Automated Amortized Multi-Potential Analysis

master thesis in computer science

by

Armin Walch

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Univ.-Prof. Dr. Georg Moser
Department of Computer Science

Innsbruck, 14 March 2025

universität innsbruck

Master Thesis

Automated Amortized Multi-Potential Analysis

Armin Walch (11907006)
armin.walch@student.uibk.ac.at

14 March 2025

Supervisor: Univ.-Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

Amortized cost analysis determines runtime cost by averaging costs across a sequence of operations, rather than focusing solely on the worst-case scenario for individual operations. Automated Amortized Resource Analysis (AARA), first introduced by Hofmann et al. is a well-established field of research that automates amortized analysis by integrating type inference with the potential method—a well-known manual technique, based on modeling the potential of data structure with a function. Despite its established broad applicability, AARA was only recently, with the inception of the prototype ATLAS, specialized for the analysis of (probabilistic) self-adjusting data structures. However, the advanced reasoning techniques introduced in these works, have so far only been applied to tree data structures, subject to one specific potential function.

In this thesis, we introduce automated amortized multi-potential analysis, a generalized potential agnostic framework that supports multiple, possibly coexisting potential functions within a single analysis. This approach preserves the strong results of ATLAS, while extending its applicability to a broader range of data structures. A key innovation of our framework is the concept of mixed potentials, which enables the combination of different potential functions to address complex analysis scenarios. Using mixed potentials, we successfully derive the precise amortized cost of sequential insertions into splay trees—an open challenge in the field.

We provide a prototype implementation of this framework, named atlas-2, developed in Haskell. Our experimental results demonstrate that atlas-2 can replicate or improve upon the bounds established by ATLAS for both deterministic and probabilistic benchmarks, with comparable runtime performance. Additionally, we extend the analysis to list-based data structures using a restricted version of multivariate polynomial potential, achieving optimal bounds for a purely functional queue—an improvement over the results obtained by RaML, a prototype introduced by Hoffmann et al.

By making potential functions modular and allowing the extension of the analysis with new data type constructors and constraints, our approach paves the way for rapid prototyping and further research in AARA. This work not only refines existing methodologies but also establishes a flexible foundation for analyzing a wider variety of data structures, thus advancing the state of automated amortized complexity analysis.

Acknowledgments

I would like to express my gratitude to my supervisor, Univ.-Prof. Dr. Georg Moser, for their invaluable guidance, support, and encouragement throughout my research. Their expertise, constructive feedback, and patience were instrumental in shaping this thesis.

My gratitude also goes to my wife Vanessa for her loving support and patience. I also want to thank my son for pulling me away from my work and refueling my joy, especially during frustrating periods.

I also would like to thank my friends and colleagues for their encouraging words.

This work was supported by the FWF project Automated Sublinear Amortised Resource Analysis of Data Structures (AUTOSARD P 36623). I would like to express my appreciation to the members of this project for our insightful discussions and the successful kickoff workshop.

Contents

1 Introduction

Amortized cost analysis determines runtime cost, by averaging over a sequence of operations, rather than only considering the worst case cost of one operation. One instance of such an analysis is the potential method, which utilizes *potential functions* to model the potential stored in a data structure. It was initially proposed to analyze *self-adjusting* data structures, such as splay trees [42, 44]. Here an explicit balancing invariant, guaranteeing a specific worst case cost for every operation, is replaced by a heuristic. With splay trees, for example, the operation *splay* moves the most recently accessed element to the root, while heuristically improving the shape of the tree through a series of rotations. With the help of an amortized analysis it can be shown that such heuristics can make self-balanced data structures as efficient as their balanced counter parts, while being much easier to implement.

Over the last two decades their has been consistent research effort to automate this kind of analysis. Automated amortized resource analysis (AARA) is a term coined by Hoffmann [12] to describe this line of research that traces back to the groundbreaking 2003 work of Hofmann and Jost. In their seminal paper, they introduced a method for automatically predicting heap space usage for first order functional programs through type inference. This early work established three key principles that characterize the analysis and have prevailed throughout subsequent research.

- The static analysis follows a syntax directed approach that is structured around inductively defined type inference rules that are efficiently checkable. The derived proof trees serve as proofs of resource bounds. The inference process follows a two-step approach: it begins with a standard type inference method, such as Hindley–Milner–Damas, and then incorporates resource annotations. These type annotations are determined by solving a numeric optimization problem, typically a linear program.
- The analysis builds on the potential method from amortized analysis [44], allowing it to account for amortization effects across sequences of operations.
- The analysis is formally proven sound based on a well-defined notion of resource consumption. This notion is captured through a cost semantics, typically a big-step semantics, that assigns an evaluation cost to programs.

This initial exploration of AARA enabled the derivation of linear resource bounds for eagerly evaluated first-order programs. The analysis has subsequently been extended in various directions, including non-linear resource bounds [11, 9, 26, 14, 29, 30], higher order functions [24], object-oriented programs [2, 18, 22], lazy-evaluation [40, 41, 45], parallel

evaluation [13], programs with side-effects [1, 4], probabilistic programs [33, 46, 30], concurrent programs with session types [6] and even term rewrite systems [20, 21, 31].

According to Hoffmann [12], the similarity between this method and the manual potential method was not immediately apparent and was first pointed out to them by a colleague around 2004. While AARA employs potential functions, much like its manual counterpart, most works in the field focus primarily on deriving worst-case (upper) or best-case (lower) bounds, without fully leveraging the potential method's capabilities.

Remarkably, the very motivation behind the development of this analysis technique—self-adjusting data structures—was largely overlooked by AARA until the work of Hofmann et al. [19, 29, 30]. Their research marked the first successful automated analysis of splay trees, splay heaps, and pairing heaps. One possible reason for this delay is the complexity introduced by logarithmic resource bounds, a challenge first addressed in prior work by Hofmann et al. [21].

Leutgeb et al. [29] provided the tool ATLAS as a prototype implementation for the analysis of self-adjusting data structures. In a follow-up work they extended their approach into an *expected amortized cost analysis* to additionally handle probabilistic data structures like *randomized splay trees*, *randomized splay heaps* or *randomized meldable heaps*. Their method improved on the previously reported coefficients for the cost bounds found in pen-and-paper proofs. These impressive results depend on a high degree of specialization, including a potential function inspired by said proofs and a novel technique that introduces non-linear reasoning through expert knowledge into the analysis. They even go as far as to suggest that these adaptations of AARA suggest a new approach for the analysis of data structures [29]. Another major difference between ATLAS and previous works is that instead of worst case bounds, they produce upper bounds on the amortized costs.

This approach marks a paradigm shift. Unlike earlier methods, such as multivariate polynomial analysis [9], where the potential function was chosen to be as generic as possible, their approach prioritizes alignment with existing pen-and-paper proofs. The original AARA framework could be adapted to arbitrary user-defined recursive data types [25] whereas multivariate polynomial analysis [9] already introduced constraints by limiting its scope to nested lists and trees. The multivariate analysis was later generalized to arbitrary data types by employing term rewrite systems and tree automate [20]. However, in this method, trees were still treated with the same potential functions as lists, effectively flattening them into a list-like representation. This approach is insufficient for analyzing structures like splay trees, where the potential function must capture the hierarchical structure of the search tree.

For this reason ATLAS further restricts itself to one potential-carrying data type, in order to support a sum-of-logs potential function tailored to trees. However, we argue that this increased specialization is a necessary trade-off to handle more complex problems, such as those posed by self-adjusting data structures. By tailoring the analysis more closely to the structures it aims to capture, ATLAS enables reasoning, that would otherwise be infeasible within a more generic framework.

Despite the specialized potential function used in ATLAS, it has successfully analyzed a variety of data structures, as previously mentioned. However, its limitations are

evident—for instance, skew heaps [43, 27] have thus far resisted automated analysis.

In this work, we introduce *automated amortized multi-potential analysis*, a method that reintroduces a degree of generality to the analysis employed in ATLAS. Unlike previous approaches, we do not achieve this by proposing a universal potential function. Instead, we abstract the existing analysis framework, making it adaptable to different potential functions. This allows us to preserve the strong results of ATLAS while extending its applicability to trees with alternative potential functions or even entirely different data types.

From a practical perspective, automated multi-potential analysis provides a pragmatic approach designed to accelerate future research. By enabling the extension of the core language with additional data type constructors and potential functions—without requiring a complete reimplementation of the analysis framework—our method facilitates rapid prototyping and experimentation. In fact, when introducing a new potential function for a specific data type, it is only necessary to define constraints for constructing and deconstructing that data type, as well as constraints for correctly handling multivariate potential. We believe that this flexibility is essential for addressing open research challenges in the field of self-adjusting data structures.

As part of this thesis, we provide a prototype implementation of the abstract framework, along with instances that replicate and extend previous analyses.

The first instance we implement is the logarithmic potential used in ATLAS, thereby extending the existing analysis. Our experimental results demonstrate that our generalized approach can reproduce—or even improve upon—the bounds previously derived by ATLAS, all while maintaining comparable performance.

Additionally, we implement a restricted version of multivariate polynomial potential, originally introduced by Hoffmann et al. [9], as another instance of our new analysis framework. We apply this instance to derive truly amortized costs for two list-based data structures and improve upon the worst-case bounds produced by RaML [10].

