in \texttt{wv}(Q).\, \exists w' \in \texttt{wv}(P).$
$\Phi(V : \Gamma \mid (P)_{w'}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_{u'}) + p$

The well-formedness judgment (1) $v : \tau$ and peak cost bound (2) are both what this case needs, so only this case's net cost bound remain to be proven. This remaining cost bound can be obtained from inequality (3) by observing that (3) covers all worldviews of $(Q, u \mapsto Q_w)$ aside from u, in which case instantiating (3) at worldview w would suffice because $(Q, u \mapsto Q_w)_u = Q_w$.

Q-CollapseL This case deals with the structural typing rule *Q-CollapseL* so that future considerations the typing judgment derivation need not consider the case that the derivation ends with application of *Q-CollapseL*.

Suppose the last rule applied for the typing judgment is *Q-CollapseL*.

$$\textsc{Q-CollapseL}\quad \frac{\Gamma \mid P \vdash e : \tau \mid Q}{\Gamma \mid P, u \mapsto \vec{a} \vdash e : \tau \mid Q}$$

Then the premiss of this rule holds by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid P \vdash e : \tau \mid Q$ to learn:

(1) $v : \tau$
(2) $\exists w' \in \texttt{wv}(P).\, \Phi(V : \Gamma \mid (P)_{w'}) \geq p$
(3) $\forall u' \in \texttt{wv}(Q).\, \exists w' \in \texttt{wv}(P).$
$\Phi(V : \Gamma \mid (P)_{w'}) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_{u'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from inequalities (2) and (3) by observing the existential quantification $\exists w' \in \texttt{wv}(P)$ already suffices, so quantifying over a larger domain $\texttt{wv}(P, u \mapsto \vec{a})$ still suffices.

Q-CollapseL This case deals with the structural typing rule *Q-CollapseR* so that future considerations the typing judgment derivation need not consider the case that the derivation ends with application of *Q-CollapseR*.

Suppose the last rule applied for the typing judgment is *Q-CollapseR*.

$$\textsc{Q-CollapseR}\quad \frac{\Gamma \mid P \vdash e : \tau \mid Q, u \mapsto \vec{a}}{\Gamma \mid P \vdash e : \tau \mid Q}$$

Then the premiss of this rule holds by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid P \vdash e : \tau \mid Q, u \mapsto \vec{a}$ to learn:

(1) $v : \tau$

(2) $\exists w' \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid (P)_{w'}) \geq p$
(3) $\forall u' \in \mathtt{wv}(Q, u \mapsto \vec{a}).\, \exists w' \in \mathtt{wv}(P).$
$\Phi(V : \Gamma \mid (P)_{w'}) + q \geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q) + p$

The well-formedness judgment (1) $v : \tau$ and peak cost bound (2) are both what this case needs, so only this case's net cost bound remains to be proven. This remaining cost bound can be obtained from inequality (3) by observing the universal quantification $\forall u' \in \mathtt{wv}(Q, u \mapsto \vec{a})$ already suffices, so quantifying over a smaller domain $\mathtt{wv}(Q)$ still suffices.

Q-Sub This case deals with the structural typing rule *Q-Sub* so that future considerations of typing judgment derivation structure need not consider the case that the derivation ends with the application of *Q-Sub*.

Suppose the last rule applied for the typing judgment is *Q-Sub*.

$$\textsc{Q-Sub}\quad \frac{\Gamma \mid P' \vdash e : \tau \mid Q' \qquad P \geq P' \qquad Q \leq Q'}{\Gamma \mid P \vdash e : \tau \mid Q}$$

Then the premisses of this rule hold by inversion.

Because both $V : \Gamma$ and $V \vdash e \Downarrow v \mid (p, q)$ by assumption, the inductive hypothesis can be applied with $\Gamma \mid P' \vdash e : \tau \mid Q'$ to learn:

(1) $v : \tau$
(2) $\exists w' \in \mathtt{wv}(P').\, \Phi(V : \Gamma \mid P'_{w'}) \geq p$
(3) $\forall u' \in \mathtt{wv}(Q').\, \exists w' \in \mathtt{wv}(P').$
$\Phi(V : \Gamma \mid P'_{w'}) + q \geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q'_{u'}) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. These remaining cost bounds can be obtained from inequalities (2) and (3) by applying the pointwise monotonicity of potential energy (Lemma 3.4.5) alongside the pointwise annotation inequalities $P \geq P'$ and $Q \leq Q'$.

E-App Suppose the last rule applied for the evaluation judgment is *E-App*.

$$\textsc{E-App}\quad \frac{V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e) \vdash f\ x \Downarrow v \mid (p, q)}$$

Then this rule's premiss holds by inversion and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-App}\quad \frac{P_u \geq 0}{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(P, \lambda w.\, \vec{b}) \vdash f\ x : \sigma \mid \Upsilon_x^{x,\mathtt{arg}}(P, \lambda w.\, \vec{c})}$$

Because $(V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$ by assumption, the rule *V-Context* can be inverted to learn $\mathtt{C}(V';\ g,\ y.\,e) : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma$. Then further, the rule *V-Fun'* can be inverted to learn that this function body can be typed in some context Γ' where $V' : \Gamma'$. Using *V-Context*, one can then use this well-formedness judgment to derive

$$(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$$

Now inspect the derivation of the type of the function closure's body. Only structural rules (like *Q-Sub*) and *Q-Fun* can conclude a typing derivation for a function, and the application a structural rule itself requires another typing derivation for the same function. Thus it can be shown by induction that the typing derivation must conclude by the rule *Q-Fun* followed by some number of uses of structural rules. The typing derivation therefore contains the following rule application:

$$\textsc{Q-Fun}\quad \frac{\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid\ \lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{c})}{\Gamma' \mid P', Q' \vdash \mathtt{fun}\ g\ y\ =\ e : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid\ P', Q'}$$

This rule's premiss holds by inversion.

Each of the following judgments have now been found:

- $V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e) \vdash e \Downarrow v \mid (p, q)$
- $(V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma)$
- $\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma \mid \lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{b}) \vdash e : \sigma \mid\ \lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{c})$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \sigma$

(2) $\exists w' \in \mathtt{wv}(\lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{b}))$.
$\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid 0 \cdot P'_{w'}, [y/\mathtt{arg}]\vec{b}) \geq p$

(3) $\forall u' \in \mathtt{wv}(\lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{c})).\, \exists w' \in \mathtt{wv}(\lambda w.\,(0 \cdot P'_w, [y/\mathtt{arg}]\vec{b}))$.
$\Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e)) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid 0 \cdot P'_{w'}, [y/\mathtt{arg}]\vec{b}) + q$
$\geq \Phi((V', y \mapsto v', g \mapsto \mathtt{C}(V';\ g,\ y.\,e), \mathtt{ret} \mapsto v) : (\Gamma', y : \tau, g : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid 0 \cdot P'_{u'}, [y/\mathtt{arg}]\vec{c}) + p$

The well-formedness judgment (1) $v : \sigma$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, first simplify inequalities (2) and (3) into inequalities (4) and (5), respectively, by removing bindings that carry no potential energy in any worldview. This removal yields inequalities that are independent of worldview.

(4) $\Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) \geq p$

(5) $\Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + q \geq \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid [y/\mathtt{arg}]\vec{c}) + p$

Now let $r = \Phi((V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid P_u)$. Then the following inequalities confirm the peak cost bound at worldview $u \in \mathtt{wv}(P) = \mathtt{wv}(\Upsilon_x^{x,\mathtt{arg}}(P, \lambda w.\, \vec{b}))$:

$$
\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(P_u, \vec{b})) && \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon_x^{x,\mathtt{arg}}(P_u, \vec{b})) && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid P_u) + \Phi((\mathtt{arg} \mapsto v') : (\mathtt{arg} : \tau) \mid \vec{b}) && \textit{Lemma } 3.4.1 \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid P_u) + \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) && \textit{relabelling} \\
&\geq \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) && \textit{algebra},\ P_u \geq 0 \\
&\geq p && (4)
\end{aligned}
$$

Now let $r = \Phi((V, f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid P_{w'})$ for some arbitrary worldview $w' \in \mathtt{wv}(P)$. Then the following inequalities confirm the net cost bound, where both quantified worldviews are that same arbitrary w'. This quantification is valid for the domain because $\mathtt{wv}(P) = \mathtt{wv}(\Upsilon_x^{x,\mathtt{arg}}(P, \lambda w.\, \vec{b})) = \mathtt{wv}(\Upsilon_x^{x,\mathtt{arg}}(P, \lambda w.\, \vec{c}))$.

$$
\begin{aligned}
&\Phi((V, x \mapsto v', f \mapsto \mathtt{C}(V';\ g,\ y.\, e)) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(P_{w'}, \vec{b})) + q && \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid \Upsilon_x^{x,\mathtt{arg}}(P_{w'}, \vec{b})) + q && \textit{def} \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid P_{w'}) + \Phi((\mathtt{arg} \mapsto v') : (\mathtt{arg} : \tau) \mid \vec{b}) + q && \textit{Lemma } 3.4.1 \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid P_{w'}) + \Phi((y \mapsto v') : (y : \tau) \mid [y/\mathtt{arg}]\vec{b}) + q && \textit{relabelling} \\
&\geq r + \Phi((x \mapsto v') : (x : \tau) \mid P_{w'}) + \Phi((y \mapsto v', \mathtt{ret} \mapsto v) : (y : \tau, \mathtt{ret} : \sigma) \mid [y/\mathtt{arg}]\vec{c}) + p && (5) \\
&= r + \Phi((x \mapsto v') : (x : \tau) \mid P_{w'}) + \Phi((\mathtt{arg} \mapsto v', \mathtt{ret} \mapsto v) : (\mathtt{arg} : \tau, \mathtt{ret} : \sigma) \mid \vec{c}) + p && \textit{relabelling} \\
&= r + \Phi((x \mapsto v', \mathtt{ret} \mapsto v) : (x : \tau, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(P_{w'}, \vec{c})) + p && \textit{Lemma } 3.4.1 \\
&= \Phi((V, x \mapsto v', f \mapsto \mathtt{C}(C';\ g,\ y.\, e), \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, f : \tau \xrightarrow{\vec{b}|\vec{c}} \sigma, \mathtt{ret} : \sigma) \mid \Upsilon_x^{x,\mathtt{arg}}(P_{w'}, \vec{c})) + p && \textit{def}
\end{aligned}
$$

E-Fun Suppose the last rule applied for the evaluation judgment is *E-Fun*.

$$
\textsc{E-Fun} \quad \frac{}{V \vdash \mathtt{fun}\ f\ x\ =\ e \Downarrow \mathtt{C}(V;\ f,\ x.\, e) \mid (0, 0)}
$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$
\textsc{Q-Fun} \quad \frac{\Gamma, x : \tau, f : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid \lambda w.\, (0 \cdot P_w, [x/\mathtt{arg}]\vec{c}) \vdash e : \sigma \mid \lambda w.\, (0 \cdot P_w, [x/\mathtt{arg}]\vec{d})}{\Gamma \mid P, Q \vdash \mathtt{fun}\ f\ x\ =\ e : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma \mid P, Q}
$$

The assumed typing judgment for the expression being evaluated therefore takes the form of this rule's conclusion.

Because $\mathtt{C}(V;\ f,\ x.\, e) : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma$ follows from *V-Fun'* and the assumed typing judgment, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in

P, Q's classically valid worldview, the peak cost bound is satisfied in that worldview. And finally, because the initial and remainder annotations are identical and functions carry no potential energy (Figure 3.9), the net cost bound is also satisfied with the following equality, no matter the choice of worldview $w \in \mathtt{wv}(P, Q)$:

$$\Phi(V : \Gamma \mid (P, Q)_w) = \Phi((V, \mathtt{ret} \mapsto \mathtt{C}(V;\ f,\ x.\, e)) : (\Gamma, \mathtt{ret} : \tau \xrightarrow{\vec{c}|\vec{d}} \sigma) \mid (P, Q)_w)$$

E-Let Suppose the last rule applied for the evaluation judgment is *E-Let*.

$$\textsc{E-Let}\quad \frac{V \vdash e_1 \Downarrow v' \mid (p, q) \qquad V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)}{V \vdash \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 \Downarrow v \mid (p + \max(0, r - q), s + \max(0, q - r))}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-Let}\quad \frac{\Gamma \mid P \vdash e_1 : \sigma \mid\ R \qquad \Gamma, x : \sigma \mid [x/\mathtt{ret}]R \vdash e_2 : \tau \mid\ Q, S \qquad S \geq 0}{\Gamma \mid P \vdash \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \tau \mid\ Q}$$

The premisses of both of these rules hold by inversion.

Because $V : \Gamma$ holds by assumption, the inductive hypothesis can be applied with the judgments $V \vdash e_1 \Downarrow v' \mid (p, q)$ and $\Gamma \mid P \vdash e_1 : \sigma \mid\ R$. to learn:

(1) $v' : \sigma$
(2) $\exists w \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid P_{w'}) \geq p$
(3) $\forall w' \in \mathtt{wv}(R).\, \exists w \in \mathtt{wv}(P).\, \Phi(V : \Gamma \mid P_w) + q \geq \Phi((V, \mathtt{ret} \mapsto v') : (\Gamma, \mathtt{ret} : \sigma) \mid R_{w'}) + p$

Because $v' : \sigma$ holds as (1) from the previous induction and both $V, x \mapsto v' \vdash e_2 \Downarrow v \mid (r, s)$ and $\Gamma, x : \sigma \mid [x/\mathtt{ret}]R \vdash e_2 : \tau \mid\ Q, S$ hold from inversion, the inductive hypothesis can be applied again to learn:

(4) $v : \tau$
(5) $\exists w' \in \mathtt{wv}(R).\, \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\mathtt{ret}]R_{w'}) \geq r$
(6) $\forall u \in \mathtt{wv}(Q, S).\, \exists w' \in R.$
$\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\mathtt{ret}]R_{w'}) + s \geq \Phi((V, x \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma) \mid (Q, S)_u) + r$

The well-formedness judgment (4) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. To do so, proceed by cases on whether $q \geq r$.

If $q \geq r$, then the cost behaviour to consider is $(p, s + (q - r))$. Then (2) confirms the peak cost bound, and the following inequalities confirm the net cost bound in some worldview $w \in \mathtt{wv}(P)$ given arbitrary $u \in \mathtt{wv}(Q)$:

$$\begin{aligned}
&\Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q_u) + p \\
\leq\ &\Phi((V, x \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma, \mathtt{ret} : \tau) \mid (Q, S)_u) + p && S \geq 0 \\
\leq\ &\Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\mathtt{ret}]R_{w'}) + p + s - r && (6) \\
=\ &\Phi((V, \mathtt{ret} \mapsto v') : (\Gamma, \mathtt{ret} : \sigma) \mid R_{w'}) + p + s - r && \textit{relabelling} \\
\leq\ &\Phi(V : \Gamma \mid P_w) + s + (q - r) && (3)
\end{aligned}$$

If $q < r$, then the cost behaviour to consider is $(p + (r - q), s)$, and $q \neq \infty$ (so can be subtracted). Then the following inequalities confirm the peak cost bound in some worldview $w \in \mathtt{wv}(P)$:

$$\begin{aligned}
&p + (r - q) \\
&\leq \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\mathtt{ret}]R_{w'}) + p - q && (5)\\
&= \Phi((V, \mathtt{ret} \mapsto v') : (\Gamma, \mathtt{ret} : \sigma) \mid R_{w'}) + p - q && \textit{relabelling}\\
&\leq \Phi(V : \Gamma \mid P_w) && (3)
\end{aligned}$$

And finally, the following inequalities confirm the net cost bound in some worldview $w \in \mathtt{wv}(P)$ given arbitrary $u \in \mathtt{wv}(Q)$:

$$\begin{aligned}
&\Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q_u) + p + (r - q) \\
&\leq \Phi((V, x \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma, \mathtt{ret} : \tau) \mid (Q, S)_u) + p + (r - q) && S \geq 0\\
&\leq \Phi((V, x \mapsto v') : (\Gamma, x : \sigma) \mid [x/\mathtt{ret}]R_{w'}) + s + p - q && (6)\\
&= \Phi((V, \mathtt{ret} \mapsto v') : (\Gamma, \mathtt{ret} : \sigma) \mid R_{w'}) + s + p - q && \textit{relabelling}\\
&\leq \Phi(V : \Gamma \mid P_w) + s && (3)
\end{aligned}$$

E-Nont Suppose the last rule applied for the evaluation judgment is *E-Nont*.

$$\frac{}{V \vdash e \Downarrow \bullet \mid (0, \infty)}\ \textsc{E-Nont}$$

Then $p = 0$, $q = \infty$, and $v = \bullet$. Because $\bullet : \tau$ by *V-Nont*, the needed well-formedness judgment holds. The potential energy of a quantumly valid initial context P is always nonnegative because it must be at least as much energy as bestowed by the classical (nonnegative) annotation in P, so therefore the peak cost bound is satisfied. And finally, because ∞ is greater than or equal to anything, the net cost bound also satisfied.

E-Tick Suppose the last rule applied for the evaluation judgment is *E-Tick*.

$$\frac{}{V \vdash \mathtt{tick}\{r\} \Downarrow \langle\rangle \mid (\max(0, r), \max(0, -r))}\ \textsc{E-Tick}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{P(\mathtt{c}) = Q(\mathtt{c}) + \lambda w.\, r \qquad \forall i \neq \mathtt{c}.\, P(i) = Q(i)}{\Gamma \mid P \vdash \mathtt{tick}\{r\} : \mathbb{1} \mid Q}\ \textsc{Q-Tick}$$

The premisses of this rule hold by inversion.

Because $\langle\rangle : \mathbb{1}$ by *V-Unit*, the needed well-formedness judgment holds.

Then the the peak cost bound can be found as follows. First, in P's classically valid worldview w, the nonnegativity of P_w ensures $\Phi(V : \Gamma \mid P_w) \geq 0$. Then in Q's classically valid worldview u:

$$\begin{aligned}
\Phi(V : \Gamma \mid P_u) &\geq P_u(\mathtt{c}) && \mathit{def} \\
&= Q_u(\mathtt{c}) + r && P(\mathtt{c}) = Q(\mathtt{c}) + \lambda w.\, r \\
&\geq r && Q_u \geq 0
\end{aligned}$$

Thus, some $w' \in \{u, w\} \subseteq \mathtt{wv}(P)$ ensures $\Phi(V : \Gamma \mid P_{w'}) \geq \max(0, r)$.

Finally, let $s = \sum_{x \in \mathtt{dom}(\Gamma)} \Phi(V(x) : \Gamma(x) \mid \lambda i.\, P_w(x.i))$ for an arbitrary worldview $w \in \mathtt{wv}(P) = \mathtt{wv}(Q)$. Then the following inequalities confirm the net cost bound.

$$\begin{aligned}
&\Phi(V : \Gamma \mid P_w) + \max(0, -r) \\
&= P_w(\mathtt{c}) + s + \max(0, -r) && \mathit{def} \\
&= Q_w(\mathtt{c}) + r + s + \max(0, -r) && P(\mathtt{c}) = Q(\mathtt{c}) + \lambda w.\, r \\
&= Q_w(\mathtt{c}) + s + \max(0, r) && \mathit{algebra} \\
&= Q_w(\mathtt{c}) + s + \Phi(\langle\rangle : \mathbb{1} \mid \cdot) + \max(0, r) && 0\ \mathit{potential} \\
&= \Phi((V, \mathtt{ret} \mapsto \langle\rangle) : (\Gamma, \mathtt{ret} : \mathbb{1}) \mid Q_w) + \max(0, r) && \mathit{def}
\end{aligned}$$

E-Var Suppose the last rule applied for the evaluation judgment is *E-Var*

$$\frac{\textsc{E-Var}}{V, x \mapsto v \vdash x \Downarrow v \mid (0, 0)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{Q-Var}}{\Gamma, x : \tau \mid \curlyvee_x^{x,\mathtt{ret}}(P) \vdash x : \tau \mid P}$$

Then $p = q = 0$ and $V, x \mapsto v : \Gamma, x : \tau$. Because $v : \tau$ follows from inverting *V-Context*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in $\curlyvee_x^{x,\mathtt{ret}}(P)$'s classically valid worldview, the peak cost bound is satisfied in that worldview. And finally, because sharing perfectly conserves potential (Lemma 3.4.1), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \mathtt{wv}(\curlyvee_x^{x,\mathtt{ret}}(P)) = \mathtt{wv}(P)$:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \curlyvee_x^{x,\mathtt{ret}}(P)_w) = \Phi((V, x \mapsto v, \mathtt{ret} \mapsto v) : (\Gamma, x : \tau, \mathtt{ret} : \tau) \mid P_w)$$

E-Pair Suppose the last rule applied for the evaluation judgment is *E-Pair*.

$$\frac{\textsc{E-Pair}}{V, x \mapsto v_1, y \mapsto v_2 \vdash \langle x,\, y\rangle \Downarrow \langle v_1,\, v_2\rangle \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\text{Q-PAIR}}{\Gamma, x : \tau, y : \sigma \mid \curlyvee_{x}^{x,\mathtt{ret.1^{st}}}(\curlyvee_{y}^{y,\mathtt{ret.2^{nd}}}(P)) \vdash \langle x,\, y\rangle : \tau \otimes \sigma \mid\ P}$$

Because $\langle v_1,\, v_2\rangle : \tau \otimes \sigma$ follows from *V-Pair* and the assumed well-formedness judgment $(V, x_1 \mapsto v_1, x_2 \mapsto v_2) : (\Gamma, x : \tau, y : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in $\curlyvee_{x}^{x,\mathtt{ret.1^{st}}}(\curlyvee_{y}^{y,\mathtt{ret.2^{nd}}}(P))$'s classically valid worldview, the peak cost bound is satisfied in that worldview. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a pair with a classical annotation is the sum of its parts' (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \mathtt{wv}(\curlyvee_{x}^{x,\mathtt{ret.1^{st}}}(\curlyvee_{y}^{y,\mathtt{ret.2^{nd}}}(P))) = \mathtt{wv}(P)$:

$$\Phi(V, x \mapsto v_1, y \mapsto v_2 : \Gamma, x : \tau, y : \sigma \mid \curlyvee_{x}^{x,\mathtt{ret.1^{st}}}(\curlyvee_{y}^{y,\mathtt{ret.2^{nd}}}(P))_w)$$
$$= \Phi(V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto \langle v_1,\, v_2\rangle : \Gamma, x : \tau, y : \sigma, \mathtt{ret} : \tau \otimes \sigma \mid P_w)$$

E-CaseP Suppose the last rule applied for the evaluation judgment is *E-CaseP*.

$$\text{E-CASEP}\quad \frac{V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)}{V, x \mapsto \langle v_1,\, v_2\rangle \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\, z\rangle \to e \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\text{Q-CASEP}\quad \frac{\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid P \vdash e : \tau \mid\ Q}{\Gamma, x : \sigma \otimes \rho \mid \curlyvee_{x.1^{\mathrm{st}}}^{y,x.1^{\mathrm{st}}}(\curlyvee_{x.2^{\mathrm{nd}}}^{z,x.2^{\mathrm{nd}}}(P)) \vdash \mathtt{case}\ x\ \mathtt{of}\ \langle y,\, z\rangle \to e : \tau \mid\ \curlyvee_{x.1^{\mathrm{st}}}^{y,x.1^{\mathrm{st}}}(\curlyvee_{x.2^{\mathrm{nd}}}^{z,x.2^{\mathrm{nd}}}(Q))}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho)$ by assumption, the rule *V-Context* can be inverted to learn $\langle v_1,\, v_2\rangle : \sigma \otimes \rho$. Then further, the rule *V-Pair* can be inverted to learn both $v_1 : \sigma$ and $v_2 : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2 \vdash e \Downarrow v \mid (p, q)$
- $(V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho)$
- $\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho \mid P \vdash e : \tau \mid\ Q$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \mathtt{wv}(P).\, \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid P_w) \geq p$

(3) $\forall u \in \texttt{wv}(Q).\, \exists w \in \texttt{wv}(P).$
$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid P_w) + q$
$\geq \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid Q_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a pair with a classical annotation is the sum of its parts' (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho) \mid P_w)$$

$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle) : (\Gamma, x : \sigma \otimes \rho) \mid \curlyvee^{y, x.1^{\text{st}}}_{x.1^{\text{st}}}(\curlyvee^{z, x.2^{\text{nd}}}_{x.2^{\text{nd}}}(P))_w)$$

$$\Phi((V, x \mapsto \langle v_1,\, v_2\rangle, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, y : \sigma, z : \rho, \texttt{ret} : \tau) \mid Q_u)$$

$$= \Phi((V, x \mapsto \langle v_1,\, v_2\rangle, \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \otimes \rho, \texttt{ret} : \tau) \mid \curlyvee^{y, x.1^{\text{st}}}_{x.1^{\text{st}}}(\curlyvee^{z, x.2^{\text{nd}}}_{x.2^{\text{nd}}}(Q))_u)$$

E-SumL Suppose the last rule applied for the evaluation judgment is *E-SumL*.

$$\textsc{E-SumL}\quad \frac{}{V, x \mapsto v \vdash \texttt{l}(x) \Downarrow \texttt{l}(v) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-SumL}\quad \frac{}{\Gamma, x : \tau \mid \curlyvee^{x, \texttt{ret.l}}_{x}(P) \vdash \texttt{l}(x) : \tau \oplus \sigma \mid P, Q}$$

Because $\texttt{l}(v) : \tau \oplus \sigma$ follows from *V-SumL* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \tau)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in $\curlyvee^{x, \texttt{ret.l}}_{x}(P)$'s classically valid worldview, the peak cost bound is satisfied. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant with a classical annotation is that of its tagged value (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \texttt{wv}(\curlyvee^{x, \texttt{ret.l}}_{x}(P)) = \texttt{wv}(P, Q)$:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \tau) \mid \curlyvee^{x, \texttt{ret.l}}_{x}(P)_w)$$

$$= \Phi((V, x \mapsto v, \texttt{ret} \mapsto \texttt{l}(v)) : (\Gamma, x : \tau, \texttt{ret} \mapsto \tau \oplus \sigma) \mid (P, Q)_w)$$

E-SumR Suppose the last rule applied for the evaluation judgment is *E-SumR*.

$$\dfrac{\text{E-SUMR}}{V, x \mapsto v \vdash \mathtt{r}(x) \Downarrow \mathtt{r}(v) \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\dfrac{\text{Q-SUMR}}{\Gamma, x : \sigma \mid \curlyvee_{x}^{x,\mathtt{ret.r}}(P) \vdash \mathtt{r}(x) : \tau \oplus \sigma \mid P, Q}$$

Because $\mathtt{r}(v) : \tau \oplus \sigma$ follows from *V-SumR* and the assumed well-formedness judgment $(V, x \mapsto v) : (\Gamma, x : \sigma)$, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in $\curlyvee_{x}^{x,\mathtt{ret.r}}(P)$'s classically valid worldview, the peak cost bound is satisfied in that worldview. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant with a classical annotation is that of its tagged value (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \mathtt{wv}(\curlyvee_{x}^{x,\mathtt{ret.r}}(P)) = \mathtt{wv}(P, Q)$:

$$\Phi((V, x \mapsto v) : (\Gamma, x : \sigma) \mid \curlyvee_{x}^{x,\mathtt{ret.r}}(P)_w)$$

$$= \Phi((V, x \mapsto v, \mathtt{ret} \mapsto \mathtt{r}(v)) : (\Gamma, x : \sigma, \mathtt{ret} \mapsto \tau \oplus \sigma) \mid (P, Q)_w)$$

E-CaseS-L Suppose the last rule applied for the evaluation judgment is *E-CaseS-L*.

$$\text{E-CASES-L}\quad \dfrac{V, x \mapsto \mathtt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)}{V, x \mapsto \mathtt{l}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\text{Q-CASES}\quad \dfrac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid P, Q, R' \vdash e_1 : \tau \mid S, T, U' \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid P, Q', R \vdash e_2 : \tau \mid S, T', U\end{array}}{\Gamma, x : \sigma \oplus \rho \mid P, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(Q), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(R) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid S, \curlyvee_{x.\mathtt{l}}^{x.\mathtt{l},y}(T), \curlyvee_{x.\mathtt{r}}^{x.\mathtt{r},z}(U)}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{l}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \sigma$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{l}(v'), y \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma)$

- $\Gamma, x : \sigma \oplus \rho, y : \sigma \mid P, Q, R' \vdash e_1 : \tau \mid S, T, U'$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \mathtt{wv}(P, Q, R').\, \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid (P, Q, R')_w) \geq p$
(3) $\forall u \in \mathtt{wv}(S, T, U').\, \exists w \in \mathtt{wv}(P, Q, R').$
$\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid (P, Q, R')_w) + q$
$\geq \Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid (S, T, U')_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant with a classical annotation is that of its tagged value (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\begin{aligned}&\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v') : (\Gamma, x : \sigma \oplus \rho, y : \sigma) \mid (P, Q, R')_w)\\ &= \Phi((V, x \mapsto \mathtt{l}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid (P, \curlyvee^{x.\mathtt{l},y}_{x.\mathtt{l}}(Q), \curlyvee^{x.\mathtt{r},z}_{x.\mathtt{r}}(R))_w)\end{aligned}$$

$$\begin{aligned}&\Phi((V, x \mapsto \mathtt{l}(v'), y \mapsto v', \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, y : \sigma, \mathtt{ret} : \tau) \mid (S, T, U')_u)\\ &= \Phi((V, x \mapsto \mathtt{l}(v'), \mathtt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \mathtt{ret} : \tau) \mid (S, \curlyvee^{x.\mathtt{l},y}_{x.\mathtt{l}}(T), \curlyvee^{x.\mathtt{r},z}_{x.\mathtt{r}}(U))_u)\end{aligned}$$

E-CaseS-R Suppose the last rule applied for the evaluation judgment is *E-CaseS-R*.

E-CASES-R
$$\frac{V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_2 \Downarrow v \mid (p, q)}{V, x_s \mapsto \mathtt{r}(v') \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

Q-CASES
$$\frac{\begin{array}{c}\Gamma, x : \sigma \oplus \rho, y : \sigma \mid P, Q, R' \vdash e_1 : \tau \mid S, T, U' \\ \Gamma, x : \sigma \oplus \rho, z : \rho \mid P, Q', R \vdash e_2 : \tau \mid S, T', U\end{array}}{\Gamma, x : \sigma \oplus \rho \mid P, \curlyvee^{x.\mathtt{l},y}_{x.\mathtt{l}}(Q), \curlyvee^{x.\mathtt{r},z}_{x.\mathtt{r}}(R) \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{l}(y) \to e_1 \mid \mathtt{r}(z) \to e_2 : \tau \mid S, \curlyvee^{x.\mathtt{l},y}_{x.\mathtt{l}}(T), \curlyvee^{x.\mathtt{r},z}_{x.\mathtt{r}}(U)}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto \mathtt{r}(v')) : (\Gamma, x : \sigma \oplus \rho)$ by assumption, the rule *V-Context* can be inverted to learn $v' : \sigma \oplus \rho$. Then further, the rule *V-SumL* can be inverted to learn $v' : \rho$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$$

Each of the following judgments have now been found:

- $V, x \mapsto \mathtt{r}(v'), z \mapsto v' \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, x \mapsto \mathtt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho)$

- $\Gamma, x : \sigma \oplus \rho, z : \rho \mid P, Q', R \vdash e_2 : \tau \mid\ S, T', U$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \texttt{wv}(P, Q', R).\, \Phi((V, x \mapsto \texttt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid (P, Q', R)_w) \geq p$
(3) $\forall u \in \texttt{wv}(S, T', U).\, \exists w \in \texttt{wv}(P, Q', R).$
$\Phi((V, x \mapsto \texttt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid (P, Q', R)_w) + q$
$\geq \Phi((V, x \mapsto \texttt{r}(v'), z \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \texttt{ret} : \tau) \mid (S, T', U)_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and the potential energy of a variant with a classical annotation is that of its tagged value (Figure 3.9), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto \texttt{r}(v'), z \mapsto v') : (\Gamma, x : \sigma \oplus \rho, z : \rho) \mid (P, Q', R)_w)$$

$$= \Phi((V, x \mapsto \texttt{r}(v')) : (\Gamma, x : \sigma \oplus \rho) \mid (P, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(Q), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(R))_w)$$

$$\Phi((V, x \mapsto \texttt{r}(v'), z \mapsto v', \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, z : \rho, \texttt{ret} : \tau) \mid (S, T', U)_u)$$

$$= \Phi((V, x \mapsto \texttt{r}(v'), \texttt{ret} \mapsto v) : (\Gamma, x : \sigma \oplus \rho, \texttt{ret} : \tau) \mid (S, \curlyvee_{x.\texttt{l}}^{x.\texttt{l},y}(T), \curlyvee_{x.\texttt{r}}^{x.\texttt{r},z}(U))_u)$$

E-Nil Suppose the last rule applied for the evaluation judgment is *E-Nil*.

$$\frac{}{V \vdash [\,] \Downarrow [\,] \mid (0, 0)}\ \textsc{E-Nil}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{}{\Gamma \mid P \vdash [\,] : L(\tau) \mid\ P, Q}\ \textsc{Q-Nil}$$