Further, our approach introduces the ability to combine different potential functions within a single analysis, a technique we refer to as *mixed potentials*. We identified the need for this approach when examining a limitation in the previous analysis of splay trees. Although amortized analysis suggests that a sequence of insertions into a splay tree should have an amortized cost of $O(n \cdot \log_2(n))$, this result had not yet been confirmed through automated analysis. More specifically, current techniques are unable to analyze the `fromList` function, which constructs a splay tree by sequentially inserting elements from a given list. During this process, the potential stored in the list is transferred to the resulting tree.

This scenario requires not only the existing logarithmic potential function used for splay trees but also a log-linear potential function to model the potential stored in the input list. Mixed potentials allow us to combine these two functions and analyze their interaction within the program, enabling an end-to-end analysis of the function `fromList`.

Our results demonstrate that automated multi-potential analysis not only refines the existing approaches but also lays the groundwork for extending automated amortized analysis to a wider range of data structures. By making potential functions modular, we anticipate this framework will foster new research avenues in AARA.

More precisely, I make the following contributions.

1. I provide a potential agnostic abstract analysis framework to facilitate automated amortized multi-potential analysis.

2. I extend the logarithmic analysis of ATLAS, as an instance of the abstract framework, reproducing its results for both the deterministic and probabilistic benchmarks and, in some cases, achieving tighter bounds on the amortized costs. Additionally we provide an instance for multivariate polynomial potential [9] for flat lists, that we use to derive amortized costs for an implementation of a *purely functional queue* and a *multi-pop-stack*. Lastly we introduce a novel log-linear potential function for lists.

3. I present a novel technique called mixed potentials for combining potential functions. Using this method we overcome the previous limitation in deriving the cost of a sequence of insertion for a splay tree and arrive at the precise bound of $2n \log_2(n) + {}^1/_2 n$ on the amortized cost of such an operation.

Outline Chapter 2 gives a high level account of our extensions to AARA and informally introduces the basic concepts underlying the manual and automated potential method based on some practical examples.

Chapter 3 provides formal definitions of the central concepts required by our analysis. We also summarize the linearization technique introduced by Hofmann et al. [19] and introduce the language and semantics underlying our analysis.

Chapter 4 presents the automated amortized multi-potential analysis, starting with the abstract analysis framework, representing a generalization of the our predecessor work. Next we provide a soundness proof for this analysis, before we continue to define instances of this framework. Finally we demonstrate an extension to the framework that introduces mixed potentials to the analysis and proof soundness of the extended method.

Chapter 5 focuses on our main result, the analysis of the function `fromList` for splay trees, and gives a detailed account on how mixed potentials are applied to derive the amortized cost of this operation.

Chapter 6 details our prototype implementation and presents experimental results, comparing our new tool against the existing tools ATLAS and RaML.

Chapter 7 provides an overview of previous research in the field of AARA and situates our work within this broader landscape, highlighting its contributions and distinctions from existing approaches.

Chapter 8 concludes this thesis by summarizing our key contributions and outlining potential directions for future research.

2 Overview

In this chapter, we provide a high-level overview of our method, beginning with an introduction to the manual potential method. We then revisit the fundamentals of automated amortized analysis using the potential method, illustrating these concepts with concrete examples: the splay tree and the functional queue. Both of these structures can be fully analyzed in an automated manner by our prototype implementation.

To build an intuitive understanding of the underlying techniques, we manually walk through the type derivations, highlighting key principles. Along the way, we introduce and explain crucial concepts such as cost-free typing, expected amortized cost analysis, and mixed potentials—our novel extension to the method. We demonstrate why mixed potentials are crucial for analyzing the function `fromList` for splay trees.

2.1 The Potential Method

Our analysis is formulated in terms of the potential method, first devised by Tarjan and Sleater [42, 44] to aid in the design of self-adjusting data structures. Similar to other techniques for amortized analysis, it considers the average cost per operation in a sequence, rather then the worst-case cost of a single operation. This enables more fine grained results than a worst-case analysis. The potential method, specifically, is useful when the cost of single operation depends on the current state of the corresponding data structure. The central idea is to assign a non-negative *potential*, denoted ϕ, to the state of a data structure. This is analogous to the concept of potential energy from physics, which is why this approach is also referred to as the physicist's method. ϕ is chosen such that the difference of the potential stored in the data structure, denoted as v, up to the execution of an operation $\phi(v)$ and the potential after the operation $\phi(f(v))$ can be used to "pay" for the actual cost the operation. This allows cheaper operations in a sequence to offset the costs of expensive operations, resulting in lower amortized costs. The amortized cost $a_f(v)$ for an operation f is then expressed by the following equation, where $c_f(v)$ is the actual cost of f, which is offset by the potential difference $\phi(f(v)) - \phi(v)$.

$$a_f(v) = c_f(v) + \phi(f(v)) - \phi(v) \tag{2.1}$$

The key challenge with this type of analysis, is to find a suitable potential function ϕ. Ideally the chosen function minimizes the amortized cost.

In the following, we demonstrate how an analysis with the potential method is conducted on pen and paper. We demonstrate the utility of the potential method on the classical amortized analysis example of the purely functional queue [35]. The queue supports the operations `head`, `tail` and `snoc`, where the name of the latter is an anagram of cons,

which reflects that an element is removed from the back of the queue. This operation in particular makes the implementation in a purely functional setting non-trivial, unlike a pointer-based implementation, which can simply store a reference to the last element in the queue. In order to guarantee constant access to the back of the queue, this data structure is typically implemented with two separate lists F and R. F contains the front elements of the queue, whereas R contains the rear elements in reverse order. We represent this data type as the tuple (F, R). For example, the list of integers $1 \dots 6$ is represented as the queue $(F = [1, 2, 3], R = [6, 5, 4])$. The head of the queue is the first element of F, so the operation `head`, just removes this element from that list and `tail` returns the rest of F, together with R. The implementation of `snoc` is also straight-forward, as a new element can just be added to the the front of R. This description does however not capture, the case in which $F = []$. At this point the list R needs to be reversed and copied to F, before removing the head and setting $R = []$. With this procedure, we can guarantee the invariant that F is only empty if R is also empty, i.e. the whole queue is empty. We give the following implementation for all three operations, together with the additional function `moveToFront`, that is used to maintain the invariant.

```
1  moveToFront :: (List Base * List Base) → (List Base * List Base)
2  moveToFront f r = match r with
3    | [] → (f, [])
4    | cons x xs → moveToFront (∼ cons x f) xs
5
6  head :: (List Base * List Base) → a
7  head f r = match f with
8    | [] → error
9    | cons x xs → ∼ x
10
11 snoc :: (List Base * List Base * a) → (List Base * List Base)
12 snoc f r x = match f with
13   | [] → ∼ (cons x [], [])
14   | f → ∼ (f, cons x r)
15
16 tail :: (List Base * List Base) → (List Base * List Base)
17 tail f r = match f with
18   | [] → ∼ error
19   | cons x xs → ∼ match ff with
20     | [] → moveToFront [] r
21     | xs → (xs, r)
```

The given code is expressed in a custom subset of ML, inspired by Resource Aware ML (RaML) [10]. In this work we utilize the syntax and semantics introduced in the tool ATLAS [29], with minor adaptions. The built-in tick operator $\sim$ is relevant only for the cost analysis and has no influence on evaluation. Its default behavior is to account a cost of one, for evaluating its argument expression. In the given program, as in previous literature, we account for costs associated with each list construction and deconstruction.

The given implementation of `tail`, has a linear worst case complexity in the length of R, due to the call to `moveToFront`. With an amortized analysis though, we can show, however, that the amortized costs are constant. Intuitively, this makes sense because we do not call `moveToFront` for every `tail` operation in a sequence, but only in certain cases.

In order to apply the potential method to the given problem, we choose the potential function $\phi(\texttt{(F, R)}) = |\texttt{R}|$, i.e. the length of the rear list.

We can then apply Equation 2.1 for the tail function.

$$a_{\texttt{tail}}(\texttt{(F, R)}) = c_{\texttt{tail}}(\texttt{(F, R)}) + \phi(\texttt{tail (F, R)}) - \phi(\texttt{(F, R)})$$

We have $c_{\texttt{tail}}(v) = |\texttt{R}| + 1$, since `moveToFront` first traverses `R` completely and finally deconstructs `F` to obtain the tail of the queue. For the initial potential we have $\phi(\texttt{(F, R)}) = |\texttt{R}|$ and after the operation, `R` is empty, so $\phi(\texttt{tail}\, v) = 0$. This gives us an amortized cost of $a_{\texttt{tail}}(\texttt{(F, R)}) = 1$. The `snoc` operation omits a potential difference $\phi(\texttt{snoc (F, R)}) - \phi(\texttt{(F, R)})$ of 1, since $|\texttt{R}| - |\texttt{R}| + 1 = 1$. Therefore its amortized cost is 2. We observe that every `snoc` operation builds up extra potential for every element added to `R`, to eventually pay for `moveToFront`. This observation concludes our manual analysis and we established that all operations of the functional queue have constant amortized costs.

2.2 Automated Amortized Resource Analysis

Now that we have seen how to conduct a pen and paper potential analysis for the purely functional queue, we move on to demonstrate how such an analysis is automated. The technique we will be discussing was developed by Hofmann et al. [19] to enable a fully automated analysis of splay trees. It builds on earlier work by the same author, which focused on linear heap space analysis [17]. To begin with, we revisit Equation 2.1, and generalize it from the single potential function ϕ to a pair of functions ϕ and ψ.

$$\phi(v) \geq c_f(v) + \psi(f(v)) \tag{2.2}$$

To see that this generalization is indeed valid, we rename the original potential function ϕ to ψ and set $\phi(v) := a_f(v) + \psi(v)$.