Because $[\,] : L(\tau)$ follows from *V-Nil*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in P's classically valid worldview, the peak cost bound is satisfied in that worldview. And finally, because the initial and remainder annotations are identical except for the empty list annotations $\vec{b}$ and empty lists carry no energy regardless of annotation (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \texttt{wv}(P) = \texttt{wv}(P, Q)$:

$$\Phi(V : \Gamma \mid P_w) = \Phi((V, \texttt{ret} \mapsto [\,]) : (\Gamma, \texttt{ret} : L(\tau)) \mid (P, Q)_w)$$

E-Cons Suppose the last rule applied for the evaluation judgment is *E-Cons*.

$$\textsc{E-Cons}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2 \vdash x :: y \Downarrow v_1 :: v_2 \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-Cons}\quad \frac{}{\Gamma, x : \tau, y : L(\tau) \mid \curlyvee_{x}^{x,x'}(\curlyvee_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(P))) \vdash x :: y : L(\tau) \mid\ P}$$

Because $v_1 :: v_2 \,:\, L(\tau)$ follows from *V-Cons* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative in $\curlyvee_{x}^{x,x'}(\curlyvee_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(P)))$'s classically valid worldview, the peak cost bound is satisfied in that worldview. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 6.4.1), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \mathtt{wv}(\curlyvee_{x}^{x,x'}(\curlyvee_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(P)))) = \mathtt{wv}(P)$:

$$\begin{aligned}&\Phi((V, x \mapsto v_1, y \mapsto v_2) : (\Gamma, x : \tau, y : L(\tau)) \mid (\curlyvee_{x}^{x,x'}(\curlyvee_{y}^{y,y'}(\overset{A}{\lhd}{}^{\mathtt{ret}}_{x',y'}(P))))_w)\\ =\ &\Phi((V, x \mapsto v_1, y \mapsto v_2, \mathtt{ret} \mapsto v_1 :: v_2) : (\Gamma, x : \tau, y : L(\tau), \mathtt{ret} : L(\tau)) \mid P_w)\end{aligned}$$

E-CaseL-Nil Suppose the last rule applied for the evaluation judgment is *E-CaseL-Nil*.

$$\textsc{E-CaseL-Nil}\quad \frac{V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)}{V, x \mapsto [\,] \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p,q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-CaseL}\quad \frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S'\\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(P,Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(R,S)\end{array}}{\Gamma, x : L(\sigma) \mid P, \curlyvee_{x}^{x,x'}(Q) \vdash \mathtt{case}\ x\ \mathtt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid\ R, \curlyvee_{x}^{x,x'}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, x \mapsto [\,] \vdash e_1 \Downarrow v \mid (p,q)$
- $(V, x \mapsto [\,]) : (\Gamma, x : L(\sigma))$
- $\Gamma, x : L(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S'$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \texttt{wv}(P, Q').\, \Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid (P, Q')_w) \geq p$
(3) $\forall u \in \texttt{wv}(R, S').\, \exists w \in \texttt{wv}(P, Q').$
$\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid (P, Q')_w) + q$
$\geq \Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid (R, S')_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because empty lists carry no potential energy regardless of annotation (Figure 3.9) both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid (P, Q')_w) = \Phi((V, x \mapsto [\,]) : (\Gamma, x : L(\sigma)) \mid (P, \Upsilon^{x,x'}_{x}(Q))_w)$$

$$\begin{aligned} &\Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid (R, S')_u) \\ = &\Phi((V, x \mapsto [\,], \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid (R, \Upsilon^{x,x'}_{x}(S))_u) \end{aligned}$$

E-CaseL-Cons Suppose the last rule applied for the evaluation judgment is *E-CaseL-Cons*.

$$\textsc{E-CaseL-Cons}\quad \frac{V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)}{V, x \mapsto v_1 :: v_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-CaseL}\quad \frac{\begin{array}{c}\Gamma, x : L(\sigma) \mid P, Q' \vdash e_1 : \tau \mid R, S' \\ \Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(R, S)\end{array}}{\Gamma, x : L(\sigma) \mid P, \Upsilon^{x,x'}_{x}(Q) \vdash \texttt{case}\ x\ \texttt{of}\ [\,] \to e_1 \mid y :: z \to e_2 : \tau \mid R, \Upsilon^{x,x'}_{x}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 : L(\sigma)$. Then further, the rule *V-Cons* can be inverted to learn both $v_1 : \sigma$ and $v_2 : L(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$$

Each of the following judgments has now been found:

- $V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma))$
- $\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{x'}_{y,z}(R, S)$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\exists w \in \texttt{wv}(\overset{A}{\lhd}{}^{x'}_{y,z}(P,Q))$.

$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid (\overset{A}{\lhd}{}^{x'}_{y,z}(P,Q))_w) \geq p$

(3) $\forall u \in \texttt{wv}(\overset{A}{\lhd}{}^{x'}_{y,z}(R,S)).\, \exists w \in \texttt{wv}(\overset{A}{\lhd}{}^{x'}_{y,z}(P,Q))$.

$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(P,Q)_w) + q$

$\geq \Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \overset{A}{\lhd}{}^{x'}_{y,z}(R,S)_u) + p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a list (Lemma 6.4.1), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma)) \mid \lhd^{x'}_{y,z}(P,Q)_w)$$

$$= \Phi((V, x \mapsto v_1 :: v_2) : (\Gamma, x : L(\sigma)) \mid (P, \curlyvee^{x,x'}_{x}(Q))_w)$$

$$\Phi((V, x \mapsto v_1 :: v_2, y \mapsto v_1, z \mapsto v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), y : \sigma, z : L(\sigma), \texttt{ret} : \tau) \mid \lhd^{x'}_{y,z}(R,S)_u)$$

$$= \Phi((V, x \mapsto v_1 :: v_2, \texttt{ret} \mapsto v) : (\Gamma, x : L(\sigma), \texttt{ret} : \tau) \mid (R, \curlyvee^{x,x'}_{x}(S))_u)$$

E-Leaf Suppose the last rule applied for the evaluation judgment is *E-Leaf*.

$$\textsc{E-Leaf}\quad \frac{}{V \vdash \texttt{leaf} \Downarrow \texttt{leaf} \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{Q-Leaf}\quad \frac{}{\Gamma \mid P \vdash \texttt{leaf} : T(\tau) \mid P, Q}$$

Because $\texttt{leaf} : T(\tau)$ follows from *V-Leaf*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in P's classically valid worldview, the peak cost bound is satisfied in that worldview. And finally, because the initial and remainder annotations are identical except for the leaf annotations $\vec{b}$ and leaves carry no energy regardless of annotation (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \texttt{wv}(P) = \texttt{wv}(P,Q)$:

$$\Phi(V : \Gamma \mid P_w) = \Phi((V, \texttt{ret} \mapsto \texttt{leaf}) : (\Gamma, \texttt{ret} : T(\tau)) \mid (P,Q)_w)$$

E-Node Suppose the last rule applied for the evaluation judgment is *E-Node*.

$$\frac{\textsc{E-Node}}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\ y,\ z) \Downarrow \texttt{node}(v_1,\ v_2,\ v_3) \mid (0, 0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{Q-Node}}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}\,{}^{\texttt{ret}}_{x',y',z'}(P)))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid\ P}$$

Because $\texttt{node}(v_1,\ v_2,\ v_3) : T(\tau)$ follows from *V-Node* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative in $\curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}\,{}^{\texttt{ret}}_{x',y',z'}(P))))$'s classically valid worldview, the peak cost bound is satisfied in that worldview. Finally, because sharing conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 6.4.1), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \texttt{wv}(\curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}\,{}^{\texttt{ret}}_{x',y',z'}(P))))) = \texttt{wv}(P)$:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}\,{}^{\texttt{ret}}_{x',y',z'}(P))))_w)$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau), \texttt{ret} : T(\tau)) \mid P_w)$$

E-CaseT-Leaf Suppose the last rule applied for the evaluation judgment is *E-CaseT-Leaf*.

$$\frac{\textsc{E-CaseT-Leaf} \qquad V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{leaf} \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\frac{\textsc{Q-CaseT} \qquad \begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}\,{}^{t'}_{x,y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}\,{}^{t'}_{x,y,z}(R, S)\end{array}}{\Gamma, t : T(\sigma) \mid P, \curlyvee_t^{t,t'}(Q) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ R, \curlyvee_t^{t,t'}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$

- $\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S'$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \mathtt{wv}(P, Q').\, \Phi((V, t \mapsto \mathtt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) \geq p$
(3) $\forall u \in \mathtt{wv}(R, S').\, \exists w \in \mathtt{wv}(P, Q').$
$\Phi((V, t \mapsto \mathtt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) + q$
$\geq \Phi((V, t \mapsto \mathtt{leaf}, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \mathtt{ret} : \tau) \mid (R, S')_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because leaves carry no potential energy regardless of annotation (Figure 3.9) both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \mathtt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) = \Phi((V, t \mapsto \mathtt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, \curlyvee_t^{t,t'}(Q))_w)$$

$$\begin{aligned}&\Phi((V, t \mapsto \mathtt{leaf}, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \mathtt{ret} : \tau) \mid (R, S')_u) \\ &= \Phi((V, t \mapsto \mathtt{leaf}, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \mathtt{ret} : \tau) \mid (R, \curlyvee_t^{t,t'}(S))_u)\end{aligned}$$

E-CaseT-Node Suppose the last rule applied for the evaluation judgment is *E-CaseT-Node*.

E-CaseT-Node
$$\frac{V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)}{V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3) \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{leaf} \to e_1 \mid \mathtt{node}(x,\, y,\, z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

Q-CaseT
$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(R, S)\end{array}}{\Gamma, t : T(\sigma) \mid P, \curlyvee_t^{t,t'}(Q) \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{leaf} \to e_1 \mid \mathtt{node}(x,\, y,\, z) \to e_2 : \tau \mid\ R, \curlyvee_t^{t,t'}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3)) : (\Gamma, t : T(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 v_3 : T(\sigma)$. Then further, the rule *V-Node* can be inverted to learn all of $v_1 : T(\sigma)$, $v_2 : \sigma$, and $v_3 : T(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$$

Each of the following judgments has now been found:

- $V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \mathtt{node}(v_1,\, v_2,\, v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$
- $\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P, Q) \vdash e_2 : \tau \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(R, S)$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\exists w \in \mathtt{wv}(\overset{A}{\lhd}{}^{t'}_{x,y,z}(P,Q))$.
$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P,Q)_w)$
$\geq p$

(3) $\forall u \in \mathtt{wv}(\overset{A}{\lhd}{}^{t'}_{x,y,z}(R,S)).\ \exists w \in \mathtt{wv}(\overset{A}{\lhd}{}^{t'}_{x,y,z}(P,Q))$.
$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P,Q)_w) + q$
$\geq \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \mathtt{ret} : \tau) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(R,S)_u)$
$+ p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 6.4.1), both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(P,Q)_w)$$

$$= \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma)) \mid (P, \Upsilon^{t,t'}_{t}(Q))_w)$$

$$\Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \mathtt{ret} : \tau) \mid \overset{A}{\lhd}{}^{t'}_{x,y,z}(R,S)_u)$$

$$= \Phi((V, t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3), \mathtt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \mathtt{ret} : \tau) \mid (R, \Upsilon^{t,t'}_{t}(S))_u)$$

□

Theorem 9.4.2 (quantum physicist's method soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid P \vdash e : \tau \mid Q$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return well-formed)*
- $\Phi(V : \Gamma \mid P) \geq p$ *(initial bounds peak)*
- $\Phi(V : \Gamma \mid P) + q \geq \Phi((V, \mathtt{ret} \mapsto v) : (\Gamma, \mathtt{ret} : \tau) \mid Q) + p$ *(diff. bounds net)*

Proof. The soundness proof follows from Lemma 9.4.1 as follows:

The well-formedness $v : \tau$ condition already is given as the first consequent of Lemma 9.4.1.

For the peak cost bound, let w be the worldview in $\mathtt{wv}(P)$ guaranteed by the second consequent of Lemma 9.4.1 so that the inequality $\mathtt{wv}(P).\ \Phi(V : \Gamma \mid P_w) \geq p$ holds. Then consider the following inequalities:

$$\begin{aligned}
\Phi(V : \Gamma \mid P) &= \max_{u \in \texttt{wv}(P)} \Phi(V : \Gamma \mid P_u) && \textit{def} \\
&\geq \Phi(V : \Gamma \mid P_w) && w \in \texttt{wv}(P) \\
&\geq p && \textit{Lemma } 9.4.1
\end{aligned}$$

For the net cost bound, let u be the worldview in $\texttt{wv}(Q)$ maximizing the quantity of energy $\Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_u)$. Then let w be the worldview in $\texttt{wv}(P)$ that the third consequent of Lemma 9.4.1 assures the existence of so that the inequality $\Phi(V : \Gamma \mid P_w) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_u) + p$ holds. Finally, consider the following inequalities:

$$\begin{aligned}
\Phi(V : \Gamma \mid P) + q &= \max_{w' \in \texttt{wv}(P)} \Phi(V : \Gamma \mid P_{w'}) + q && \textit{def} \\
&\geq \Phi(V : \Gamma \mid P_w) + q && w \in \texttt{wv}(P) \\
&\geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_u) + p && \textit{Lemma } 9.4.1 \\
&= \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q) + p && \textit{def}
\end{aligned}$$

□

9.5 Tree Height

Now that this chapter has established the basics of the quantum physicist's method, its worldviews can be leveraged to give resource functions based on tree height. Many programs, like binary tree search, naturally have costs that depend on the height of trees. By combining such height-based resource functions with resource tunneling, AARA can be made to automatically infer tight peak cost bounds for the call-stack usage of tree traversals.

The trick to supporting resource functions based on tree height is based on maxima. Tree height is defined in terms of maxima: the height of a leaf is 0 and the height of a node is the maximum height of its subtrees plus 1. This height definition interacts nicely with the maximum operation used to define potential energy in Figure 9.12. In particular, worldviews are useful for reasoning about maxima: if worldviews witness distinct typings, then certainly whichever of those typing assigns the maximum energy valid, even if one does not know which such typing that is. These observations all lead to the following key idea for representing tree height: If the left subtree can be given some height-based annotation in one worldview, and the right subtree can be given the same annotation in another worldview (holding all other annotations constant), then the full tree can be given that height-based annotation (unshifted) as well.

For the purposes of this section, the resource functions $R_k(n)$ given by the recurrence matrix A are assumed to be nondecreasing. Such monotonicity is a typical and desirable property of resource functions, so monotonicity is not a significant restriction. The binomial coefficients and offset Stirling numbers both meet this condition (and thus so do their products). This condition guarantees the inequality $a \cdot R_k(m) + b \cdot R_k(n) \leq (a + b) \cdot R_k(\max(m, n))$ for $a, b \geq 0$, which is important for some of the reasoning of this section.

$$Ind(T(\tau)) = \{\texttt{d}'_n, \texttt{h}_n \mid 1 \leq n \leq D_{max}\} \cup \texttt{e}'.Ind(\tau)$$

Figure 9.13: New annotation index definition using tree height

$$\textsc{H-Leaf}\quad \frac{}{\Gamma \mid P \vdash \texttt{leaf} : T(\tau) \mid\ P, Q}$$

$$\textsc{H-Node}\quad \frac{Q \in \overset{A}{\trianglelefteq}{}^{\texttt{ret}}_{x',y',z'}(P) \qquad \forall w, j.\, Q_w(x'.\texttt{h}_j) = 0 \qquad R \in \overset{A}{\trianglelefteq}{}^{\texttt{ret}}_{x',y',z'}(P) \qquad \forall w, j.\, R_w(z'.\texttt{h}_j) = 0}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \Upsilon^{x,x'}_{x}(\Upsilon^{y,y'}_{y}(\Upsilon^{z,z'}_{z}(Q, R))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid\ P}$$

$$\textsc{H-CaseT}\quad \frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ U \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U_w(x.\texttt{h}_j) = 0 \qquad U' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U'_w(z.\texttt{h}_j) = 0 \\ T' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(P, Q) \qquad \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid T' \vdash e_2 : \tau \mid\ U, U'\end{array}}{\Gamma, t : T(\sigma) \mid P, \Upsilon^{t,t'}_{t}(Q) \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ R, \Upsilon^{t,t'}_{t}(S)}$$

Figure 9.14: New tree typing rules using tree height

9.5.1 Annotation Indices

First, a new annotation index must be created for resource functions based on maxima. This new index is $\texttt{h}_k$ and can be assigned formally by replacing the tree case of Figure 3.5 with the definition of Figure 9.13. resource function. The annotation at $\texttt{h}_k$ corresponds to the scalar of the resource function $R_k(n)$ in the potential energy of a tree, where the parameter n is chosen to be that tree's height.