When analyzing functional programs with this method, the core idea is that the above inequality must hold at every point in the program evaluation. In this context, we assign potential to program states instead of data structure states. An operation corresponds then to an evaluation step of the program under analysis. Therefore the inequality expresses that, $\phi(v)$, the potential before the evaluation of the expression $f(v)$, covers the actual cost of this evaluation $c_f(v)$ and the left-over potential $\psi(f(v))$, held by the result of the evaluation.

With this generalized form, an upper bound on the amortized cost of function f can be directly read of, i.e. we have

$$a_f(v) \leq \phi(v) - \psi(v)$$

So in order to derive the amortized costs, the analysis needs to determine suitable functions ϕ and ψ, i.e. a suitable potential function and the corresponding amortized cost. Once ϕ and ψ have been found, the upper bound on the amortized cost of a function can be computed.

Example 2.1. Assume that an analysis of the function `tail` resulted in $\phi(f, r) = |r| + 1$ and $\psi(\texttt{tail f r}) = |r|$, then $a_{\texttt{tail}} = 1$.

The core idea of an automated amortized analysis in the style of [29] is to determine suitable potential functions for every expression in the program, such that Equation 2.1 holds. These potential functions are represented by type annotations, denoted by $|$, which allows to derive them by type inference. With annotated types Equation 2.1 can be represented as the following type judgment.

$$v : \alpha | \phi \vdash f(v) : \beta | \psi$$

Further, in our setting program states are characterized by *typing contexts*, i.e. mappings from variables to types. So the value v, is actually a typing context Γ. In this sense the following typing judgment represents the potentials of the typing context before the evaluation and potential of resulting expression e.

$$\Gamma | \phi \vdash e : \beta | \psi$$

The functions ϕ and ψ are expressed as *template potential functions*, which we denote by upper case Greek letters, e.g. Φ and Ψ. Those are potential functions of a certain shape with undetermined coefficients. Expressing the potential as templates is crucial to derive a linear constraint system over the unknown coefficients during type inference. A specific instance of a potential function template, denoted as $\Phi(\cdot \mid Q)$, is parameterized by a finite sequence of non-zero coefficients, Q. We refer to such a sequence as a *resource annotation*. The analysis employs a dedicated *type-and-effect system*, in which standard types are annotated with such a resource annotation. The constraints produced by a type derivation based on this system are then solved by an of-the-shelf constraint solver, to obtain the potentials and finally the amortized cost.

Example 2.2. We could for example express the function ϕ for our previous analysis of the functional queue as the template potential function Φ with two coefficients, q_1 and q_2. In this examples q_1 governs the size of of the rear list and q_2 a constant term for the amortized costs.

$$\Phi(f : \mathsf{L}, r : \mathsf{L} \mid [q_1, q_2]) = q_1 \cdot |r| + q_2 \cdot 1$$

Consequently, this refinement brings us to the precise formulation of the type judgment previously introduced.

$$\Gamma | Q \vdash e : \beta | Q'$$

In the following we demonstrate type checking of the function `tail`, with the potential function from our previous pen and paper analysis. For this we will guide the reader step by step through the type derivation and give a basic intuition of the involved type rules. In order to aid readability we refrain from representing template potential as resource annotations, but instead give the functions represented directly.

We start with the type derivation for the function `moveToFront`, as its type is required by the function call in `tail`. The outermost expression of this function is a match statement, so we apply (match) rule, to type the function body.

$$\frac{(2.4) \qquad (2.5)}{f : \mathsf{L}, r : \mathsf{L} || r | \vdash \texttt{match}\ r\ \texttt{with} \mid [] \rightarrow e_1 \mid \texttt{cons}\ x\ xs \rightarrow e_2 : (\mathsf{L}, \mathsf{L}) || r' |}\ (\mathsf{match}) \tag{2.3}$$

When examining the given judgment, we notice, that the amortized costs are zero, since $|r| - |r| = 0$. This supports our previous notion, that this operation uses up the entire potential to pay for its actual costs. The purpose of the (match) rule is to distribute the potential stored in the matched variable according to the corresponding pattern for each case. Equations 2.4 and 2.5 give the type derivations of the `nil` and `cons` case respectively.

$$\dfrac{\dfrac{}{\varnothing|0 \vdash \texttt{[]} : \mathsf{L}||\texttt{[]}|}\,(\mathsf{const}) \qquad \dfrac{}{x_1 : \mathsf{L}||x_1| \vdash \texttt{(f, x1)} : (\mathsf{L},\mathsf{L})||r'|}\,(\mathsf{const})}{f : \mathsf{L}|0 \vdash \texttt{let}\ x_1 = \texttt{[]}\ \texttt{in}\ \texttt{(f, x1)} : (\mathsf{L},\mathsf{L})||r'|}\,(\mathsf{let}) \tag{2.4}$$

We have zero potential to type the `nil` case, since the list r has zero length in this case. In contrast to the listing for `tail`, presented earlier, in this branch we have `let` x_1 = `[] in (f, x1)` instead of just `(f, [])`. Here we observe the result of transforming the given program into *let-normal* form, a common intermediate representation in compilers. This transformation is preformed just before the type derivation and is required by our type system, to handle function composition with the (let) rule. The obligations of (let) include two derivations, one for the expression in the binding, `[]`, and one for the expression in the body, `(f, x1)`. In principle the potential is split according to the occurring variables, but constants can be used up in any of the two derivations. As the binding expression does not include any variables and there is no constants, we have zero potential, when applying the (const) rule. This rule asserts that enough potential is available to construct a data value with the desired potential. In this example we construct the empty list, which requires zero potential, i.e. we omit the trivial constraint $0 \geq 0$. For the body of the let expression we require another application of (const), this time for the tuple `(f, x1)`. In this case the rule needs to check that $|x_1| \geq |r'|$, which holds, given we can rename x_1 to r', as they both denote the second component of the tuple.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{B}, f : \mathsf{L}|1 \vdash \texttt{cons x f} : \mathsf{L}|1}\,(\mathsf{const})}{x : \mathsf{B}, f : \mathsf{L}|1 \vdash (\texttt{cons x f})^{\checkmark} : \mathsf{L}|0}\,(\mathsf{tick}) \qquad \dfrac{}{\begin{array}{l} x_1 : \mathsf{L}, xs : \mathsf{L} \mid |xs| \\ \vdash \texttt{moveToFront x1 xs} \\ : (\mathsf{L},\mathsf{L})||r'| \end{array}}\,(\mathsf{app})}{x : \mathsf{B}, xs : \mathsf{L}, f : \mathsf{L}||xs| + 1 \vdash \texttt{let}\ x_1 = \texttt{cons x f}\ \texttt{in}\ \texttt{(f, x1)} : (\mathsf{L},\mathsf{L})||r'|}\,(\mathsf{let}) \tag{2.5}$$

In the `cons` case we find another let expression introduced by the let-normal form. Note that (match) transfers our initial potential to the tail and a constant, since $|r| = |xs| + 1$. The constant 1 is transferred to the binding, to pay for the costs introduced by the (tick) rule, before this branch is concluded by an application of the (const) rule, to assert $1 \geq 1$. For the body of the let expression the (app) rule is applied. This rule governs function application and assures that the call has the same type as the signature of the called function. This concludes the type derivation for the `moveToFront`. Such a derivation can also be understood as an inductive proof of the amortized costs of the given function, based on its internal structure.

We continue with the type derivation of `tail`. We type it with $|r| + 1$, where $|r|$ is our potential function and 1 is the amortized cost.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{r : \mathsf{L}||r| + 1 \vdash \texttt{(xs, r)} : (\mathsf{L}, \mathsf{L})||r'| + 1}\,(\mathsf{const})}{x : \mathsf{B}, xs : \mathsf{L}, r : \mathsf{L}||r| + 1 \vdash \texttt{match}\ xs\ \texttt{with}\ |\ [] \to e_2\ |\ xs \to\ e_3 : (\mathsf{L}, \mathsf{L})||r'| + 1}\,(\mathsf{match})}{x : \mathsf{B}, xs : \mathsf{L}, r : \mathsf{L}||r| + 1 \vdash (\texttt{match}\ xs\ \texttt{with}\ |\ [] \to e_2\ |\ xs \to\ e_3)^{\checkmark} : (\mathsf{L}, \mathsf{L})||r'|}\,(\mathsf{tick})}{f : \mathsf{L}, r : \mathsf{L}||r| + 1 \vdash \texttt{match}\ f\ \texttt{with}\ |\ [] \to\ \ldots\ |\ \texttt{cons}\ x\ xs\ \to\ e_1 : (\mathsf{L}, \mathsf{L})||r'|}\,(\mathsf{match}) \tag{2.6}$$

As before, we begin with an application of (match). This time the annotation remains the same, as the matched variable, does not carry any potential. We omit the `nil` case for brevity and just note that the `error` constant unifies with any type. In the `cons` case, (tick) again raises the costs by one. We then see another application of the (match) rule, without a modification of the annotation, for the variable xs. For the non-nil case, we do not actually split the potential, so we can apply (const) and verify $|r| + 1 \geq |r'| + 1$, where $r = r'$.

$$\dfrac{(\mathsf{const})\ \dfrac{}{\varnothing|0 \vdash \texttt{[]} : \mathsf{L}|0} \qquad \dfrac{\dfrac{}{r : \mathsf{L}||r| \vdash \texttt{moveToFront x1 r} : (\mathsf{L}, \mathsf{L})||r'|}\,(\mathsf{app})}{r : \mathsf{L}||r| + 1 \vdash \texttt{moveToFront x1 r} : (\mathsf{L}, \mathsf{L})||r'| + 1}\,(\mathsf{shift})}{r : \mathsf{L}||r| + 1 \vdash \texttt{let}\ x_1 = \texttt{[]}\ \texttt{in}\ \texttt{moveToFront x1 r} : (\mathsf{L}, \mathsf{L})||r'| + 1}\,(\mathsf{let}) \tag{2.7}$$

Finally in the `nil` case we have a let expression to bind `[]` for the call to `moveToFront`. Here the full potential is transferred to the body, where the rule (shift) is applied. This rule allows to shift the potential on both sides of the judgment by a constant. The branch concludes with an application of (app). Note that, as required, the type aligns with the signature of `moveToFront`, which we derived earlier. This concludes the type derivation of `tail`. In this example, the choice of rules, with the exception of the rule `shift`, follows directly from the program's syntax. It should be evident that this process can be easily automated. More complex examples require additional structural rules, which, in principle, can be applied at any point during type derivation. To manage this complexity, the type inference algorithm employs heuristics to generate tactics that determine when to apply these rules.

Next, we explain why, in addition to the kind of type derivation we have seen so far, the analysis relies on an alternative type derivation known as *cost-free typing*, which ignores the cost emitted from tick expressions.

2.2.1 Cost-Free Typing

Before we continue with cost-free typing, which is another crucial aspect of AARA, we have to briefly review the logarithmic analysis of splay trees [29]. As stated earlier splay trees are a self-balancing binary trees without an explicit balancing invariant. The operation `splay` restructures the search tree in such a way that the accessed element

is moved closer, to the root, based on the assumption that it will be accessed again soon [42]. For this reason a potential analysis requires a potential function that reflects how well a tree is balanced. Leutgeb et al. deploy a modified version of the rank function rk introduced by Schoenmakers [36]. Let $|t|$, be the number of leaves of tree t, then rk is defined as follows.

$$\mathsf{rk}(\texttt{leaf}) := 1$$
$$\mathsf{rk}(\texttt{node}\ l\ d\ r) := \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)$$

In the analysis the function rk is combined with additional terms to represent logarithmic amortized costs. Using the method described earlier, a bound of $3/2 \log_2(t)$ for the function `splay` and a bound of $2\log_2(t) + 1/2$ for the function `insert`, was inferred. The corresponding potential function was found to be $1/2\,\mathsf{rk}(t) + 1$.

One challenge, present in AARA, is resource-polymorphic recursion. During the analysis a signature for every function in the program is tracked. Take as an example the function `splay`, with the following signature.

$$\texttt{splay} : \mathsf{B} \times \mathsf{T} \mid 3/2 \log_2(t) + \mathsf{rk}(t) + 1 \to \mathsf{T} \mid \mathsf{rk}(t') + 1$$

These signatures are employed to type function calls, but in contrast to standard Hindley-Milner type inference, having a single signature for every function is not sufficient in AARA. The reason for this is that, in some cases, it is necessary to preserve potential through a function call so that it can be used later.

When typing the function `insert`, we initially have $2\log_2(t) + \mathsf{rk}(t) + 3/2$ units of potential available. $3/2 \log_2(t) + \mathsf{rk}(t) + 1$ are spend to invoke `splay` and the left-over potential is $1/2 \log_2(t) + 1/2$, which pays for the rest of the function body. If the application rule would require the potential to exactly match the signature for `splay`, this left-over potential would go to waste, as it can not be transferred to the result of `splay`. For this reason the application rule is defined as follows.

$$\frac{\alpha_1 \times \cdots \times \alpha_n | P \to \beta | P' \in \mathcal{F}(f) \quad \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q' \in \mathcal{F}^{\mathrm{cf}}(f) \quad K \in \mathbb{Q}_0^+}{x_1 : \alpha_1, \ldots, x_n : \alpha_n | (P + K \cdot Q) \vdash f(x_1, \ldots, x_n) : \beta | (P' + K \cdot Q')} \ (\mathsf{app})$$

Here $\mathcal{F}$ donates the cost-incurring signatures of the program under analysis, which corresponds to the signature we have seen above. $\mathcal{F}^{\mathrm{cf}}$ are additional cost-free signatures. A cost-free signature is inferred with the same technique as the cost-incurring signature, with exception that no costs are accounted for. We then allow a function call to be typed with the cost-incurring signature and a scalar multiple of any cost-free signature for this function. This is sound because, first, we only need to take the costs into account once. Secondly, for any cost-free typing the invariant given in Equation 2.2, simplifies to the following inequality, since the cost term $c_f(v)$ is zero.

$$\phi(v) \geq \psi(f(v)) \tag{2.8}$$

Obviously adding a scalar multiple of such an inequality to our original invariant, preserves soundness.

In order to understand how this solves the problem with typing `insert`, take as an example following cost-free signature for `splay`.

$$\texttt{splay}\colon \mathsf{B} \times \mathsf{T} \mid \log_2(t) \to \mathsf{T} \mid \log_2(t')$$

If we interpret the functions ϕ and ψ as size-measuring functions—which is valid since they are required to be non-negative—then a cost-free typing can be viewed as a form of size analysis that satisfies Equation 2.8. For `splay` in particular, we have $\log_2(t) \geq \log_2(t')$, since `splay` only performs rotations and therefore does not increase the size of the tree. Note that this property is established fully automatically through a cost-free analysis. Finally the application rule gives rise to the following judgment for the call to `splay` in `insert`, which preserves the potential $1/2 \log_2(t) + 1/2$, for later use.

$$t\colon \mathsf{T} | 2\log_2(t) + \mathsf{rk}(t) + 3/2 \vdash \texttt{splay a t}\colon \mathsf{T} | \mathsf{rk}\, t + 1/2 \log_2(t) + 3/2$$

2.2.2 Expected Amortized Cost Analysis

Next, we briefly explore an extension of AARA that is also incorporated in our work. Originally proposed by Leutgeb et al. [30], this extension enables the automated analysis of probabilistic programs. They extend their language by introducing an if-then-else expression that models a coin flip with a given probability. To account for expressions that do not always return a concrete value but instead yield a finite probability distribution, the analysis is adapted to incorporate the expected value of an expression, leading to an expected amortized analysis. The corresponding invariant is again a variation of Equation 2.1 and is given by the following inequality, where $f(v)$ evaluates to the distribution μ.

$$\phi(v) \geq c + \mathbb{E}_{\mu}(\lambda w.\psi(w))$$

Of course, this change requires a semantics that properly accounts for the evaluation of probabilistic expressions. In the original work, both a small-step and a big-step semantics are provided. However, we base our work on the big-step version, which we present in Chapter 3.

Additionally they introduce an alternative cost-free typing denoted by the judgment $\vdash^{\mathrm{cf}}_{\mathrm{ndt}}$. For such a typing the invariant is relaxed back to $\phi(v) \geq \psi(f(v))$, because during the analysis every coin flip is replaced by a non-deterministic choice. This addition is essential to ensure the soundness of the (let) rule. When we look at the cost-free typing as a size-analysis as before, this corresponds to taking the non-deterministic maximum over the different branches in the computation, which gives an upper bound on the size of the resulting data structure.

2.3 Mixed Potentials

In this section we outline the main contribution of this thesis. Consider the function `fromList` given in Figure 2.1. It inserts the items of the given list `l` into an empty splay

Figure 2.1: Naive implementation of `SplayTree.fromList`.

```
1 fromList :: List Base  → Tree Base
2 fromList l = match l with
3   | [] → leaf
4   | cons x xs → insert x (fromList xs)
```

tree. We know that the amortized cost of a single insertion is $O(\log_2(|\mathtt{t}|))$, so the cost of `fromList` should be $O(|\mathtt{l}| \cdot \log_2(|\mathtt{t}|)) = O(|\mathtt{l}| \cdot \log_2(|\mathtt{l}|))$. This means that the amortized cost is expressed in terms of the list `l`. If we assume for a moment that we can represent this cost with a suitable template potential function defined on lists, we still have to pay potential for every call to `insert`, which is typed with a logarithmic potential function for its tree argument.

Leutgeb et al. only allow for tree types and logarithmic potential. This restriction is lifted in our work by utilizing a potential agnostic framework, to support different data types and potential functions. For the previous example of the functional queue, we employ one specific instance of this generic framework, that handles polynomial potential of lists. We describe this approach in detail in Chapter 4. Unfortunately even with such a generic framework it is still only possible to analyze one data type at a time. In the following we motivate a further modification based on mixed potential functions. It allows to have different potential functions for every data type in the program. A mixed potential function is than constructed as the sum of the individual functions. We can, however, not apply this approach directly to our specification of `fromList`. Remember that the bound on the amortized cost in the automated amortized analysis is computed according to the following equation.

$$a_f(v) \leq \phi(v) - \psi(v)$$

Which in this specific case, applying mixed potential functions, reads

$$a_{\mathtt{fromList}}(l : \mathsf{L}) \leq (\Phi_{\mathsf{L}}(l : \mathsf{L}) + \Phi_{\mathsf{T}}(\varnothing)) - (\Psi_{\mathsf{L}}(l : \mathsf{L}) + \Psi_{\mathsf{T}}(\varnothing)) \ .$$

Here, the individual potential components are denoted with a subscript for the data type. By definition, Ψ is the potential function applied in the potential method. $\Phi_{\mathsf{T}}(\varnothing)$ contains only constant basic potential functions, therefore we cannot express the potential $1/2\,\mathsf{rk}(t) + 1$ required by `insert`. We can however explicitly provide the required potential by introducing a second argument `t`, which takes a tree. Figure 2.2 gives the adapted implementation of the function. Note that we make sure that the tree is `leaf`. This is required, as otherwise we could not guarantee that the resulting tree has the same length as the input list and therefore we could not assume $O(|\mathtt{l}| \cdot \log_2(|\mathtt{t}|)) = O(|\mathtt{l}| \cdot \log_2(|\mathtt{l}|))$.