9.5.2 Typing Rules

To support tree height, the typing judgment needs a slightly changed meaning from Section 9.4.1. This change is to partially revoke the ability for annotations to be negative. In particular, there is a new side condition that the height-based annotations are nonnegative. The reason for this restriction is that the maxima that are used to define height-based potential energy in Section 9.5.3 do not allow negative numbers to distribute over them (essentially, $-1 \cdot \max(5, 0) \neq \max(-5, 0)$), but they do allow nonnegative numbers to do so. Thus, the techniques described here work with nonnegative height-based annotations, but not negative ones.

The new typing rules in support of tree height can be found in Figure 9.14. Otherwise, all typing rules are the same as in Figures 9.6 to 9.8. These new typing rules make use of Definition 9.5.1, which is an extension to the shifting operator of Definition 6.3.2.

Definition 9.5.1 (height shifting)**.** *The height-shifting operator $\overset{A}{\trianglelefteq}$ extends Definition 6.3.2's shifting operator $\overset{A}{\triangleleft}$ to include height-based annotation indices in its domain and to lift its codomain from a single annotation map to a set of annotation maps.*
Let $\mathtt{c}$ *act as 0 and* $t.\mathtt{h}_i$ *act as* i *for vector indexing, and fix* $\vec{b}$ *such that* $\vec{b}_i = a(i)$ *for such indices.*

$$\overset{A}{\trianglelefteq}{}^{t}_{x,y,z}(a) = \left\{ \lambda i. \begin{cases} \vec{c}_i & i = x.\mathtt{h}_j \\ \vec{d}_i & i = z.\mathtt{h}_j \\ \overset{A}{\triangleleft}{}^{t}_{x,y,z}(a)_{\mathtt{c}} + (A \cdot \vec{b})_{\mathtt{c}} & i = \mathtt{c} \\ \overset{A}{\triangleleft}{}^{t}_{x,y,z}(a)_i & \mathit{otherwise} \end{cases} \middle| \; \curlyvee^{x,z}_{t}(\vec{c}, \vec{d}) = A \cdot \vec{b} \wedge \vec{c} \geq \vec{0} \wedge \vec{d} \geq \vec{0} \right\}$$

The rule *H-Leaf* is syntactically the same as *Q-Leaf*, but I include it again to emphasize that it now must account for the new height-based annotation indices when introducing the unconstrained annotation map Q.

The rule *H-Node* for the first time introduces a many-to-one correspondence between initial and remainder worldviews, which is key for reasoning correctly about tree height. This correspondence is represented notationally by letting Q, R represent two versions of $\overset{A}{\trianglelefteq}{}^{\mathtt{ret}}_{x',y',z'}(P)$ with distinct worldviews. For each worldview $w \in \mathtt{wv}(P)$, there is a worldview $w_0 \in Q$ and worldview $w_1 \in R$ representing the two extremes for how height-based energy might be allocated. In w_0, all such energy is put on the second subtree (because the first is constrainted to have 0), while in w_1, all energy is put on the first (because the second is constrainted to have 0). Even though one cannot in general statically determine which subtree is higher, one of these two worldviews witnesses the assignment of all height-based energy to the highest subtree, which will then correctly justify the remainder annotation P. This typing rule's reasoning is essentially the same as using $a \geq b$ and $a \geq c$ to conclude $a \geq \max(b, c)$ for unknowns b and c.

The rule *H-CaseT* pattern matches trees with height-based energy by splitting the height-based energy annotations across the subtrees. While there are many ways to split this annotation up, the choice maximizing energy occurs when all energy is assigned to the highest subtree; all other choices can only (safely) lose energy. Then for uncomputing the remainder context, the conditions match those of *H-Node*.

Example 9.5.1. To see how these new rules enable height-based cost analysis, recall the binary tree search function `mem` from Figure 9.3. The time it takes to perform a binary tree search is at worst proportional to the height of the tree.

This cost model can be simulated in AARA by adding tick expressions as in Figure 9.15. To match the desired cost bound, it should be expected that the function `mem` can be given the type $T^1(\mathbb{Z}) \to \mathbb{B} \sim T^0(\mathbb{Z})$, where here the tree's superscript represents the height-based, linear resource function's annotation.

Indeed, `mem` can be given the desired type. This typing is witnessed by the energy comments included in Figure 9.15. Notationally, I write $\mathtt{t1} : \mathtt{p/q}, \mathtt{t2} : \mathtt{r/s}, \mathtt{c} : \mathtt{t/u}$ to indicate that, in the first worldview, the trees `t1` and `t2` have p and r units of linear, height-based energy, respectively, alongside t units of free energy. Simultaneously, in the second worldview, the trees `t1` and `t2` have q and s units of linear, height-based energy, respectively, alongside u units of free

```
1      fun mem (x,tr) = case tr of
2        | Leaf -> false
3        | Node (t1,y,t2) ->     (* t1:1/0, t2:0/1, c:1/1 *)
4          let _ = tick{1} in    (* t1:1/0, t2:0/1, c:0/0 *)
5          if y = x
6          then true
7          else if y < x
8          then mem (x,t1)       (* t1:0/-1, t2:0/1,  c:0/0 *)
9          else mem (x,t2)       (* t1:1/0,  t2:-1/0, c:0/0 *)
```

Figure 9.15: Code for binary tree search with energy comments

```
1     fun size tr = case tr of
2       | Leaf -> 0
3       | Node (t1,_,t2) ->         (* t1:1/0, t2:0/1, c:1/1 *)
4         let _ = tick{1} in        (* t1:1/0, t2:0/1, c:0/0 *)
5         let t1size = size t1 in   (* t1:1/0, t2:0/1, c:0/0 *)
6         let _ = tick{-1} in       (* t1:1/0, t2:0/1, c:1/1 *)
7         let _ = tick{1} in        (* t1:1/0, t2:0/1, c:0/0 *)
8         let t2size = size t2 in   (* t1:1/0, t2:0/1, c:0/0 *)
9         let _ = tick{-1} in       (* t1:1/0, t2:0/1, c:1/1 *)
10        t1size + t2size + 1
```

Figure 9.16: Code for binary tree size with energy comments

energy. Remainder annotations are elided because there is no remainder energy.

The new height-based reasoning comes into play in line 3. There, two different worldviews are used to capture two different shifting options from the set given by $\overset{A}{\trianglelefteq}$ (where A gives the polynomial recurrence matrix). In the first worldview, all the energy is put onto the left subtree. In the second worldview, all the energy is put onto the right subtree. Because each worldview individually only considers the height-based energy assigned to one such subtree, the amount of energy that each worldview sees is no more than that of the highest subtree (plus one for the free energy). This maximum among subtrees yields height-based energy.

Note that lines 8 and 9 correspond to differing branches of the search, and their comments disagree as to which worldview witnesses quantum validity. In line 8, the first worldview's annotations are nonnegative, whereas in line 9 the second worldview's annotations are. This disagreement is fine due to the ability to collapse worldviews. When typing this branch, collapse rules would be used to remove the second worldview in line 8 and the first worldview in line 9. After doing so, both branches are left with matching annotations, exactly as needed for typing branches.

Example 9.5.2. These new rules also enable reasoning about height in tree traversals. Consider the tree size function `size` given in Figure 9.4. If one were interested in bounding the naive call stack usage of `size`, one would find that this amount is bounded by the height of the input tree.

$$\Phi(\texttt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \vec{a})$$

$$= \underline{\delta}(A, \vec{a}) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}'.i}) + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} \left(\Phi(v_1 : T(\tau) \mid \vec{b}) + \Phi(v_3 : T(\tau) \mid \vec{c})\right)$$

Figure 9.17: New potential energy definition using tree height

To find this bound with AARA, the code can be put into let-normal form and instrumented with ticks as in Figure 9.16. Then the function `size` can be given the type $T^1(\mathbb{Z}) \to \mathbb{Z} \sim T^1(\mathbb{Z})$ where the superscript represents the annotation for the height-based, linear resource function. This type expresses a net cost of zero (because all stack is returned), and a peak cost bound linear in the height of the input tree (matching the desired stack bound).

To see how `size` can be given that type, look to the comments in the code. These comments give the annotations of the initial type context using the same notation as in Example 9.5.1. Then note that, at the end of the function body, the conditions are met to uncompute the subtrees `t1` and `t2` back into `tr` with a type of $T^1(\mathbb{Z})$. Specifically, there are two worldviews that match annotations everywhere aside from the two subtrees, each subtree has zero energy in one of the worldviews, and each worldview assigns annotations to the subtrees (and free energy) according to $\overset{A}{\trianglelefteq}$. Here, those annotations from $\overset{A}{\trianglelefteq}$ are just where all energy is assigned to one of the two subtrees (aside from one unit of freed energy).

9.5.3 Potential Energy

Figure 9.17 formally represents the energy of trees with height-based annotations. This definition makes use of Definitions 9.5.2 and 9.5.3 and contains a maximum operation. This maximum is over all the ways annotations can be assigned to the subtrees when pattern matched, and it turns out this value gets maximized when all potential energy is assigned to the highest subtree. This relation between a tree's potential energy and height height is formalized via Lemma 9.5.1.

Definition 9.5.2 (height potential shifting)**.** *The the height shifting operator* $\overset{A}{\underline{\blacktriangleleft}}$ *extends Definition 6.3.2's shifting operator* $\overset{A}{\blacktriangleleft}$ *to include height-based annotation indices in its domain and to lift its codomain from an annotation map to a set of pairs of annotation maps. These pairs are complementary maps where the annotations for the left and right subtrees are switched.*
For trees annotated by a, let $b(t.i) = a(i)$. Then formally:

$$\overset{A}{\underline{\blacktriangleleft}}(a) = \{\langle \lambda i.\, c(x.i), \lambda i.\, c(z.i)\rangle \mid c \in \overset{A}{\trianglelefteq}{}^{t}_{x,y,z}(b)\}$$

Definition 9.5.3 (height constant-difference operator)**.** *The height constant-difference operator $\underline{\delta}$ extends Definition 6.4.1's constant-difference operator δ to handle the behaviour of trees' height-based annotation indices.*
Let $\mathtt{c}$ act as 0 and both $\mathtt{d}'_i$ and $\mathtt{h}_i$ act as i for vector indexing. Let $\vec{b}$ collect those entries of $\vec{a}$ with indices of the form $\mathtt{c}$ and $\mathtt{d}'_i$, and let $\vec{c}$ collect those entries of $\vec{a}$ with indices of the form $\mathtt{c}$ and $\mathtt{h}_i$.

$$\underline{\delta}(A, \vec{a}) = (A \cdot \vec{b})_{\mathtt{c}} + (A \cdot \vec{c})_{\mathtt{c}} - \vec{a}_{\mathtt{c}}$$

By convention, if $\mathtt{c}$ does not index $\vec{a}$, then it is treated as if $\vec{a}_{\mathtt{c}} = 0$.

Lemma 9.5.1 (tree-height potential energy)**.** *Let the entries of $\vec{a}$ be 0 except possibly at indices of the form $\mathtt{h}_i$, where the entries may be nonnegative. The potential of a tree v with annotation $\vec{a}$ is a function of the v's height h.*

$$\Phi(v : T(\tau) \mid \vec{a}) = \sum_{i=1}^{D_{max}} \vec{a}_{\mathtt{h}_i} \cdot R_i(h)$$

Proof. The proof of this statement proceeds by structural induction over the tree v. To make notation easier, this proof uses the convention that annotation indices like $\mathtt{h}_i$ act as i for vector indexing.

$v = \mathtt{leaf}$ In this case, the list v's height h is 0 and the following equalities hold:

$$\begin{aligned}
\Phi(\mathtt{leaf} : T(\tau) \mid \vec{a}) &= 0 && \textit{def} \\
&= \sum_{i=1}^{D_{max}} \vec{a}_{\mathtt{h}_i} \cdot R_i(h) && R_i(0) = 0
\end{aligned}$$

$v = \mathtt{node}(v_1,\, v_2,\, v_3)$ In this case, let the height of the subtree v_1 be h_1 and the height of the subtree v_2 be h_2. The list v's height h is $\max(h_1, h_2) + 1$, and the following equalities hold:

$$\begin{aligned}
&\Phi(\mathtt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \vec{a}) \\
&= \underline{\delta}(A, \vec{a}) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{\mathtt{e}'.i}) + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} (\Phi(v_1 : T(\tau) \mid \vec{b}) + \Phi(v_3 : T(\tau) \mid \vec{c})) && \textit{def} \\
&= \underline{\delta}(A, \vec{a}) + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} (\Phi(v_1 : T(\tau) \mid \vec{b}) + \Phi(v_3 : T(\tau) \mid \vec{c})) && \vec{a}_{\mathtt{e}'.i} = 0 \\
&= (A \cdot \vec{a})_{\mathtt{c}} + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} (\Phi(v_1 : T(\tau) \mid \vec{b}) + \Phi(v_3 : T(\tau) \mid \vec{c})) && \textit{def} \\
&= (A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} (\Phi(v_1 : T(\tau) \mid \vec{b}) + \Phi(v_3 : T(\tau) \mid \vec{c})) && R_0(n) = 1 \\
&= (A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \max_{\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} (\sum_{i=1}^{D_{max}} \vec{b}_{\mathtt{h}_i} \cdot R_i(h_1) + \sum_{i=1}^{D_{max}} \vec{c}_{\mathtt{h}_i} \cdot R_i(h_2)) && \textit{IH}
\end{aligned}$$

Consider any pair of annotation maps $\langle \vec{b}, \vec{c}\rangle$ in $\overset{A}{\underline{\blacktriangleleft}}(a)$. By chasing definitions, one finds the following invariant relating $\vec{b}, \vec{c}$, and $\vec{a}$.

$$\begin{aligned}
&\langle \vec{b}, \vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(a) && \textit{def} \\
\Longrightarrow\ & \langle \vec{b}, \vec{c}\rangle = \langle \lambda i.\, d(x.i), \lambda i.\, d(z.i)\rangle \quad \wedge \quad d \in \overset{A}{\trianglelefteq}{}^{t}_{x,y,z}(\vec{e}) \quad \wedge \quad \vec{e}_{t.i} = a(i) && \textit{def} \\
\Longrightarrow\ & \Upsilon_t^{x,z}(d) = A \cdot \vec{e} \quad \wedge \quad \forall i.\, d_i \geq 0 && \textit{def} \\
\Longrightarrow\ & \forall i.\, d(x.\mathtt{h}_i) + d(y.\mathtt{h}_i) = (A \cdot \vec{e})_{t.\mathtt{h}_i} \quad \wedge \quad d(x.\mathtt{h}_i) \geq 0 \quad \wedge \quad d(y.\mathtt{h}_i) \geq 0 && \textit{def} \\
\Longrightarrow\ & \forall i.\, \vec{b}_{\mathtt{h}_i} + \vec{c}_{\mathtt{h}_i} = (A \cdot \vec{e})_{t.\mathtt{h}_i} \quad \wedge \quad \vec{b}_{\mathtt{h}_i} \geq 0 \quad \wedge \quad \vec{c}_{\mathtt{h}_i} \geq 0 && \textit{def} \\
\Longrightarrow\ & \forall i.\, \vec{b}_{\mathtt{h}_i} + \vec{c}_{\mathtt{h}_i} = (A \cdot \vec{a})_{\mathtt{h}_i} \quad \wedge \quad \vec{b}_{\mathtt{h}_i} \geq 0 \quad \wedge \quad \vec{c}_{\mathtt{h}_i} \geq 0 && \textit{def}
\end{aligned}$$

With this invariant in mind, the following inequalities hold:

$$\begin{aligned}
&\vec{b}_{\mathtt{h}_i} \cdot R_i(h_1) + \vec{c}_{\mathtt{h}_i} \cdot R_i(h_2) \\
&\leq (\vec{b}_{\mathtt{h}_i} + \vec{c}_{\mathtt{h}_i}) \cdot R_i(\max(h_1, h_2)) && \vec{b}_{\mathtt{h}_i}, \vec{c}_{\mathtt{h}_i} \geq 0,\ R_i(n)\ \textit{nondecreasing} \\
&= (A \cdot \vec{a})_{\mathtt{h}_i} \cdot R_i(\max(h_1, h_2)) && \vec{b}_{\mathtt{h}_i} + \vec{c}_{\mathtt{h}_i} = (A \cdot \vec{a})_{\mathtt{h}_i} \\
&= (A \cdot \vec{a})_{\mathtt{h}_i} \cdot R_i(\max(h_1, h_2)) + 0 \cdot R_i(\min(h_1, h_2)) && \textit{algebra}
\end{aligned}$$

Since either of $\vec{b}, \vec{c}$ could have been chosen to be $A \cdot \vec{a}$ (leaving the other $\vec{0}$), the maximum value of the above expression over any such $\vec{b}, \vec{c}$ occurs at $(A \cdot \vec{a})_i$ scaling $R_i(\max(h_1, h_2))$ and and 0 scaling $R_i(\min(h_1, h_2))$. Knowing this fact allows this case to be finished with the following equalities:

$$\begin{aligned}
&(A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \max_{\langle \vec{b},\vec{c}\rangle \in \overset{A}{\underline{\blacktriangleleft}}(\vec{a})} \left(\sum_{i=1}^{D_{max}} \vec{b}_{\mathtt{h}_i} \cdot R_i(h_1) + \sum_{i=1}^{D_{max}} \vec{c}_{\mathtt{h}_i} \cdot R_i(h_2)\right) && \mathit{IH} \\
&= (A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \sum_{i=1}^{D_{max}} (A \cdot \vec{a})_{\mathtt{h}_i} \cdot R_i(\max(h_1,h_2)) + \sum_{i=1}^{D_{max}} 0 \cdot R_i(\min(h_1,h_2)) && \mathit{max} \\
&= (A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \sum_{i=1}^{D_{max}} (A \cdot \vec{a})_{\mathtt{h}_i} \cdot R_i(\max(h_1,h_2)) && \mathit{algebra} \\
&= (A \cdot \vec{a})_{\mathtt{c}} \cdot R_0(h-1) + \sum_{i=1}^{D_{max}} (A \cdot \vec{a})_{\mathtt{h}_i} \cdot R_i(h-1) && \mathit{def} \\
&= (A \cdot \vec{a}) \cdot \vec{R}(h-1) && \mathit{algebra} \\
&= \vec{a} \cdot \vec{R}(h) && \mathit{def} \\
&= \sum_{i=0}^{D_{max}} \vec{a}_i \cdot R_i(h) && \mathit{algebra} \\
&= \sum_{i=1}^{D_{max}} \vec{a}_{\mathtt{h}_i} \cdot R_i(h) && \vec{a}_{\mathtt{c}} = 0
\end{aligned}$$

$\square$

Of course, the new shifting operators $\overset{A}{\trianglelefteq}, \overset{A}{\underline{\blacktriangleleft}}$, do not need to allocate energy annotations according to the maximum. Such flexibility is key to automatic reasoning about height-based potential energy because it is not generally computable which subtree is higher. Thus, shifting may no longer perfectly preserve potential energy for trees as in Lemma 6.4.1. However, an analogous inequality can be obtained as Lemma 9.5.2.

Lemma 9.5.2 (height shifting at most conserves energy)**.** *For all $\vec{b} \in \overset{A}{\trianglelefteq}{}^{t}_{x,y,z}(\vec{a})$:*

$$\Phi((t \mapsto \mathtt{node}(v_1,\ v_2,\ v_3)) : (x : T(\tau)) \mid \vec{a})$$

$$\geq \Phi((x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (x : T(\tau), y : \tau, z : T(\tau)) \mid \vec{b})$$

and this inequality is an equality for some $\vec{b}$ where either $\forall j.\, \vec{b}_{z.\mathtt{h}_j} = 0$ or $\forall j.\, \vec{b}_{x.\mathtt{h}_j} = 0$

Proof. To prove this statement, I first prove the inequality. Let $\lambda i.\, \vec{a}_{t.i} = \vec{a'} + \vec{a''}$ where $\vec{a'}$ is zero on those entries indexed by an annotation index of the form $\mathtt{h}_i$ and $\vec{a''}$ is zero on the rest.

$$\begin{aligned}
&\Phi((x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (x : T(\tau), y : \tau, z : T(\tau)) \mid \vec{b}) \\
&= \vec{b}_{\mathsf{c}} + \Phi(v_2 : \tau \mid \lambda i.\, \vec{b}_{y.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, \vec{b}_{x.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, \vec{b}_{z.i}) && \textit{def} \\
&= \overset{A}{\lhd}{}^{t}_{x,y,z}(\vec{a})_{\mathsf{c}} + (A \cdot \vec{a''})_{\mathsf{c}} + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{t.\mathsf{e}'.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, \vec{b}_{x.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, \vec{b}_{z.i}) && \textit{def} \\
&= (A \cdot \vec{a'})_{\mathsf{c}} + (A \cdot \vec{a''})_{\mathsf{c}} + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{t.\mathsf{e}'.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, \vec{b}_{x.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, \vec{b}_{z.i}) && \textit{def} \\
&= \vec{a}_{\mathsf{c}} + \underline{\delta}(A, \vec{a}) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{t.\mathsf{e}'.i}) + \Phi(v_1 : T(\tau) \mid \lambda i.\, \vec{b}_{x.i}) + \Phi(v_3 : T(\tau) \mid \lambda i.\, \vec{b}_{z.i}) && \textit{def} \\
&\leq \vec{a}_{\mathsf{c}} + \underline{\delta}(A, \vec{a}) + \Phi(v_2 : \tau \mid \lambda i.\, \vec{a}_{t.\mathsf{e}'.i}) + \max_{\langle \vec{c}, \vec{d} \rangle \in \overset{A}{\underline{\blacktriangleleft}} (\lambda i.\, \vec{a}_{t.i})} (\Phi(v_1 : T(\tau) \mid \vec{c}) + \Phi(v_3 : T(\tau) \mid \vec{d})) && \langle \lambda i.\, \vec{b}_{x.i}, \lambda i.\, \vec{b}_{z.i} \rangle \in \overset{A}{\underline{\blacktriangleleft}} (\lambda i.\, \vec{a}_{t.i}) \\
&= \vec{a}_{\mathsf{c}} + \Phi(\texttt{node}(v_1,\, v_2,\, v_3) : T(\tau) \mid \lambda i.\, \vec{a}_{t.i}) && \textit{def} \\
&= \Phi((t \mapsto \texttt{node}(v_1,\, v_2,\, v_3)) : (t : T(\tau)) \mid \vec{a}) && \textit{def}
\end{aligned}$$

Now it only remains to consider the $\vec{b}$ rendering this inequality an equality. There is only one step introducing an inequality in the above reasoning: where the maximum over annotation pairs $\langle \vec{c}, \vec{d} \rangle \in \overset{A}{\underline{\blacktriangleleft}} (\lambda i.\, \vec{a}_{t.i})$ is introduced. Let $\langle \vec{c'}, \vec{d'} \rangle$ be such a maximizer. By the definitions of $\overset{A}{\unlhd}, \overset{A}{\underline{\blacktriangleleft}}$, the vectors $\vec{c'}$ and $\vec{d'}$ are invariant on annotation indices that are not of the form $\mathtt{h}_j$. Thus, only indices of the form $\mathtt{h}_j$ matter for the maximizer. As found in the proof of Lemma 9.5.1, this maximization occurs when such indices of one of $\vec{c'}, \vec{d'}$ coincide with $A \cdot \vec{a''}$ and the indices of the other are 0. Indeed, if one is 0, the other must be $A \cdot \vec{a''}$ by the definitions of $\overset{A}{\unlhd}, \overset{A}{\underline{\blacktriangleleft}}$. Therefore, (at least) one of the two choices for $\vec{b}$ tightens the above inequality into an equality, completing the proof.

□

9.5.4 Soundness

To extend the soundness proof of Theorem 9.4.2, it is only necessary to extend the resource-tunneling-soundness of Lemma 9.4.1. This extension is provided via Lemma 9.5.3.

Lemma 9.5.3 (height-based resource tunneling soundness)**.** *If*

- $V \vdash e \Downarrow v \mid (p, q)$ *(an expression evaluates with some cost behavior)*
- $V : \Gamma$ *(the environment of the evaluation is well-formed)*
- $\Gamma \mid P \vdash e : \tau \mid Q$ *(AARA types the expression in that environment)*

then

- $v : \tau$ *(return value is well-formed)*
- $\exists w \in \texttt{wv}(P).\, \Phi(V : \Gamma \mid P_w) \geq p$ *(initial energy bounds peak cost in some worldview)*
- $\forall u \in \texttt{wv}(Q).\, \exists w \in \texttt{wv}(P).\, \Phi(V : \Gamma \mid P_w) + q \geq \Phi((V, \texttt{ret} \mapsto v) : (\Gamma, \texttt{ret} : \tau) \mid Q_u) + p$ *(every rem. worldview has some init. worldview s.t. difference in energy bounds net cost)*

Proof. The soundness proof proceeds by lexicographic induction over the derivation of the evaluation judgment followed by the typing judgment.

Only the cases for the new height-based tree rules need to be considered, as all other cases for of the proof are the same as in Lemma 9.4.1.

E-Leaf This case does not actually change to consider height-based annotations, but I include it anyway for completeness.

Suppose the last rule applied for the evaluation judgment is *E-Leaf*.

$$\textsc{E-Leaf}\quad \frac{}{V \vdash \texttt{leaf} \Downarrow \texttt{leaf} \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{H-Leaf}\quad \frac{}{\Gamma \mid P \vdash \texttt{leaf} : T(\tau) \mid\ P, Q}$$

Because $\texttt{leaf} : T(\tau)$ follows from *V-Leaf*, the needed well-formedness judgment holds. Then because potential energy is always nonnegative in P's classically valid worldview, the peak cost bound is satisfied in that worldview. And finally, because the initial and remainder annotations are identical except for the leaf annotations $\vec{b}$ and leaves carry no energy regardless of annotation (Figure 3.9), the net cost bound is also satisfied with the following equality regardless of worldview $w \in \texttt{wv}(P) = \texttt{wv}(P, Q)$:

$$\Phi(V : \Gamma \mid P_w) = \Phi((V, \texttt{ret} \mapsto \texttt{leaf}) : (\Gamma, \texttt{ret} : T(\tau)) \mid (P, Q)_w)$$

E-Node Suppose the last rule applied for the evaluation judgment is *E-Node*.

$$\textsc{E-Node}\quad \frac{}{V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash \texttt{node}(x,\ y,\ z) \Downarrow \texttt{node}(v_1,\ v_2,\ v_3) \mid (0,0)}$$

Then $p = q = 0$ and only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{H-Node}\quad \frac{Q \in \overset{A}{\trianglelefteq}{}^{\texttt{ret}}_{x',y',z'}(P) \qquad \forall w, j.\, Q_w(x'.\texttt{h}_j) = 0 \qquad R \in \overset{A}{\trianglelefteq}{}^{\texttt{ret}}_{x',y',z'}(P) \qquad \forall w, j.\, R_w(z'.\texttt{h}_j) = 0}{\Gamma, x : T(\tau), y : \tau, z : T(\tau) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(Q, R))) \vdash \texttt{node}(x,\ y,\ z) : T(\tau) \mid\ P}$$

Because $\texttt{node}(v_1,\ v_2,\ v_3) : T(\tau)$ follows from *V-Node* and the assumed well-formedness judgment $(V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau))$, the needed well-formedness judgment holds. Then because the potential energy is always nonnegative in $\curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\triangleleft}{}^{\texttt{ret}}_{x',y',z'}(Q, R))))$'s classically valid worldview, the peak cost bound is satisfied in that worldview. Finally, note that, for any worldview $w \in \texttt{wv}(P)$, shifting P_w perfectly conserves potential energy in some worldview u in $\texttt{wv}(Q, R)$ by *Lemma* 9.5.2 because some premisses ensure that Q and R exhaust the ways of assigning zero to the height-based annotations. Then because sharing conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 9.5.2) for one of Q, R, the net cost bound is also satisfied with the following equality:

$$\Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau)) \mid \curlyvee_x^{x,x'}(\curlyvee_y^{y,y'}(\curlyvee_z^{z,z'}(\overset{A}{\lhd}{}^{\texttt{ret}}_{x',y',z'}(Q, R))))_u)$$

$$= \Phi((V, x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, x : T(\tau), y : \tau, z : T(\tau), \texttt{ret} : T(\tau)) \mid P_w)$$

E-CaseT-Leaf This case does not actually change to consider height-based annotations, but I inlcude it anyway for completeness

Suppose the last rule applied for the evaluation judgment is *E-CaseT-Leaf*.

E-CaseT-Leaf
$$\frac{V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{leaf} \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

H-CaseT
$$\frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ U \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U_w(x.\texttt{h}_j) = 0 \qquad U' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U'_w(z.\texttt{h}_j) = 0 \\ T' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(P, Q) \qquad \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid T' \vdash e_2 : \tau \mid\ U, U'\end{array}}{\Gamma, t : T(\sigma) \mid P, \curlyvee_t^{t,t'}(Q) \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ R, \curlyvee_t^{t,t'}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$ holds by assumption, each of the following judgments have now been found:

- $V, t \mapsto \texttt{leaf} \vdash e_1 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma))$
- $\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S'$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$
(2) $\exists w \in \texttt{wv}(P, Q').\, \Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) \geq p$
(3) $\forall u \in \texttt{wv}(R, S').\, \exists w \in \texttt{wv}(P, Q').$
$\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) + q$
$\geq \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid (R, S')_u) + p$

The well-formedness judgment (1) $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven. Finally, because leaves carry no potential energy regardless of annotation (Figure 3.9) both these cost bounds follow from (2) and (3) using the following equalities:

$$\Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, Q')_w) = \Phi((V, t \mapsto \texttt{leaf}) : (\Gamma, t : T(\sigma)) \mid (P, \curlyvee_t^{t,t'}(Q))_w)$$

$$\Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid (R, S')_u)$$
$$= \Phi((V, t \mapsto \texttt{leaf}, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid (R, \curlyvee_t^{t,t'}(S))_u)$$

E-CaseT-Node Suppose the last rule applied for the evaluation judgment is *E-CaseT-Node*.

$$\textsc{E-CaseT-Node}\quad \frac{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)}{V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 \Downarrow v \mid (p, q)}$$

Then only one typing rule remains that could be used to conclude the typing derivation:

$$\textsc{H-CaseT}\quad \frac{\begin{array}{c}\Gamma, t : T(\sigma) \mid P, Q' \vdash e_1 : \tau \mid\ R, S' \\ U \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U_w(x.\mathtt{h}_j) = 0 \qquad U' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(R, S) \qquad \forall w, j.\, U'_w(z.\mathtt{h}_j) = 0 \\ T' \in \overset{A}{\trianglelefteq}{}^{t'}_{x,y,z}(P, Q) \qquad \Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid T' \vdash e_2 : \tau \mid\ U, U'\end{array}}{\Gamma, t : T(\sigma) \mid P, \curlyvee^{t,t'}_{t}(Q) \vdash \texttt{case}\ t\ \texttt{of leaf} \to e_1 \mid \texttt{node}(x,\ y,\ z) \to e_2 : \tau \mid\ R, \curlyvee^{t,t'}_{t}(S)}$$

Both of these rules' premisses hold by inversion.

Because $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma))$ by assumption, the rule *V-Context* can be inverted to learn $v_1 :: v_2 v_3 : T(\sigma)$. Then further, the rule *V-Node* can be inverted to learn all of $v_1 : T(\sigma)$, $v_2 : \sigma$, and $v_3 : T(\sigma)$. Using *V-Context*, one can then use these well-formedness judgments to derive

$$(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$$

Each of the following judgments has now been found:

- $V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3 \vdash e_2 \Downarrow v \mid (p, q)$
- $(V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma))$
- $\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma) \mid T' \vdash e_2 : \tau \mid\ U, U'$

With these judgments, the inductive hypothesis can be applied to learn:

(1) $v : \tau$

(2) $\exists w \in \mathtt{wv}(T')$.
$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid T'_w) \geq p$

(3) $\forall u \in \mathtt{wv}(U, U').\, \exists w \in \mathtt{wv}(T')$.
$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid T'_w) + q$
$\geq \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid (U, U')_u)$
$+ p$

The well-formedness judgment $v : \tau$ is what this case needs, so only this case's cost bounds remain to be proven.