Using mixed potential we can then express the following judgment for `fromList`.

$$(l : \mathsf{L} \mid \underbrace{Q_{\mathsf{L}}}_{|l| \cdot \log_2 |l| + 1/2 |l|}), (t : \mathsf{T} \mid \underbrace{Q_{\mathsf{T}}}_{1/2\,\mathsf{rk}(t)+1}) \vdash \mathtt{fromList\ l\ t} : \mathsf{T} \mid \underbrace{Q'_{\mathsf{T}}}_{1/2\,\mathsf{rk}(t)+1}$$

Note that we have separate type annotations for the two data types in the signature. It is also evident that the log-linear potential is just used to express the amortized costs,

Figure 2.2: Code for `SplayTree.fromList`.

```
1 fromList :: (List Base * Tree Base) → Tree Base
2 fromList l t = match t with
3   | node x1 _ x2 → error
4   | leaf → match l with
5     | [] → leaf
6     | cons x xs → insert x (fromList xs leaf)
```

whereas the actual potential function is $1/2\,\mathsf{rk}(t)+1$. Like in previous works we are able to derive this signature fully automatically. The framework for mixed potential is presented in Chapter 4 and an in-depth explanation of the type derivation for `fromList`, including the necessary additions to the underlying method, is given as a separate case study in Chapter 5.

3 Preliminaries

Before delving into the details of our generalized analysis, this chapter provides a formal introduction to several core underlying concepts. It serves both as a review of the existing method and as an opportunity to introduce new notation essential for our generalization.

We begin by introducing template potential functions, which act as type annotations and form the foundation of automated amortized analysis. Next, we define resource annotations, finite instances of template potential functions consisting of unknown coefficients to be determined by a constraint solver.

Thereafter, we revisit the essential type rule (w), introduced by Hofmann et al. [19], alongside the linearization technique required to perform the symbolic comparisons necessary for its application.

Finally, we present the functional language that is the subject of our analysis and provide its corresponding big-step semantics.

3.1 Template Potential Functions

As mentioned before the automated amortized analysis relies on template potential functions, for which coefficients can be determined by solving a linear constraint system. In general the shape of these templates determines the kind of bound that can be established. Finding a bound of $\log_2(n)$, for example, requires logarithmic terms, whereas a bound of n^2 has polynomial terms. In this way the complexity of the template increases the expressivity of the analysis, but limits its efficiency, since more constraints need to be solved. Another problem is that functions like $\log_2$, require careful design considerations to handle non-linearity. Additionally templates need be closed under the basic operations of the programming language. For example supporting list construction requires the potential to be closed under addition of one.

In this section, we first provide a general definition of template potential functions, which is complemented by the introduction of specific potential functions tailored to concrete analyses, such as the logarithmic analysis, in Chapter 4.

First assume we have types $\alpha, \beta, \gamma, \ldots$ representing sets of values. We then denote $[\![\alpha]\!]$ as the set of values of a type α. For each type α we define a set of *basic potential functions*, $\mathcal{BF}(\alpha) : [\![\alpha]\!] \to \mathbb{Q}_0^+$. An *annotated type* is defined as pair of a type α and an *annotation*, $Q : \mathcal{BF} \to \mathbb{Q}_0^+$ which provides of a coefficient for each basic potential function.

Definition 3.1. For each annotated type $\alpha \mid Q$ a *potential function* $\Phi : [\![\alpha]\!] \to \mathbb{Q}_0^+$ is

defined as a linear combination of basic potential functions, with coefficients from Q.

$$\Phi(v \mid Q) := \sum_{\phi \in \mathcal{BF}(\alpha)} Q(\phi) \cdot \phi(v)$$

Extending our definition of basic potential functions to product types $\mathcal{BF}(\{\alpha_1 \times \cdots \times \alpha_n\}) : [\![\alpha_1]\!] \times \cdots \times [\![\alpha_n]\!] \to \mathbb{Q}_0^+$ allows the definition of multivariate potential functions.

Definition 3.2. For each annotated type $\alpha_1 \times \cdots \times \alpha_n \mid Q$ a *multivariate potential function* $\Phi : [\![\alpha_1]\!] \times \cdots \times [\![\alpha_n]\!] \to \mathbb{Q}_0^+$ is given by

$$\Phi(v_1, \ldots, v_n \mid Q) := \sum_{\phi \in \mathcal{BF}(\alpha_1 \times \cdots \times \alpha_n)} Q(\phi) \cdot \phi(v_1, \ldots, v_n)$$

Note that $\mathcal{BF}(\{\alpha_1 \times \cdots \times \alpha_n\})$ can include both univariate and multivariate basic potential functions. This means that a potential function can have terms that depend on a single argument and terms that include multiple arguments, which results in more precise bounds. In the following when speaking about potential functions, we always refer to multivariate potential functions and therefore omit "multivariate".

3.2 Resource Annotations

Whereas the notion of an annotation as seen above, was defined quite abstract as mapping that provides coefficients for each basic potential function, in the following we require a more precise definition. In particular we will introduce an indexing scheme that lets us refer to individual coefficients in a uniform way. Throughout previous work in the area of AARA, a number of different indexing schemes with various degrees of complexity have been devised, e.g. [9, 19]. With this version we aim to distinguish different basic potential functions, based on the set of their arguments. Additionally we include for every variable a *degree*, that encodes some of the internal structure of the addressed function. Assume for example, we have basic potential functions of the form $\phi(x) = |x|^a$. Here the degree would denote the exponent. For additional flexibility the degree is defined as a sequence of natural numbers rather than a single number.

Definition 3.3. We define the index set I_n of variables $X = \{x_1, \ldots, x_k\}$ as follows.

$$I_n(X) = \left\{ \{x_1^{a_1}, \ldots, x_k^{a_k}\} \mid a_1, \ldots, a_k \in \mathbb{N}^n \right\}$$

In the remainder of the text, we use I_n without explicitly specifying n, assuming that the appropriate version is inferred from the context.

Definition 3.4. Let $\mathsf{vars}(\Gamma)$ denote the variables of typing context Γ. A resource annotation Q for a typing context Γ is then defined as a sequence

$$Q = [q_{(x)} \mid x \in \mathsf{vars}(\Gamma)] \cup [(q_{(\vec{x} \cup \{c\})}) \mid \vec{x} \in I(\mathsf{vars}(\Gamma)), c \in \mathbb{N}]$$

with only finitely many coefficients $q_{(i)} \neq 0$.

In the following we omit set notation for coefficient indices and write $q_{(x^{\vec{a}},y^{\vec{b}},\ldots,c)}$ directly instead of $q_{(\{x^{\vec{a}},y^{\vec{b}},\ldots,c\})}$, as well as $q_{(\vec{x},c)}$, instead of $q_{(\vec{x}\cup\{c\})}$. We also abbreviate $q_{(x^{(a)})}$ as $q_{(x^a)}$ and associate $q_{(\vec{x},x^{\vec{0}})} = q_{(\vec{x})}$.

Next we define arithmetic operations on resource annotations, which are needed in the premises of the typing rules. The first one is adding a constant to an annotation, which is defined by adding the constant to the coefficient for the constant potential function $\phi(x_1,\ldots,x_n) = 1$, denoted as $\mathsf{one}(Q)$, for any resource annotation Q.

Definition 3.5. We define $Q + k$ as

$$Q + k := (Q \setminus \mathsf{one}(Q)) \cup (\mathsf{one}(Q) + k)$$

We also define for resource annotations Q, P multiplication, denoted by $Q \cdot P$, and addition, denoted by $Q + P$, pointwise.

For brevity, in the following, we introduce notation, that allows us to restrict the basic potential functions in a potential function.

Definition 3.6. Let Φ be a potential function. Then $\Phi^{nc}(\Gamma \mid Q)$ restricts the basic potential functions in Φ to non-constant indices.

$$\Phi^{nc}(\Gamma \mid Q) := \sum_{x\in\mathsf{vars}(\Gamma)} q_{(x)}\cdot\phi_{(x)} + \sum_{\substack{\vec{x}\in I(\mathsf{vars}(\Gamma)),\vec{x}\neq\emptyset,\\ c\in\mathbb{N}}} q_{(\vec{x},c)}\cdot\phi_{(\vec{x},c)}$$

Definition 3.7. Let Φ be a potential function. Then $\Phi^{v}(\Gamma \mid Q)$ restricts the basic potential functions in Φ to indices only containing variables.

$$\Phi^{v}(\Gamma \mid Q) := \sum_{x\in\mathsf{vars}(\Gamma)} q_{(x)}\cdot\phi_{(x)} + \sum_{\vec{x}\in I(\mathsf{vars}(\Gamma)),\vec{x}\neq\emptyset} q_{(\vec{x})}\cdot\phi_{(\vec{x})}$$

Definition 3.8. Let Φ be a potential function. Then $\Phi^{\times}(\Gamma,\Delta \mid Q)$ restricts the basic potential functions in Φ to indices containing variables from both Γ and Δ.

$$\Phi^{\times}(\Gamma,\Delta \mid Q) := \sum_{\substack{\vec{x}\in I(\mathsf{vars}(\Gamma)),\vec{x}\neq\emptyset,\\ \vec{y}\in I(\mathsf{vars}(\Delta)),\vec{y}\neq\emptyset,\\ c\in\mathbb{N}}} q_{(\vec{x},\vec{y},c)}\cdot\phi_{(\vec{x},\vec{y},c)}$$

Definition 3.9. Let Φ be a potential function. Then $\Phi^{\times_\Delta}(\Gamma,\Delta \mid Q)$ restricts the basic potential functions in Φ to indices containing variables Δ, together with variables from Γ or just constants.

$$\Phi^{\times_\Delta}(\Gamma,\Delta \mid Q) := \sum_{\substack{\vec{x}\in I(\mathsf{vars}(\Gamma)),\\ \vec{y}\in I(\mathsf{vars}(\Delta)),\vec{y}\neq\emptyset,\\ c\in\mathbb{N},\\ \vec{x}\neq\emptyset\vee c\neq 0}} q_{(\vec{x},\vec{y},c)}\cdot\phi_{(\vec{x},\vec{y},c)}$$

3.3 Linearization and Expert Knowledge

As we will observe in the following chapter, some basic facts about potential functions are integrated into the analysis through the data constructors of the language. These facts express how the potential changes when values are constructed or deconstructed, based on the associated potential function. These fundamental properties, in combination with basic coefficient comparison, already enable powerful reasoning — for instance, they suffice for the polynomial analysis performed by RaML [10].

However, for the analysis of trees with logarithmic potential, Hofmann et al. [19] observed that additional facts are required. To better understand this, let us first review the (w) rule from their type system.

$$\frac{\Gamma|P \vdash e:\alpha|P' \quad \Phi(\Gamma|P) \leqslant \Phi(\Gamma|Q) \quad \Phi(\Gamma|P') \geqslant \Phi(\Gamma|Q')}{\Gamma|Q \vdash e:\alpha|Q'} \ (\mathsf{w})$$

This rule allows to weaken assumptions at any point, in a type derivation. In the polynomial analysis defined by Hoffmann [9], it suffices to compare the coefficients of annotations P and Q, to discharge $\Phi(\Gamma|P) \leqslant \Phi(\Gamma|Q)$. When we have logarithmic basic potential functions, this is no longer the case. For this reason, Hofmann et al. introduced a novel technique that systematically linearizes different properties of the logarithmic function, transforming them into an existential constraint satisfaction problem. This symbolic comparison is based on the following variant of Farkas' Lemma. For a proof of this lemma we refer the reader to [19].

Lemma 3.10 (Farkas' Lemma)**.** *Suppose $A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant 0$ is solvable. Then the following assertions are equivalent. (i) $\forall \vec{x} \geqslant 0.\ A\vec{x} \leqslant \vec{b} \Rightarrow \vec{u}^T\vec{x} \leqslant \lambda$ and (ii) $\exists \vec{f} \geqslant 0.\ \vec{u}^T \leqslant \vec{f}^T A \wedge \vec{f}^T\vec{b} \leqslant \lambda$.*

According to our definition of template potential functions, the inequality $\Phi(\Gamma|P) \leqslant \Phi(\Gamma|Q)$ can be represented by

$$\vec{p}^T\vec{x} + c_p \leq \vec{q}^T\vec{x} + c_q \ ,$$

where the variables $\vec{x}$ stand for the non-constant basic potential functions of Φ, and c_p and c_q denote the constant potential basic functions respectively. $(A, \vec{b})$, subject to $A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant 0$, can now be interpreted as expert knowledge about the basic potential functions $\vec{x}$, as every row represents a corresponding inequality. We can then reformulate our original problem, to incorporate this expert knowledge and satisfy the assumptions of Lemma 3.10 by setting $\vec{u}^T = \vec{p}^T - \vec{q}^T$.

$$\forall \vec{x} \geq 0.\ A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant 0 \Rightarrow (\vec{p}^T - \vec{q}^T)\vec{x} \leq c_q - c_p$$

By Farkas' lemma we then obtain the following equivalent constraints.

$$\exists \vec{f} \geq 0.\ \vec{p}^T \leqslant \vec{f}^T A + \vec{q}^T \wedge \vec{f}^T\vec{b} + c_p \leqslant c_q$$

This alternative formulation, allows the constraint solver to utilize the weakening rule, not only to decrease potential, but also to shift it horizontally between the different

$$
\begin{array}{lll}
\circ ::= \texttt{<} \mid \texttt{>} \mid \texttt{=} & & \\
c ::= \texttt{false} \mid \texttt{true} \mid \texttt{error} & & \\
\quad \mid \texttt{leaf} \mid \texttt{node}\ x_1\ x_2\ x_3 & & \\
\quad \mid \texttt{nil} \mid \texttt{cons}\ x_1\ x_2 & & \\
\quad \mid \texttt{(}\ x_1\ \texttt{,}\ x_2\ \texttt{)} & & \\
e ::= x & & f\ x_1\ \ldots\ x_n \\
\quad \mid e^{\checkmark\, a/b} & & e_1 \circ e_2 \\
\quad \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & & \mid \texttt{if nondet then}\ e_1\ \texttt{else}\ e_2 \\
\quad \mid \texttt{if coin}\ a/b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & & \mid \texttt{match}\ x\ \texttt{with}\ \{\mid c \to e\} \\
\quad \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & &
\end{array}
$$

Figure 3.1: A Core Probabilistic (First-Order) Programming Language

basic potential functions. This property is crucial for the logarithmic analysis which we describe in detail in the next chapter. To conclude this section, we demonstrate the described linearization technique with following example.

Example 3.11. Assume we want to utilize the basic fact $\forall x, y \geq 1.\ \log_2(x+1) \leq \log_2(x+y)$ to discharge the obligations of the (w) rule. In the following we denote the function $\log_2(x+1)$, with $x_{(x,1)}$ and $\log_2(x+y)$ as $x_{(x,y)}$, as well as their corresponding coefficients in the potential function Φ as $q_{(x,1)}$ and $q_{(x,y)}$. The expert knowledge can than be represented as the matrix $A = (1, -1)$, and $\vec{b} = 0$. We can now instantiate Farkas' Lemma and since $\vec{b} = 0$, we omit the constraint $\vec{f}^T\vec{b} + c_p \leqslant c_q$ and are left with the following constraints.

$$
\begin{aligned}
p_{(x,1)} &\leq q_{(x,1)} + f \\
p_{(x,y)} &\leq q_{(x,y)} - f
\end{aligned}
$$

So choosing an $f \geq 0$, allows to shift potential from $\log_2(x+y)$ to $\log_2(x+1)$.

3.4 Language and Semantics

In this section we first introduce the syntax of the functional language which is the subject of our analysis and give a suitable big-step semantics for it.

We largely adopt the language established by Leutgeb et al. [30], as shown in Figure 3.1. It is a first-order language with syntax similar to ML, supporting probabilistic branching, via the built-in expression `coin a/b`. The most notable change, we introduced, is the generic notion of data constructors. With this version `match` can now deconstruct arbitrary data types, instead of being restricted to just `leaf` and `node` for trees. As part of this change, we added constructors for lists, as well as an error constant, that unifies with every type, to aid the implementation of partial functions. At first glance, not allowing

$$\frac{e \text{ is not a value}}{\sigma \left|\frac{0}{0}\right. e \Rightarrow \{\}} \qquad \frac{\mathsf{c} \text{ is an } n\text{-ary data constructor for type A} \quad x_1\sigma = v_1, \ldots, x_n\sigma = v_n}{\sigma \left|\frac{0}{0}\right. \mathsf{c}\, x_1 \ \ldots \ x_n \Rightarrow \{(\mathsf{c}\, v_1 \ \ldots \ v_n)^1\}} \qquad \frac{x\sigma = v}{\sigma \left|\frac{0}{0}\right. x \Rightarrow \{v^1\}}$$

$$\frac{f \ y_1 \ \ldots \ y_k \ \texttt{=} \ e \in \mathsf{P} \quad \sigma' \left|\frac{c}{n}\right. e \Rightarrow \mu}{\sigma \left|\frac{c}{n+1}\right. f \ x_1 \ \ldots \ x_k \Rightarrow \mu}$$

$$\frac{\sigma \left|\frac{c_1}{n}\right. e_1 \Rightarrow \nu \quad \text{for all } w \in \mathsf{supp}(\nu)\text{: } \sigma[x \mapsto w] \left|\frac{c_w}{n}\right. e_2 \Rightarrow \mu_w}{\sigma \left|\frac{c_1 + \mathbb{E}_\nu(\lambda w.c_w)}{n+1}\right. \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow \sum_{w \in \mathsf{supp}(\nu)} \nu(w) \cdot \mu_w}$$

$$\frac{\begin{array}{c}\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \\ x : \alpha \qquad x\sigma = \mathsf{c}\, v_1 \ \ldots, v_n \qquad \sigma'' \left|\frac{c}{n}\right. e \Rightarrow \mu\end{array}}{\begin{array}{l}\sigma \left|\frac{c}{n+1}\right. \texttt{match } x \texttt{ with } \ldots \\ \qquad \texttt{| } (\mathsf{c}\, x_1 \ \ldots \ x_n) \texttt{ -> } e \\ \qquad \ldots\end{array} \Rightarrow \mu}$$