Because sharing perfectly conserves potential energy (Lemma 3.4.1) and shifting does not gain energy (Lemma 9.5.2). the peak cost bound follows from (2) with the following inequality, which holds regardless of worldview $w \in \mathtt{wv}(T') = \mathtt{wv}((P, \curlyvee^{t,t'}_{t}(Q)))$:

$$\Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma)) \mid T'_w)$$

$$\leq \Phi((V, t \mapsto \texttt{node}(v_1,\ v_2,\ v_3)) : (\Gamma, t : T(\sigma)) \mid (P, \curlyvee^{t,t'}_{t}(Q))_w)$$

Finally, note that, for any worldview $u \in \texttt{wv}(R, S)$, shifting $(R, S)_u$ perfectly conserves potential energy in some worldview u' in $\texttt{wv}(U, U')$ by *Lemma* 9.5.2 because some premisses ensure that U and U' exhause the ways of assigning zero to the height based annotations. Then because sharing conserves potential energy (Lemma 3.4.1) and shifting conserves the potential energy of a tree (Lemma 9.5.2) for one of U, U', the net cost bound follows from (3) using the previous inequality and following equality:

$$\Phi((V, t \mapsto \texttt{node}(v_1,\, v_2,\, v_3), x \mapsto v_1, y \mapsto v_2, z \mapsto v_3, \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), x : T(\sigma), y : \sigma, z : T(\sigma), \texttt{ret} : \tau) \mid (U, U')_{u'})$$

$$= \Phi((V, t \mapsto \texttt{node}(v_1,\, v_2,\, v_3), \texttt{ret} \mapsto v) : (\Gamma, t : T(\sigma), \texttt{ret} : \tau) \mid (R, \curlyvee_t^{t,t'}(S))_u)$$

□

9.5.5 Demotion

There are some additional opportunities for optimizing the dynamics of height-based annotations when using the mixed polynomial and exponential energy introduced in Section 6.6. Much like Section 6.6, these optimizations take the form of *demotion*, wherein some annotations can be converted to others. This extra flexibility allows the typing rules to more optimally allocate potential energy, yielding tighter cost bounds.

I use this section to point out the key relations needed for demoting height-based energy. Note, however, that the development of demotion rules based on these relations is not meaningfully different from the development in Section 6.6. Thus, rather than belabour the point, I only explain the relations here, and I forgo making formal rules and proving them sound.

To begin, I recall the demotion of Section 6.6. In Section 6.6, demotion was possible due to the relation between the offset Stirling number $\left\{ {n+1 \atop 2} \right\}$ and the sum of binomial coefficients $\binom{n}{k}$. Specifically,

$$\left\{ {n+1 \atop 2} \right\} = 2^n - 1 = \sum_{k=1}^{\infty} \binom{n}{k} \geq \sum_{k=1}^{D_{max}} \binom{n}{k}$$

For height-based annotations, very similar relations hold. Here, the relevant height-based relations derive from trees. In particular, let h be the height of a (binary) tree and let n be the number of nodes of the tree. Then both of the following hold:

$$\binom{n}{1} = n \geq h = \binom{h}{1}$$

$$\left\{ {h+1 \atop 2} \right\} = 2^h - 1 \geq n = \binom{n}{1}$$

Thus, one unit of node-based linear energy ($\texttt{d}'_{1,1}$) can be safely converted to one unit of height-based linear energy ($\texttt{h}_{1,1}$). Likewise, one unit of height-based base-2 energy ($\texttt{d}'_{0,2}$) can be safely converted to one unit of node-based linear energy ($\texttt{h}_{1,1}$).

9.6 Automation

While the core of the quantum physicist's method type system induces many of the same linear contraints as the systems of previous chapters, the system does pose a few new problems for automation. This section goes over those difficulties and provides an analysis the time complexity of type inference, focusing on the full system with the features of Section 9.5.

The first obstacle to discuss is the inference of the shifting operator $\overset{A}{\trianglelefteq}$. However, it turns out there is no relation in this operator that is not linear. For all $\vec{b}$ in $\overset{A}{\trianglelefteq}$, much of their entries are given by the linear relations of previous chapters. The only new feature to inspect is the behaviour of such vectors' entries with height-based indices like $\mathtt{h}_i$. As laid out in Definition 9.5.1, these indices lie in a sharing relation with each other and are constained to be nonnegative. The former of these is already known to be expressible with linear constraints (specifically of the form $a+b = c$), and the latter already is a linear constraint. Thus, the entire space of annotations in $\overset{A}{\trianglelefteq}$ is expressible with linear constraints, and a linear program should be able to pick out whichever solution is best.

The next obstacle to discuss is the use of worldviews. For the most part, type inference *in* a given worldview is no different than in previous chapters. The question, then, is how properly manage collections of worldviews. In particular, the real difficulty comes from the existential quantification in the definition of being quantumly valid: *some* worldview's annotations must all be nonnegative. It is not completely trivial to decide which worldview this should be. Nonetheless, it turns out this obstacle can be handled rather simply: when in doubt, introduce a new worldview and constrain it to be nonnegative. I discuss this solution in more detail in the following paragraphs.

To manage collections of worldviews for type inference, it first suffices to realize that the use of the structural rules concerning superposition and collapse can be structured in a particular way. In particular, if ever an additional worldview might be needed for the type derivation of a particular function, then that worldview can be introduced from the start using *Q-SuperposeL*. Then all worldviews can be kept around as long as possible until each extra worldview is eliminated from the remainder context using *Q-CollapseR*—no structural worldview rules need to be used anywhere else. Thus the question reduces to determining how many worldviews might need to be introduced at the start.

The number of worldviews needed is influenced by many different factors, but it is not too difficult to get an upper bound on the maximum number that could possibly be of use. The following paragraphs describe such a bound.

Firstly, every subexpression might need one worldview witnessing the quantum validity of the initial context, and another worldview witnessing the quantum validity of the remainder context. (Function applications can also make use of the initial context's classically valid worldview for resource-tunneling purposes.) These worldviews can be constrained to be nonnegative, which is sufficient to satisfy the requirements of quantum validity. The problem of figuring out which worldview is classically valid is thusly handled by creating a worldview specifically for that purpose, even if another worldview might already suffice. This approach introduces many unneeded worldviews but is nonetheless sufficient to allow type inference.

Finally, the number of needed worldviews might double for typing expressions using *H-Node* and *H-CaseT*, as each of these typing rules introduce a 2:1 relation between different contexts' worldviews. For *H-Node*, each worldview assigning some height-based energy to the tree being typed requires two worldviews in the initial context that assign all of that (shifted) energy to one of the two subtrees. *H-CaseT* requires a similar relation to uncompute the tree being pattern-matched. As a result of this doubling, the number of worldviews that might be needed for typing a function is at worst in $O(2^n)$ where n is the size of the function body.

With this all in mind, the full type-inference algorithm can be laid out as follows:

1. basic type inference
2. set up worldviews
3. collect and solve linear contraints

After the first step (which I discuss later), the second step is to set up worldviews as described above. This setup can be accomplished by passing over the code, calculating the worst-case number of worldviews that could be needed, and creating a type-derivation skeleton with these numbers. The bottleneck here is just the number of worldviews that need slots in the type derivation, of which there may be exponentially many in the size of the function body.

Then the third step then can run in time that is polynomial in the size of the derivation skeleton. All this step does is set up the linear constraints and solve. Thus, the worst-case time complexity of the second and third steps is exponential in the size of a function body.

This exponential-time complexity does not actually change the true theoretical bottleneck of AARA type inference. That bottleneck is still using Hindley-Milner type inference [72, 106] to complete the first step. Hindley-Milner type inference is DEXPTIME-complete [72, 106], which already contains the complexity of the remaining inference steps.

In practice, this worst-case time complexity does not arise. Hindley-Milner type inference is well-known to be efficient in practice. In addition, the number of worldviews needed is not usually the worst-case number because that requires code both with no helper functions and where every operation is a tree manipulation. The practical efficiency of type inference is supported by the experiments of Section 9.7.

9.7 Experiments

To test the efficacy of the quantum physicist's method type system, a prototype analyzer was implemented, and its performance was compared to the Resource Aware ML (RaML) implementation of AARA (version 1.4.2). Experiments were then run to determine both how accurately and how fast the two can analyze naive stack bounds for the OCaml standard library Set module [91]. The experiments support that the quantum physicist's method greatly increases the accuracy of the AARA analysis with only moderate performance tradeoff. All experiments were implemented in OCaml 4.06.0, run on a Mac with a 2.3 GHz Dual-Core Intel Core i5 processor, and use the Coin-Or linear program solver version 1.16 [34].

Experimental Setup The experiments of this section were designed to answer the following questions in a real-world code environment:

Q1. How much does the quantum physicist's method improve the bounds AARA can find?

Q2. Is using the quantum physicist's method practically efficient?

To set up the experiments, a prototype implementation of the quantum physicist's method type inference was implemented. This prototype uses the method of automation described in Section 9.6, wherein exponentially-many worldviews might be used in the size of a function body; it does not attempt to reduce the number of worldviews needed. The prototype supports univariate polynomial resource functions, and it can parameterize its resource functions on tree depth using the system of Section 9.5.2.

To evaluate the efficacy of the prototype's bound inference, the state-of-the-art AARA implementation Resource Aware ML (RaML) was used as a control for comparison. RaML is a mature and optimized implementation of AARA [81, 82]. It supports many features the prototype does not, including multivariate resource functions and cost-freedom. However, this version (1.4.2) of RaML does not use remainder contexts.

The Set module from OCaml's standard library [91] was used as test code. This module implements sets using binary trees. For parsing reasons, some of OCaml's syntactic sugar was manually desugared.

To compare both implementations' performances, each was used to infer a naive stack bound for the Set module functions. The naive stack-cost metric counts the number of call stack frames needed without accounting for tail-calls or other optimization. That is, one stack frame is consumed prior to each function call, and one stack frame is returned upon each function return.

Measuring naive stack bounds in the Set module should test multiple pertinent features of this chapter's type system. Because stack frames are a resource which is returned after use, potential barriers naturally arise for resource tunneling to deal with. Further, because the Set module is implemented using trees, tree depth a relevant parameter.

The recorded data from each of the implementations concern the time taken, the number of linear constraints generated, and the resulting cost bound. When gathering the experimental data, each implementation was run at the lowest degree D_{max} that could successfully analyze the code. In the event of failure, the data from the linear bound inference was used. As a result, each implementation usually only searched for linear-cost bounds. When only linear resource functions are used, RaML does not make use of cost-free typing or multivariate resource functions—these features therefore do not impact RaML's performance for most of the experimental results. Nonetheless, the prototype implementation needs quadratic bounds for filter, and RaML needs them for 4 functions (compare_aux through subset).

When analyzing a function, each typechecker must also analyze all helper functions, which sometimes exceeds one hundred of lines of code. To exclude performance data unrelated to the AARA analysis, each implementation was only timed after Hindley-Milner unification assigned base types to the code. As a result, the time data only includes the time each implementation takes to generate and solve its linear programs. To make this comparison fair, both implementations used the same linear program solver (COIN-OR [34]).

Findings The answers to the experimental questions can be summarized as follows:

		RaML				Prototype			
Function	LoC	Time(s)	Constrs	Stack Bound	Returned	Time(s)	Constrs	Stack Bound	Returned
ordcompare	3	0.01	3	0	0	0.00	23	0	0
height	5	0.01	8	0	0	0.00	214	0	0
create	4	0.02	31	0	0	0.04	10722	0	0
bal	44	0.17	505	1	1	3.67	513451	1	1
add	62	1.11	1085	$n + 1$	1	7.33	697545	$d + 1$	$d + 1$
singleton	1	0.00	2	0	0	0.00	548	0	0
add_min_elt	52	0.45	532	$n + 1$	1	3.69	538691	$d + 1$	$d + 1$
add_max_elt	52	0.47	534	$n + 1$	1	3.85	538763	$d + 1$	$d + 1$
join	71	4.29	2197	$n_0 + n_1 + 2$	1	6.43	764565	$d_0 + d_1 + 2$	$d_0 + d_1 + 2$
min_elt	9	0.01	26	n	1	0.01	3616	d	d
min_elt_opt	9	0.01	32	$n + 1$	1	0.01	4224	d	d
max_elt	9	0.01	26	n	1	0.01	3616	d	d
max_elt_opt	9	0.01	32	$n + 1$	1	0.01	4224	d	d
remove_min_elt	54	0.46	540	n	1	3.68	539700	d	d
merge	75	0.81	1124	$2n_1 + 1$	1	6.36	601655	$d_1 + 1$	$d_1 + 1$
concat	102	5.96	2816	$n_0 + 2n_1 + 1$	1	10.57	903682	$n_0 + n_1 + .5d_1 + 4.5$	$n'_0 + 4$
split	91	17.46	4447	fail	fail	11.72	941207	$n_0 + 4$	$n'_0 + n'_1 + 4$
is_empty	1	0.01	8	0	0	0.01	160	0	0
mem	10	0.02	34	n	0	0.03	12566	d	d
remove	96	5.03	2211	$2n + 1$	1	8.56	880667	$d + 1$	$d + 1$
union	127	164.91	15568	fail	fail	25.14	1586889	fail	fail
inter	137	94.03	9532	fail	fail	24.33	1486736	$n_0 + n_1 + 5$	$n' + 5$
diff	137	95.01	9535	fail	fail	21.40	1433676	$n_0 + n_1 + 5$	$n' + 5$
cons_enum	7	0.01	27	$n_0 + 1$	1	0.02	6338	d_0	d_0
compare_aux	27	0.13	509	$n_0 s_0 + n_1 s_1 + s_1 + 1$	1	0.21	61545	$.5s_0 + 1.5n_0 + .5s_1 + 1.5n_1 + 1$	1
compare	30	0.17	739	$n_0 + 2n_1 + 2$	1	0.29	82131	$1.5n_0 + 1.5n_1 + 2$	2
equal	33	0.19	745	$n_0 + 2n_1 + 3$	0	0.30	87239	$1.5n_0 + 1.5n_1 + 3$	3
subset	20	0.42	1607	$n_0 n_1$	0	0.61	156721	$d_0 + d_1$	$d_0 + d_1$
iter	4	0.01	28	$n + 1$	1	0.04	11161	d	d
fold	4	0.02	28	$n + 1$	1	0.04	15726	d	d
for_all	3	0.02	32	n	0	0.04	11581	d	d
exists	3	0.02	32	n	0	0.04	11581	d	d
filter	115	30.88	5074	fail	fail	61.35	2170433	$.5n^2 + .5n + 5$	$n' + 5$
partition	114	74.91	10076	fail	fail	14.33	1158301	$2n + 5$	$n'_0 + n'_1 + 5$
cardinal	3	0.01	27	n	0	0.01	3763	d	d
elements_aux	4	0.01	32	n_1	1	0.02	6953	d	d
elements	7	0.01	36	$n + 2$	1	0.03	8182	$d + 1$	$d + 1$

Table 9.2: Experimental statistics and inferred stack bounds

A1. The quantum physicist's method allowed AARA to find a tighter stack bound than RaML in 30 out of 37 analyzed functions, which is a significant improvement. In all other cases, the implementations find matching bounds.

A2. While the prototype implementation does have more performance overhead than RaML, both implementations usually take similar orders of magnitude of time to run.

The results of the analysis are in table 9.2. For each of 36 functions from the Set module (and the Ord module comparison ordcompare), the table contains the following data: the number of lines of code in the test file (LoC), the time each implementation takes to infer type annotations (Time), the number of linear constraints generated during inference (Constraints), the inferred upper bound on call stack frames (Stack Bound), and the number of stack frames returned according to the analyses (Returned). In resource bounds, n is used to describe the node count of tree arguments, d the depth of tree arguments, and s the size of list arguments. Subscripts disambiguate arguments by index, and a prime (') is added to refer to function returns instead of arguments.

The data shows that the quantum physicist's method analysis yields significantly tighter results. In 30 cases, the prototype implementation finds a tighter bound than RaML, and in the remaining 7 cases they find the tight bound. In one of these 7 cases (union), neither implementation finds a bound, and in the remaining 6 each finds the same tight bound. The prototype implementation provides a tight bound on returned resources in 31 out of 37 cases, whereas RaML never infers that more than 1 stack frame is returned (because RaML cannot make use of remainder contexts). The ground truth is that each function should always return all its stack.

Interestingly, the prototype finds good bounds even though the Set module performs many tree operations based on *semantic* properties of the code, rather than structural. For instance, in bal, trees are balanced by tracking their tree height. The type system completely ignores that those integers track tree height, and is nonetheless able to infer good cost bounds.

The general trend of Table 9.2 is that the prototype can perform more accurate resource analyses at the expense of slower performance. However, on difficult-to-analyze code like the join function, the speed of the prototype's analysis approaches RaML's order of magnitude. This similar time data shows that the prototype can already achieve plausible performance. Furthermore, the prototype can analyze difficult code like the split function, where RaML fails to derive a bound. The prototype implementation is only unable to handle union, while RaML fails on 6 different functions.

The table also shows that the time taken by the prototype to generate linear constraints is roughly equal to the time taken to solve them. This performance profile suggests that time efficiency could be improved by either a stronger LP solver or more aggressive heuristics and strategies for constraint generation. In particular, it may be possible to improve on the naive generation of worldviews.

That the prototype performs as quickly as it does compared to RaML is somewhat surprising. The prototype internally maintains a completely decorated type derivation tree, while RaML does not. To keep up performance, RaML aggressively reuses annotations where possible, in general aiming to generate as few linear constraints as possible. However, as Table 9.2 shows, the multiple-orders-of-magnitude more constraints generated in this fashion did not result in commensurate slowdown. To explore the reasons behind this lack of slowdown, the performance statistics for some runs of the prototype were broken down further and recorded in table 9.3.

The performance statistics in table 9.3 break down the prototype implementation's performance data in the following way: Timing is subdivided into constraining and running the LP solver, and the constraint count breaks down into variable identities ($x = y$ for variables x, y), constant offsets ($x = y + k$ for variables x, y and constant k), other equality constraints, and any remaining inequalities.

It would seem that slowdown was avoided because easy constraints comprise a majority of the implementation's constraints, and modern LP solvers can eliminate such easy constraints quickly. For example, constraints of the form $x = y$ comprise 55.56% of generated constraints.

9.8 Proof-Theoretical Observations

I now take some time to briefly explore some of the deeper proof-theoretical implications of this type system, particularly of its worldviews. In this section, I explain their connection to the additive product and to intersection types. These connections are potentially obscured by the way that worldviews have otherwise been presented, but this section should make the connections clear.

Additive Products Interestingly, despite admitting a form of contraction and weakening, these worldviews are very much linear. In particular, they behave like a connective from linear logic:

Function	Constrain Time	LP Time	Var IDs	Offsets	Eqs	Other Ineqs
ordcompare	0.00	0.00	7	0	3	13
height	0.00	0.00	83	0	26	105
create	0.00	0.04	5138	0	1205	4379
bal	1.83	1.84	278669	262	49868	184652
add	3.32	4.01	391214	335	60411	245585
singleton	0.00	0.00	208	0	43	297
add_min_element	1.94	1.75	292062	279	50773	195577
add_max_element	1.96	1.89	292124	279	50773	195587
join	3.96	2.47	417754	357	64253	282201
min_elt	0.00	0.01	1738	9	392	1477
min_elt_opt	0.00	0.01	2117	9	403	1695
max_elt	0.00	0.01	1738	9	392	1477
max_elt_opt	0.00	0.01	2117	9	403	1695
remove_min_elt	2 .00	1.68	292644	284	51707	195065
merge	2.39	3.97	325913	332	56022	219388
concat	5.24	5.33	492858	427	73863	336534
split	5.81	5.91	523681	414	72934	344178
is_empty	0.00	0.01	55	0	26	79
mem	0.00	0.03	7388	22	578	4578
remove	5.09	3.46	496624	414	71456	312173
union	16.24	8.90	894505	625	111227	580532
inter	14.28	10.05	840854	603	100644	544635

Table 9.3: Prototype statistic breakdown

the additive product ("with" or &) [63]. To see this, consider the rules for the additive product in Figure 9.5, and note the correspondence between linear logical sequents and quantum physicist's method typing judgments provided via Theorem 9.8.1. The key to the this correspondence is that worldviews act like additive products over entire typing contexts, and logical contexts can be put into a matching normal form wherein additive products are the outermost connective.

Theorem 9.8.1 (worldviews are additive multiplicands)**.** *Additive products and typings with worldviews have essentially the same behaviour. That is, linear logic sequents concerning additive products (over entire contexts) and AARA typing judgments concerning worldviews each can simulate the other.*

Proof. To show this simulation, I show it for each of their characterizing rules. The simulation is given through the correspondence where where the linear logical proposition $P\&Q$ and linear logical sequent $P \vdash Q$ each correspond to the collection of (individually quantumly valid) worldviews P, Q and typing judgment $\Gamma \mid P \vdash e : \tau \mid\ Q$, respectively.

First, I show that the quantum physicist's method typing rules provide corresponding implications between typing judgments that simulate each of the rules for the additive product. For this purpose, it suffices to consider only the rules of Figure 9.5 with singleton or empty logical contexts because of the equivalence $P, Q\&R \vdash S \iff (P \otimes Q)\&(P \otimes R) \vdash S$ which lets the additive product absorb other context members. This form gives the natural correspondence to typing judgments, which put worldviews on the outermost level.

&L1 and &L2 Given $\Gamma \mid P \vdash e : \tau \mid\ S$, derive $\Gamma \mid P, Q \vdash e : \tau \mid\ S$ and $\Gamma \mid Q, P \vdash e : \tau \mid\ S$.

$$\begin{aligned} &\Gamma \mid P \vdash e : \tau \mid\ R \\ \Longrightarrow\ &\Gamma \mid P, Q \vdash e : \tau \mid\ R \quad \wedge \quad \Gamma \mid Q, P \vdash e : \tau \mid\ R \qquad\qquad Q-CollapseL \end{aligned}$$

Note that this case uses the notation P, Q for two annotation maps P and Q with distinct worldviews.

&R Given $\Gamma \mid P \vdash e : \tau \mid\ Q$ and $\Gamma \mid P \vdash e : \tau \mid\ R$, derive $\Gamma \mid P \vdash e : \tau \mid\ Q, R$.

$$\begin{aligned} &\Gamma \mid P \vdash e : \tau \mid\ Q \quad \wedge \quad \Gamma \mid P \vdash e : \tau \mid\ R \\ \Longrightarrow\ &\Gamma \mid P, P \vdash e : \tau \mid\ Q, R \qquad\qquad \textit{simultaneous typing} \\ \Longrightarrow\ &\Gamma \mid P \vdash e : \tau \mid\ Q, R \qquad\qquad Q-SuperposeL \end{aligned}$$

Note that this case makes use of the fact that two typing judgments can be derived simultaneously in one judgment across separate worldviews. Also note that this case uses the notation P, P to stand for two maps with distinct worldviews that are otherwise copies of P.

Now I show that linear logic provides corresponding derivations simulating each of the superposition/collapse rules. For this purpose, I consider the alternative forms of the superposition/collapse rules which act over arbitrary numbers of worldviews at once, rather than one at a time. The desired derivations are the following:

Q-SuperposeL Given $P\&Q\&Q \vdash R$, derive $P\&Q \vdash R$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\mathit{axiom}}{P\&Q \vdash P}\,\&L1 \qquad \dfrac{\dfrac{\dfrac{}{Q \vdash Q}\,\mathit{axiom} \quad \dfrac{}{Q \vdash Q}\,\mathit{axiom}}{Q \vdash Q\&Q}\,\&R}{P\&Q \vdash Q\&Q}\,\&L2}{P\&Q \vdash P\&(Q\&Q)}\,\&R \qquad P\&(Q\&Q) \vdash R}{P\&Q \vdash R}\,\mathit{cut}$$

Q-SuperposeR Given $P \vdash Q\&R$, derive $P \vdash Q\&R\&R$ as follows:

$$\dfrac{\dfrac{P \vdash Q\&R \quad \dfrac{\dfrac{}{Q \vdash Q}\,\mathit{axiom}}{Q\&R \vdash Q}\,\&L1}{P \vdash Q}\,\mathit{cut} \qquad \dfrac{\dfrac{P \vdash Q\&R \quad \dfrac{\dfrac{}{R \vdash R}\,\mathit{axiom}}{Q\&R \vdash R}\,\&L2}{P \vdash R}\,\mathit{cut} \quad \dfrac{P \vdash Q\&R \quad \dfrac{\dfrac{}{R \vdash R}\,\mathit{axiom}}{Q\&R \vdash R}\,\&L2}{P \vdash R}\,\mathit{cut}}{P \vdash R\&R}\,\&R}{P \vdash Q\&(R\&R)}\,\&R$$

Q-CollapseL Given $P \vdash R$, derive $P\&Q \vdash R$ as follows:

$$\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\mathit{axiom}}{P\&Q \vdash P}\,\&L1 \quad P \vdash R}{P\&Q \vdash R}\,\mathit{cut}$$

Q-CollapseR Given $P \vdash Q\&R$, derive $P \vdash Q$ as follows:

$$\dfrac{P \vdash Q\&R \qquad \dfrac{\dfrac{}{Q \vdash Q}\,\mathit{axiom}}{Q\&R \vdash Q}\,\&L1}{P \vdash Q}\,\mathit{cut}$$

$\square$

Intersection Types Collections of worldviews can also be given another characterization as intersection types [14, 35, 36]. Whenever $\Gamma \mid P \vdash e : \tau \mid (w \mapsto \vec{a}, u \mapsto \vec{b})$, it is as if the expression e (and the remainder) could be typed with either $\vec{a}$ or $\vec{b}$ as annotations. The purpose of intersection types is to describe such double-typings. Further, intersection types can be given introduction and elimination rules as in Figure 9.18, and these rules are essentially the same as the introduction and elimination rules for the additive product.

9.9 Related Work

In this section I detail some related work. While I know of no other work using comparable cost-analysis ideas, some preexisting work does exhibit comparable results. I go over some of those results here.

$$\cap\text{I}\quad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \tau \cap \sigma} \qquad \cap\text{E1}\quad \frac{\Gamma \vdash e : \tau \cap \sigma}{\Gamma \vdash e : \tau} \qquad \cap\text{E2}\quad \frac{\Gamma \vdash e : \tau \cap \sigma}{\Gamma \vdash e : \sigma}$$

Figure 9.18: Typing rules for intersection types

Tree Height in AARA The ability to find stack bounds based on tree height in AARA is not new. Campbell has already created an extension to AARA for this purpose [25, 26]. Rather than building off of quantum physical principles, his work builds off of bunched logic [116, 123], which is a close relative of linear logic.

Bunched typing admits two kinds of context formers (essentially products), one of which Campbell uses to represent adding the potential energy of its multiplicands, and one of which Campbell uses to represent taking the maximum potential energy of its multiplicands. These context formers are much like the additive and multiplicative products represented in the quantum physicist's method system.

Nonetheless, bunched contexts are rather different than the worldviews of the quantum physicist's method. Bunched contexts are structured as trees, and working with this extra structure requires additional machinery. Campbell introduces a set of axiomatic equivalences which manipulate the context structure. To infer bounds in Campbell's system, a user initially supplies a skeleton indicating the kinds of resource functions to consider, and then a nontrivial procedure must handle how the context should be manipulated.

The resource tunneling of the quantum physicist's method is also more general than the reasoning of Campbell's system. Because Campbell's system only reasons about stack and stack is fully reusable, every net cost is 0. As a result, Campbell's system never grapples with dynamics of net costs. One of the effects of avoiding net costs is that Campbell's system never needs negative annotations, as such annotations only arise when net costs are subtracted. By focusing only on stack, Campbell's system can therefore sidestep some of the main problems that resource tunneling solves for resources with nonzero net costs.

Maxima in AARA

Aside from this chapter's work and Campbell's work, there is one other AARA system that uses maxima in its reasoning: Hoffmann and Shao's system for parallel programs [79]. The span (parallel cost) of a parallel program depends upon the maximum cost of a given line of execution. Unlike any of the work discussed so far in this section, their work uses *cost-free types* (see Chapter 8) to get at maxima. Their approach treats two branches of parallel execution as if one branch accrues no cost and as if the branches are executed sequentially. This treatment allows the system to obtain two sets of annotations, one each where cost only comes from one of the two branches. The pointwise upper bound of these two annotations then provides an upper bound on the maximum cost, just as it does for the typing of branches in non-parallel AARA.

Tree Height Outside of Functional Programming

Many automatic cost analyses for non-functional progams do not focus as much on data structures, and instead provide bounds based on the integers in a program. However, it is still possible for such analyses to provide bounds in terms of tree height. For example, SPEED allows users to define quantitative functions that can capture tree height [69]. However, SPEED does not ensure that such user-defined functions actually describe tree height.

Reusable Resources

One of the key uses of the quantum physicist's method is to reallocate resource that can be reused. However, the analysis of costs in terms of resources that can be reused is largely ignored by many cost analysis systems in favor of analyzing time costs specifically. Reusable resources are generally harder to reason about because their peak costs do not align with their net costs. Those that can reason about reusable resources often reason specifically about space (e.g., [137]), where the net cost is *a priori* known to be zero.

Nonetheless, at least one other line of work designed for the automatic inference of general peak costs: that of Albert et al. [6]. This work considers abstract program executions to identify costs at each point in the program, and then it reduces this bound by the amount of resources it identifies that could be reused. This process leaves a bound on the maximum number of resources in use at one time, i.e., the peak cost.

In principle, other techniques should be able to reason about peak costs as well. In particular, by augmenting program counters with some counter to track their maximum value, numerical techniques could be employed like the invariant-based techniques described in Chapter 4.

Hyperproperties

If one views the evolution of each worldview as a separate program trace, then the quantum physicist's method might be viewed in terms of hyperproperties [33]. A *hyperproperty* holds for *sets* of program traces, while in contrast a *trace property* holds for singular traces. The definition of amortized cost (Definition 9.3.1) describes a trace property, as amortized cost validity can be considered with respect to individual traces. However, the conditions of quantum cost from Section 9.3 describe a hyperproperty, as quantum cost validity is considered with respect to a set of traces, one for each worldview.

It does not appear that the concerns of cost analysis have previously been expressed in terms of hyperproperties. However, future work might apply hyperproperty techniques to the quantum physicist's method. In particular, such techniques might improve upon the naive inference algorithm presented in Section 9.6.

Chapter 10

Conclusion

To begin the conclusion, I first recall my thesis statement:

Thesis Statement:

1. **AARA's state-of-the-art automatic capabilities can be improved to derive tighter cost bounds more efficiently for more kinds of programs and more kinds of resource costs.**
2. **Such improvements can be made by leveraging key features of *linearity* intrinsic to the AARA type system.**

I believe the work I have provided here has now proven both counts. Each contribution of this thesis was motivated by a problem concerning the first count, and each solution was derived from the second. I now recount how each chapter contributed to these counts in more detail:

- For the remainder contexts of Chapter 5, I addressed how AARA would naively lose potential energy and derive poor cost bounds for reusable resources in otherwise-simple program examples. My solution of remainder contexts then came with deep connections to linear-logic proof-search techniques, and introduced some beautiful symmetries into the AARA typing rules. Furthermore, remainder contexts were also shown to have an interesting connection to physical (un)computation. It is well-established that linear logic can be applied to physical computation (like quantum computing), and I find it quite intersting that here I have found some sort of converse: physical principles being applied to a linear type theory (and I do so again later with the quantum physicist's method).

  Compared to their simplicity, the impact of remainder contexts has been disproprtionately large. Not only do they solve the problem they were designed for, but also they smooth over some rough edges of AARA surrounding sharing and let expressions. As a result, remainder contexts have already begun to worm their way into "standard" AARA techniques, showing up in Grosen et al.'s multivariate system [65] and the implementation of the Nomos typechecker [47].

- For the exponential system of Chapter 6, I addressed the disparity between the kinds of programs one could easily write and the kinds of bounds one could easily analyze. It is easy to write exponential-cost programs—just use two recursive calls—but AARA could not successfully analyze such programs. My solution to ths problem came in the form of

Stirling numbers, which turned out to be exactly the right choice of basis resource functions for exponentials. Not only were their conical combinations maximally expressive, but also they came with a nice linear recurrence that could be handled by the linear program solver that powers AARA. Furthermore, these results were generalized to a variety of resource functions satisfying a linear recurrence, allowing easy integration of other resource functions in the future.

Of course combinatorics of Stirling numbers also played a very interesting role. When mixed with polynomials, combinatorial properties enabled interesting demotion rules necessary for tight cost bounds. Moreover, such combinatorial properties enabled my later multivariate extension.

- In Chapter 7, I addressed AARA's need for multivariate resource functions to handle my new exponential resource functions in, e.g., the accumulator code pattern employed in tail-recursive functions. I solved this by taking a deep combinatorial inspection of the linear recurrence used by Stirling numbers and using this understanding to generalize Stirling numbers over program values. Crucially, conical combinations of resource functions in the resulting system are closed under products, allowing sharing to not only be defined, but inferrable via linear programming. I also found the way that all the complex definitions clicked smoothly together to be quite satisfying.
- In Chapter 8, I addressed a problem of efficiency: cost-free types were expensive to infer but necessary for many of AARA's analyses. Moreover, they could not be inferred at all for my exponential system. To address this problem, *maps* over annotation vectors were inferred, rather than the annotation vectors themselves. This kind of approach was only possible because AARA's annotations are often manipulated linearly at each subexpression of a program.

  While not every cost-free type could be inferred this way, empirical testing showed that this approach could drastically improve the efficiency of the AARA analysis. In my own experience with AARA, I have found that cost-free type inference really is one of the slower parts of the analysis, and I am glad to have found some way to improve it.
- Finally, in Chapter 9, I addressed poor cost bounds arising from poor (re)allocation of resources in AARA. This problem showed most strongly during a tree traversal: how could one soundly and automatically reallocate energy stored on one subtree to a different subtree using only local energy manipulation? Here the solution was to delve deeper into the linear-physical connection and take a solution inspired by quantum physics. By placing worldviews in the role of superpositions, resource tunneling could reallocate resources around potential barriers in a similar way to quantum tunneling. The resulting system could then not only reason about reallocating resources quite flexibly, but also enabled cost bounds to depend on tree depth. Empirical testing also confirmed that this approach is efficient enough to analyze real code.