$$\frac{x\sigma = \texttt{true} \quad \sigma \left|\frac{c}{n}\right. e_1 \Rightarrow \mu}{\sigma \left|\frac{c}{n+1}\right. \texttt{if } x \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow \mu} \qquad \frac{\sigma \left|\frac{c}{n}\right. e_1 \Rightarrow \mu}{\sigma \left|\frac{c}{n+1}\right. \texttt{if nondet then } e_1 \texttt{ else } e_2 \Rightarrow \mu}$$

$$\frac{x\sigma = \texttt{false} \quad \sigma \left|\frac{c}{n}\right. e_2 \Rightarrow \mu}{\sigma \left|\frac{c}{n+1}\right. \texttt{if } x \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow \mu} \qquad \frac{\sigma \left|\frac{c}{n}\right. e_2 \Rightarrow \mu}{\sigma \left|\frac{c}{n+1}\right. \texttt{if nondet then } e_1 \texttt{ else } e_2 \Rightarrow \mu}$$

$$\frac{\sigma \left|\frac{c}{n}\right. e \Rightarrow \mu}{\sigma \left|\frac{c+|\mu|\cdot a/b}{n+1}\right. e^{\checkmark\, a/b} \Rightarrow \mu} \qquad \frac{\sigma \left|\frac{c_1}{n}\right. e_1 \Rightarrow \mu_1 \quad \sigma \left|\frac{c_2}{n}\right. e_2 \Rightarrow \mu_2 \quad p = a/b}{\sigma \left|\frac{pc_1 + (1-p)c_2}{n+1}\right. \texttt{if coin } a/b \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow p\mu_1 + (1-p)\mu_2}$$

Here, $\sigma[x \mapsto w]$ represents the environment σ updated so that $\sigma[x \mapsto w](x) = w$, while all other variable mappings remain unchanged. For function application, we define σ' as $\sigma \uplus \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. In the rules handling `match`, we define $\sigma'' := \sigma \uplus \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ [30].

Figure 3.2: Big-Step Semantics.

higher order functions, might seem very restricting, but it turns out that in many cases they can be avoided and in fact all our benchmarks can be implemented.

Next we define the data types used in our analysis. Trees are denoted by $\mathsf{T}\ \alpha$ and lists by $\mathsf{L}\ \alpha$, where α is a type variable. Additionally, we introduce a placeholder type B, representing the "basic" elements of lists and trees. This type is never subjected to a potential function and can, for example, be replaced with integers without affecting the analysis. For lists and trees we mostly omit the element type and assume it to be B, i.e. $\mathsf{T} = \mathsf{T}\ \mathsf{B}$. Lastly booleans are donated by the type Bool.

A big-step semantics for our language is given in Figure 3.2. In contrast to [30] we omit the definition of a small-step semantic, because we chose to only implement the defer variant of the tick rule, presented in that work.

4 Multi-Potential Analysis

In this chapter, we provide a formal explanation of our main contribution: the multi-potential analysis, which was introduced in Chapter 2. In Section 4.1, we define an abstract analysis framework that is independent of the potential function used in a specific amortized analysis. This framework can be regarded as a generalized version of the type system presented in [29]. Next, in Section 4.2, we demonstrate how to instantiate this framework. We also provide concrete instances for logarithmic, polynomial, and log-linear potentials. Finally, in Section 4.3, we conclude the chapter with mixed potentials, a modification of the previous type system that enables the combination of different potential functions in an additive manner.

4.1 Abstract Framework

In order to obtain an analysis that supports multiple potential functions and data types, we introduce a new type system, that abstracts away the specifics of the employed potential function. This is achieved by expressing the premises of type rules at the level of potential functions, rather than imposing constraints on the coefficients, as done in previous works [19, 29, 30].

In the following we explain the more interesting rules in detail and refer to Figure 4.1 and Figure 4.2 for a complete list. First we lift the restriction to one data type, by replacing the previous (leaf) and (node) rules with a generic constructor rule (const).

$$\frac{\begin{array}{c}\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \\ \Phi(x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid Q) = \Phi(\mathsf{c}\, x_1 \ \ldots \ x_n : \alpha \mid Q')\end{array}}{x_1 : \alpha, \ldots, x_n : \alpha_n | Q \vdash \mathsf{c}\, x_1 \ \ldots \ x_n : \alpha | Q'} \ (\mathsf{const})$$

This rule expresses that a constructed value has the potential of its constructor arguments. In the case of a zero-ary constructor, e.g `leaf`, the potential transferred to the new value is constant. Note that the previous (node) rule and (leaf) rules are instances of (const), when we restrict our analysis to tree types.

Building on the adaptation of the (const) rule, we also modify the (match) rule to support arbitrary data types. Instead of only considering trees which deconstruct to `node` l v r and `leaf`, we now have generic constructor patterns.

$$\frac{\Phi(\Gamma, \mathsf{c}_i\, \vec{x_i} \mid Q) = \Phi(\Gamma, \vec{x_i} \mid Q_i) \qquad \Gamma, \vec{x_i} | Q_i \vdash e_i : \beta | Q'}{\Gamma, x : \alpha | Q \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \mathsf{c}_1\, \vec{x_1}\ \texttt{->}\ e_1 \texttt{|}\ \ldots\ \texttt{|}\ \mathsf{c}_m\, \vec{x_m}\ \texttt{->}\ e_m : \beta | Q'} \ (\mathsf{match})$$

The (match) rule, guarantees that the potential, freed by the deconstruction of the data value, is distributed correctly to the pattern variables.

In previous work, there is a rule for typing control operators like <. Since these operators do not manipulate trees, this rule simply requires that the potential remains unchanged. We generalize this principle into an application rule, (app : base), for functions that neither emit costs nor alter the potential of their arguments or return types.

$$\frac{a_i \neq T \qquad \beta \neq T \qquad \Phi(x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid Q) = \Phi(f(x_1, \ldots, x_n) : \beta \mid Q')}{x_1 : \alpha_1, \ldots, x_n : \alpha_n | Q \vdash f(x_1, \ldots, x_n) : \beta | Q'} \text{ (app : base)}$$

In the premises of this rule T denotes the potential bearing type of the analysis. (app : base) allows to type built-in functions, such as arithmetic expressions. For example, since numbers in our analysis are never subjected to a potential function, the expression $k - 1$ does not alter potential and can therefore be treated in the same way as comparison operators were before.

The most involved rule in the type system is the (let) rule.

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \qquad \Phi^{v}(\Delta \mid R) = \Phi^{v}(\Delta \mid Q) \\ \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \qquad \Phi^{nc}(x : \alpha \mid R) = \Phi^{nc}(e_1 : \alpha | P') \\ \Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x : \alpha, \Delta \mid R) \qquad \Gamma | P \vdash e_1 : \alpha | P' \qquad \Delta, x : \alpha | R \vdash e_2 : \beta | Q'\end{array}}{\Gamma, \Delta | Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \beta | Q'} \text{ (let)}$$

The purpose of this rule is to transfer the initial potential $\Phi(\Gamma, \Delta \mid Q)$ to $\Phi(x : \alpha, \Delta \mid R)$, the potential for evaluating e_2, while respecting the potential consumed in e_1. In the premises, we split the initial context into Γ, which contains all free variables of e_1 and Δ, which contains the remaining variables. Building on this separation, the initial potential can be decomposed according to the following equation.

$$\Phi(\Gamma, \Delta \mid Q) = \Phi^{nc}(\Gamma \mid P) + \Phi^{v}(\Delta \mid Q) + \Phi^{\times\Delta}(\Gamma, \Delta \mid Q) + \Phi(\varnothing \mid Q)$$

Recall that $\Phi^{nc}(\Gamma \mid P)$ denotes the basic potential functions consisting of variables or constants from Γ, which means they only depend on Γ. $\Phi^{v}(\Delta \mid Q)$ denotes basic potential functions consisting of terms with variables from Δ and no constants, so this part only depends on Δ. $\Phi(\varnothing \mid Q)$, the potential of an empty typing context consists only of constants. And finally we recall that $\Phi^{\times\Delta}(\Gamma, \Delta \mid Q)$, denotes terms with variables from Δ together with either variables from Γ or constants, meaning this term depends on both variables from Γ and Δ. Note that we have $\Phi^{v}(\Delta \mid Q)$, instead of $\Phi^{nc}(\Delta \mid Q)$, since terms with constants are covered in $\Phi^{\times\Delta}(\Gamma, \Delta \mid Q)$.

The potential covering the evaluation of e_2, is composed in a similar way of potential that depends solely on Δ, the potential obtained by evaluating e_1 and constant potential.

The last term again is a combination of the variable x and the variables in Δ.

$$\Phi(x:\alpha, \Delta \mid R) = \Phi^{nc}(x:\alpha \mid R) + \Phi^{v}(\Delta \mid R) + \Phi^{\times\Delta}(x:\alpha, \Delta \mid R) + \Phi(\varnothing \mid R)$$

In order to understand how (let) enforces the correct potential flow, first consider that it transfers the potential $\Phi^{nc}(\Gamma|P)$ to $\Phi^{nc}(e_1:\alpha|P')$, which is assigned to $\Phi^{nc}(x:\alpha \mid R)$, via the judgment $\Gamma|P \vdash e_1:\alpha|P'$. The potential $\Phi^{v}(\Delta \mid Q)$ is not affected by e_1 and is directly assigned to $\Phi^{v}(\Delta \mid R)$. Any constant potential is split according to the following equation.

$$\Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P')$$

It expresses that the constant potential for evaluating e_2 consists of the part of the input potential that was not used to evaluate e_1 and the leftover potential after evaluating e_1. For the shared terms it is sufficient to guarantee that the resulting potential is covered by the remaining initial potential, i.e. $\Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x:\alpha, \Delta \mid R)$. This constraint is an explicit premise of the rule, but can usually be discharged with a number of cost-free typings, a method we describe in the next section.