  While the methods of this chapter were probably the most unusual in this thesis, I do not think the results were a fluke. It is well-established that quantum physics is governed by a linear flavor of logic, just like the AARA type system, and worldviews are indeed just linear additive products. There is something intrinsic about this linear behaviour, and I am proud to have brought it to fruition in this work.

Thusly, I have improved AARA's handling of reusable resources, the kinds of cost bounds AARA can infer, and even improved AARA's efficiency. As a result, AARA can more competently handle analyses involving memory, multiple recursive calls, trees, tail recursion, non-tail recursion, and more.

Similarly, I have shown that the many facets of linearity (logical, algebraic, etc.) form a key route to improving AARA. In my work, I have drawn upon linear-logic proof search, linear recurrences, linear maps, quantum mechanical principles, and more. Indeed, by always sticking to such principles, the resulting type system has been able to be inferred efficiently by a linear-program solver.

I now finish the conclusion with some words about the journey of this thesis. When I began my graduate-school career, I was amazed by AARA's ability to automatically and efficiently derive cost bounds which, as an undergraduate, I had thought must be derived by hand. And now, after that veil has long been lifted, after I have become well-aware of the power of automatic cost analyses, I am still amazed by AARA. Now, however, that amazement is because of the incredible ways in which features have coincided to allow AARA to work. Without linear types, AARA would not be able to analyze resources at all. Without linear programming, that analysis would not be fast. Without linear recurrences, that analysis would not yield interesting bounds. And without the physicality of the physicist's method of amortized-cost analysis, more linearity would not have been able to sneak in to support reusable resources in the way that this thesis has done. Yet somehow all this linearity has come together, and I have been able to make it into my thesis. I hope that readers of this work will find it even half as interesting as I have.

Bibliography

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical computer science*, 111(1-2):3–57, 1993. 3.8

[2] Samson Abramsky and Bob Coecke. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures*, 2:261–325, 2009. 9.2

[3] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16*, pages 157–172. Springer, 2007. 1.1

[4] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures 6*, pages 113–132. Springer, 2008. 1.1, 4.3

[5] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *Journal of automated reasoning*, 46:161–203, 2011. 4.3, 4.3

[6] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative resource analysis. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 85–100. Springer, 2015. 9.9

[7] Elvira Albert, Miquel Bofill, Cristina Borralleras, Enrique Martin-Martin, and Albert Rubio. Resource analysis driven by (conditional) termination proofs. *Theory and Practice of Logic Programming*, 19(5-6):722–739, 2019. 4.3

[8] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings 17*, pages 117–133. Springer, 2010. 4.3

[9] Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. Precise cost analysis via local reasoning. In *Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 319–

333. Springer, 2013. 4.3

[10] Robert Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming*, pages 85–103. Springer, 2010. 4.2, 4.3, 5.7, 8.11

[11] Robert Atkey. Polynomial time and dependent types. *Proceedings of the ACM on Programming Languages*, 8(POPL):2288–2317, 2024. 4.1

[12] Martin Avanzini and Georg Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016. 4.3

[13] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 104–124. Springer, 2010. 4.1

[14] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment1. *The journal of symbolic logic*, 48(4): 931–940, 1983. 9.8

[15] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. A calculus for amortized expected runtimes. *Proceedings of the ACM on Programming Languages*, 7(POPL):1957–1986, 2023. 4.2, 8.11

[16] John S Bell. On the einstein podolsky rosen paradox. *Physics Physique Fizika*, 1(3):195, 1964. 9.2

[17] Charles H Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973. 1.2, 5, 5.2, 5.2.2

[18] Jon Louis Bentley, Dorothea Haken, and James B Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980. 4.3

[19] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2020. 5.2.2

[20] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. Templates and recurrences: better together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 688–702, 2020. 1.1, 4.3

[21] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):1–50, 2016. 1.1, 4.3

[22] Bruno Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin*, 10(3):19–29, 1976. 4.3

[23] David Cachera, Thomas Jensen, Arnaud Jobin, and Florent Kirchner. Inference of polynomial invariants for imperative programs: A farewell to gröbner bases. *Science of Computer Programming*, 93:89–109, 2014. 4.3

[24] Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Open-sourcing facebook infer: Identify bugs before you ship. *code. facebook. com blog post*, 11, 2015. 4.3

[25] Brian Campbell. Type-based amortized stack memory prediction. 2008. 5.7, 9.9

[26] Brian Campbell. Amortised memory analysis using the depth of data structures. In *European Symposium on Programming*, pages 190–204. Springer, 2009. 3.1, 9.9

[27] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 467–478, 2015. 3.1, 4.2, 5.7

[28] Iliano Cervesato, Joshua S Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Extensions of Logic Programming: 5th International Workshop, ELP'96 Leipzig, Germany, March 28–30, 1996 Proceedings 5*, pages 67–81. Springer, 1996. 1.2, 5, 5.2

[29] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. 2003. 3.8

[30] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3):331–365, 2019. 8.11

[31] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(4):1–52, 2019. 4.3, 6.8

[32] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2):261–300, 2001. 4.1

[33] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. 9.9

[34] COIN-OR. COIN-OR clp. `https://projects.coin-or.org/Clp`. Accessed: 2020. 9.7, 9.7

[35] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19:139–156, 1978. 9.8

[36] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Salle'. Functional characterization of some semantic equalities inside λ-calculus. In *Automata, Languages and Programming: Sixth Colloquium, Graz, Austria, July 16–20, 1979 6*, pages 133–146. Springer, 1979. 9.8

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977. 4.3, 8.3, 8.10

[38] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978. 4.3, 8.3, 8.11

[39] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, 2000. 4.1

[40] Joseph W Cutler, Daniel R Licata, and Norman Danner. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. 4.2, 8.11

[41] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8, 2012. 4.1

[42] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. *Logical Methods in Computer Science*, 6, 2010. 4.1

[43] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 115–126, 2012. 4.1

[44] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. *ACM SIGPLAN Notices*, 43(1):133–144, 2008. 4.1

[45] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018. 3.1, 4.1

[46] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 305–314, 2018. 3.1

[47] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021. 3.1, 4.2, 10

[48] Steven De Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings 14*, pages 479–494. Springer, 2016. 4.3, 7.11

[49] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Synthesizing invariants by solving solvable loops. In *International Symposium on Automated Technology for Verification and Analysis*, pages 327–343. Springer, 2017. 4.3, 8.11

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 8.8.2

[51] Mario Dehesa-Azuara, Matthew Fredrikson, Jan Hoffmann, et al. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728. IEEE, 2017. 3.1

[52] Henry DeYoung and Frank Pfenning. Data layout from a type-theoretic perspective. *Electronic Notes in Theoretical Informatics and Computer Science*, 1, 2023. 3

[53] Ross Duncan. Types for quantum computing. 2006. 9.2

[54] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM (JACM)*, 57(6):1–47, 2010. 4.3, 8.10, 8.11

[55] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64. IEEE, 2015. 4.3

[56] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. From shapes to amortized complexity. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings 19*, pages 205–225. Springer, 2018. 4.2, 8.11

[57] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, 1993. 2.1

[58] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, pages 254–273. Springer, 2016. 4.2, 8.11

[59] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Asian Symposium on Programming Languages and Systems*, pages 275–295. Springer, 2014. 4.2, 4.3

[60] Stuart J Freedman and John F Clauser. Experimental test of local hidden-variable theories. *Physical review letters*, 28(14):938, 1972. 9.2

[61] Dan R Ghica and Alex I Smith. Bounded linear types in a resource semiring. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*, pages 331–350. Springer, 2014. 4.1

[62] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58:3–31, 2017. 4.3

[63] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987. 3.8, 5.2.1, 9.2, 9.8

[64] Bernd Grobauer. Cost recurrences for dml programs. *ACM SIGPLAN Notices*, 36(10): 253–264, 2001. 4.1

[65] Jessie Grosen, David M Kahn, and Jan Hoffmann. Automatic amortized resource analysis with regular recursive types. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2023. 2.1, 3.1, 3.4, 6.3, 7, 7.1, 7.2.2, 7.3, 7.8, 7.11, 10

[66] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming*, pages 533–560. Springer, 2018. 1.1, 4.2, 8.11

[67] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. For-

mal proof and analysis of an incremental cycle detection algorithm. In *Interactive Theorem Proving*, number 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. 8.5

[68] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 292–304, 2010. 4.3

[69] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices*, 44(1):127–139, 2009. 1.1, 4.3, 4.3, 9.9

[70] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL `https://www.gurobi.com`. 8.8

[71] Martin AT Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019. 4.1

[72] Roger Hindley et al. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969. 3.7, 9.6

[73] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *International Joint Conference on Automated Reasoning*, pages 364–379. Springer, 2008. 4.3

[74] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 2.4, 5.3.2, 5.7

[75] Joshua S Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and computation*, 110(2):327–365, 1994. 1.2, 5, 5.2

[76] Jan Hoffmann. *Types with potential: polynomial resource bounds via automatic amortized analysis*. PhD thesis, lmu, 2011. 4, 3.4.1, 3.4.2, 3.4.4, 3.5, 7.3

[77] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 287–306. Springer, 2010. 1.2, 3, 3.1, 3.4.3, 3.5.1, 3.5, 3.8, 6.2, 8.1, 8.2, 8.8.1

[78] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Asian Symposium on Programming Languages and Systems*, pages 172–187. Springer, 2010. 2.4, 8

[79] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*, pages 132–157. Springer, 2015. 3.1, 5.7, 8, 9.9

[80] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012. 4, 3.1, 7, 7.1, 7.1, 3, 7.8, 7.11, 8, 8.2,

8.8.1

[81] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, pages 781–786. Springer, 2012. 7.3, 8.8.1, 9.7

[82] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 359–373, 2017. 2.1, 3.1, 9.7

[83] Martin Hofmann. A type system for bounded space and functional in-place update. In *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9*, pages 165–179. Springer, 2000. 1, 4.1, 6.1

[84] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003. 4.1, 6.1

[85] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices*, 38(1):185–197, 2003. 1.1, 3.1, 3.8

[86] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *Programming Languages and Systems: 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings 15*, pages 22–37. Springer, 2006. 3.1

[87] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *International Conference on Rewriting Techniques and Applications*, pages 272–286. Springer, 2014. 8.11

[88] Martin Hofmann and Georg Moser. Multivariate amortised resource analysis for term rewrite systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015. 6.8, 7.11

[89] Martin Hofmann, Lorenz Leutgeb, David Obwaller, Georg Moser, and Florian Zuleger. Type-based analysis of logarithmic amortised complexity. *Mathematical Structures in Computer Science*, 32(6):794–826, 2022. 3.1, 6.7, 6.8, 8

[90] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 410–423, 1996. 4.1

[91] Inria. Ocaml/stdlib/set.ml. `https://github.com/ocaml/ocaml/blob/4.00/stdlib/set.ml`. Accessed: 2020. 9.7, 9.7

[92] Alan Jeffrey. Ltl types frp: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 49–60, 2012. 4.1

[93] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis.

In *16th International Symposium on Formal Methods (FM'09)*, pages 354–369, 2009. 3.1

[94] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th Symposium on Principles of Programming Languages (POPL'10)*, pages 223–236, 2010. 3.1, 3.2.1

[95] David M Kahn and Jan Hoffmann. Exponential automatic amortized resource analysis. In *Foundations of Software Science and Computation Structures: 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings 23*, pages 359–380. Springer International Publishing, 2020. 3.1, 6

[96] David M Kahn and Jan Hoffmann. Automatic amortized resource analysis with the quantum physicist's method. *Proceedings of the ACM on Programming Languages*, 5(ICFP): 1–29, 2021. 3.1, 1, 8.5

[97] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2): 133–151, 1976. 4.3

[98] Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. A hoare logic for energy consumption analysis. In *Foundational and Practical Aspects of Resource Analysis: Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers 3*, pages 93–109. Springer, 2014. 4.3

[99] Assaf J Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Ml typability is dexptime-complete. In *Colloquium on Trees in Algebra and Programming*, pages 206–220. Springer, 1990. 9

[100] Maximiliano Klemen, Miguel Á Carreira-Perpiñán, and Pedro Lopez-Garcia. Solving recurrence relations using machine learning, with application to cost analysis. *arXiv preprint arXiv:2309.07259*, 2023. 4.3

[101] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 227–242. Springer, 2011. 4.3

[102] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961. 5.2.2

[103] G Khaciyan Leonid. A polynomial algorithm for linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979. 3.7

[104] Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Selectively-amortized resource bounding. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings 28*, pages 286–307. Springer, 2021. 4.2, 8.11

[105] Harry G Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401, 1989. 9

[106] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978. 3.7, 9.6

[107] Georg Moser and Manuel Schneckenreither. Automated amortised resource analysis for term rewrite systems. *Science of Computer Programming*, 185:102306, 2020. 4.2, 8.11

[108] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021. 4.1

[109] Stefan K Muller and Jan Hoffmann. Modeling and analyzing evaluation cost of cuda kernels. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021. 3.1

[110] Markus Müller-Olm and Helmut Seidl. Polynomial constants are decidable. In *International Static Analysis Symposium*, pages 4–19. Springer, 2002. 4.3

[111] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):29–es, 2007. 8.11

[112] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *39th Conference on Programming Language Design and Implementation (PLDI'18)*, 2018. 3.1, 6

[113] Tobias Nipkow and Hauke Brinkop. Amortized complexity verified. *Journal of Automated Reasoning*, 62:367–391, 2019. 4.2, 8.11

[114] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, 2022. 1.1, 4.1

[115] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51: 27–56, 2013. 4.3

[116] Peter O'hearn. On bunched typing. *Journal of functional Programming*, 13(4):747–796, 2003. 9.9

[117] Bernhard Ömer. *Quantum programming in QCL*. PhD thesis, 2000. 5.2.2

[118] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019. 4.1

[119] Joël Ouaknine and James Worrell. Decision problems for linear recurrence sequences. In *International Workshop on Reachability Problems*, pages 21–28. Springer, 2012. 6.2.2, 8.5, 8.11

[120] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, pages 1–19. Springer, 2001. 6, 3, 8.11

[121] M Pnueli and Micha Sharir. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981. 8.3

[122] Vaughan R Pratt. Linear logic for generalized quantum mechanics. In *Proceedings Workshop on Physics and Computation, Dallas, IEEE Computer Society*. Citeseer, 1993. 9.2

[123] David J Pym. *The semantics and proof theory of the logic of bunched implications*, vol-

ume 26. Springer Science & Business Media, 2013. 9.9

[124] Vineet Rajani. A type-theory for higher-order amortized analysis. 2020. 4.1

[125] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. A unifying type-theory for higher-order (amortized) cost analysis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021. 4.1, 8.11

[126] Pritom Rajkhowa and Fangzhen Lin. Viap-automated system for verifying integer assignment programs with loops. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 137–144. IEEE, 2017. 4.3

[127] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. Newtonian program analysis via tensor product. *ACM Trans. Program. Lang. Syst.*, 39(2):9:1–9:72, 2017. doi: 10.1145/3024084. URL `https://doi.org/10.1145/3024084`. 8.11

[128] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 266–273, 2004. 4.3, 7.11

[129] Tushar Sharma and Thomas Reps. A new abstraction framework for affine transformers. *Formal Methods in System Design*, 54:110–143, 2019. 8.11

[130] Moritz Sinn and Florian Zuleger. Loopus-a tool for computing loop bounds for c programs. In *WING@ ETAPS/IJCAR*, pages 185–186, 2010. 1.1, 4.3

[131] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *International Conference on Computer Aided Verification*, pages 745–761. Springer, 2014. 4.2, 4.3, 8.11

[132] James Stirling. *The Differential Method: Or, A Treatise Concerning Summation and Interpolation of Infinite Series*. E. Cave, 1749. 6, 6.3

[133] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6, August 1985. 1.1, 3.1, 4.2, 9, 9.2, 9.3.1

[134] The Coq Development Team. The Coq reference manual – release 8.19.0. `https://coq.inria.fr/doc/V8.19.0/refman`, 2024. 4.1

[135] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936. 1.1, 3.7

[136] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM review*, 38 (1):49–95, 1996. 8.10

[137] Pedro B Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008. 4.1, 9.9

[138] Di Wang, David M Kahn, and Jan Hoffmann. Raising expectations: automating expected cost analysis with types. *Proceedings of the ACM on Programming Languages*, 4(ICFP): 1–31, 2020. 3.1

[139] Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):1–26, 2017. 4.1

[140] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975. 1.1, 4.3, 4.3

[141] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*, pages 280–297. Springer, 2011. 4.3, 4.3