As in the original formulation, we additionally provide a simplified version of (let).

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \qquad \Phi^{nc}(\Delta \mid R) = \Phi^{nc}(\Delta \mid Q) \\ \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \\ \Gamma|P \vdash e_1:\alpha|P' \qquad \Delta, x:\alpha|R \vdash e_2:\beta|Q' \qquad \alpha \neq T\end{array}}{\Gamma, \Delta|Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2:\beta|Q'}\ (\mathsf{let}:\mathsf{base})$$

The rule (let : base) applies when the type of the value bound in the let expression does not bear any potential. In this case, the terms $\Phi^{nc}(x:\alpha \mid P')$ and $\Phi^{\times\Delta}(x:\alpha, \Delta \mid R)$ are both zero and can be disregarded in the premises. Specifically, it is not necessary to ensure that $\Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x:\alpha, \Delta \mid R)$.

In addition to the syntax directed rules, there are also structural rules, that can be applied at any point of a type derivation. For a detailed explanation we refer to [29].

In contrast to previous works we introduce the rule (shiftm), which is an alteration of (shift), that allows to shift the arguments of a single basic potential function by a constant, in a cost-free derivation.

For example $\log_2(x)$ could be shifted to $\log_2(x + c)$, where $c \in \mathbb{N}$. This change is required to derive a typing for the `fromList` function. We will elaborate on this fact in great in detail in Chapter 5.

The premises of the rule are defined as follows.

$$\frac{\begin{array}{c}\Gamma|P \vdash^{\mathrm{cf}} e:\alpha|P' \qquad \phi \text{ is monotone} \\ \Phi(\Gamma \mid P) = q_\phi \cdot \phi(x) + c \qquad \Phi(\Gamma \mid Q) = q_\phi \cdot \phi(x + k) + c \\ \Phi(e:\alpha \mid P') = q_\phi \cdot \phi(y) + c \qquad \Phi(e:\alpha \mid Q') = q_\phi \cdot \phi(y + k) + c\end{array}}{\Gamma|Q \vdash^{\mathrm{cf}} e:\alpha|Q'}\ (\mathsf{shiftm})$$

Note that both Q and Q' consist only of the same basic potential function ϕ and a constant c, i.e the obligations of rule demand that all coefficients other than q_ϕ are zero. This condition, together with the requirement that ϕ is monotone is essential for the soundness the (shiftm) rule.

We conclude this section by turning our attention to the soundness of the generalized analysis. First, we present the definition of a *well-typed* program in the sense of the analysis defined by Hofmann et al. [19].

Definition 4.1. A program P is called *well-typed* if for any definition $f(x_1, \ldots, x_k) = e \in \mathsf{P}$ and any annotated signature $\alpha_1 \times \cdots \times \alpha_k | Q \to \beta | Q' \in \mathcal{F}(f)$, we have $x_1 : \alpha_1, \ldots, x_k : \alpha_k | Q \vdash e : \beta | Q'$, as well as for any annotated signature $\alpha_1 \times \cdots \times \alpha_k | Q \to \beta | Q' \in \mathcal{F}^{\mathrm{cf}}(f)$, we have $x_1 : \alpha_1, \ldots, x_k : \alpha_k | Q \vdash^{\mathrm{cf}} e : \beta | Q'$.

In the following we restate the soundness theorem that Leutgeb et al. [30] formulated for their probabilistic analysis. We proceed to proof the theorem for the abstract type system.

Theorem 4.2. *Soundness Theorem. Let P be well-typed and let σ be a substitution. Suppose $\Gamma | Q \vdash e : \alpha | Q'$ and $\sigma \overset{c}{\vdash} e \Rightarrow \mu$. Then, we have $\Phi(\sigma; \Gamma \mid Q) \geq c + \mathbb{E}_\mu(\lambda v. \Phi(v \mid Q'))$. Further, if $\Gamma | Q \vdash^{cf} e : \alpha | Q'$, then $\Phi(\sigma; \Gamma \mid Q) \geq \Phi(v \mid Q')$*

Note that in contrast to Leutgeb et al. we require $\Phi(\sigma; \Gamma \mid Q) \geq \Phi(v \mid Q')$ for $\Gamma | Q \vdash^{\mathrm{cf}} e : \alpha | Q'$, instead of $\Phi(\sigma; \Gamma \mid Q) \geq \mathbb{E}_\mu(\lambda v. \Phi(v \mid Q'))$. As described in Chapter 2, $\vdash^{\mathrm{cf}}_{\mathrm{ndt}}$ corresponds to a cost-free analysis, in which the actual program is abstracted by replacing each coin flip with a non-deterministic choice, whereas $\vdash^{\mathrm{cf}}$ corresponds to a standard cost-free analysis, in which the rule (ite : coin) is utilized. We found that distinguishing between $\vdash^{\mathrm{cf}}_{\mathrm{ndt}}$ and $\vdash^{\mathrm{cf}}$ is unnecessary, as in fact, only the former is actually required. This approach can also be viewed as considering the maximum possible outcome across different branches. This way, when viewing a cost free derivation as a size analysis, we obtain the deterministic upper bound. As a consequence we do not have do consider the expected potential as the soundness criterion.

Proof. The proof is conducted via main induction on $\Pi : \sigma \overset{c}{\underset{n}{\vdash}} e \Rightarrow \mu$ and side induction on $\Xi : \Gamma | Q \vdash e : \alpha | Q'$. In the following we only cover the cases involving rules that where adapted or added, the other cases can be easily derived from the original formulation [30].

Case. Π derives $\sigma \overset{0}{\underset{0}{\vdash}} \mathsf{c}\, x_1 \ldots x_n \Rightarrow \{(\mathsf{c}\, v_1 \ldots v_n)^1\}$. Then Ξ consists of a single application of the rule (const):

$$\frac{\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \qquad \Phi(x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid Q) = \Phi(\mathsf{c}\, x_1 \ldots x_n : \alpha \mid Q')}{x_1 : \alpha, \ldots, x_n : \alpha_n | Q \vdash \mathsf{c}\, x_1 \ldots x_n : \alpha | Q'} \ (\mathsf{const})$$

We set $\Gamma := x_1 : \alpha_1, \ldots, x_n : \alpha_n$ and $x_1\sigma = v_1, \ldots, x_n\sigma = v_n$, then $\Phi(\sigma; \Gamma \mid Q) = \Phi(v_1, \ldots, v_n \mid Q)$ and the theorem is obtained from the second premise of the rule.

Case. Π derives

$$\sigma \left|\frac{c}{n+1}\right. \texttt{match}\ x\ \texttt{with}\ \ldots \quad \Rightarrow \mu$$
$$\texttt{|}\ (\mathsf{c}\, x_i\ \ldots\ x_n)\ \texttt{->}\ e$$
$$\ldots$$

This corresponds to an application of the (match) rule:

$$\frac{\Phi(\Gamma, \mathsf{c}_i\, \vec{x_i} \mid Q) = \Phi(\Gamma, \vec{x_i} \mid Q_i) \qquad \Gamma, \vec{x_i} | Q_i \vdash e_i : \beta | Q'}{\Gamma, x : \alpha | Q \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \mathsf{c}_1\, \vec{x_1}\ \texttt{->}\ e_1 \texttt{|}\ \ldots\ \texttt{|}\ \mathsf{c}_m\, \vec{x_m}\ \texttt{->}\ e_m : \beta | Q'} \ (\mathsf{match})$$

WLOG, assume that $x\sigma = \mathsf{c}_i\, v_1\, \ldots, v_n$. From the premises of (match), we have $\Gamma, \vec{x_i} | Q_i \vdash e_i : \alpha | Q'$. From the premise of (match), we obtain

$$\Phi(\sigma; \Gamma, x : \alpha \mid Q) = \Phi(\sigma''; \Gamma, \vec{x_i} \mid Q_i)$$

By MIH we then have $\Phi(\sigma''; \Gamma, \vec{x_i} \mid Q_i) \geq c + \mathbb{E}_\mu(\lambda v.\Phi(v \mid Q'))$, from which the case follows directly.

Case. Consider $e = \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$, that is, Π ends in the following rule:

$$\frac{\sigma \left|\frac{c_1}{n}\right. e_1 \Rightarrow \nu \quad \text{for all } w \in \mathsf{supp}(\nu)\text{: } \sigma[x \mapsto w] \left|\frac{c_w}{n}\right. e_2 \Rightarrow \mu_w}{\sigma \left|\frac{c_1 + \mathbb{E}_\nu(\lambda w.c_w)}{n+1}\right. \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow \sum_{w \in \mathsf{supp}(\nu)} \nu(w) \cdot \mu_w}$$

where $c = c_1 + \mathbb{E}_\nu(\lambda w.c_w)$. In this case Ξ ends either in an application of the (let : base) or the (let) rule.

Due to definition, the potential $\Phi(\sigma; \Gamma, \Delta, \mid Q)$ can be divided into basic potential functions that with variables Γ from the binding expression, variables Δ from the body of the let expression, functions with variables from both Γ and Δ and functions with no variables.

$$\Phi(\sigma; \Gamma, \Delta \mid Q) = \Phi^{nc}(\Gamma \mid Q) + \Phi^{v}(\Delta \mid Q) + \Phi^{\times_\Delta}(\Gamma, \Delta \mid Q) + \Phi(\varnothing \mid Q) \tag{4.1}$$

The potential $\Phi(\sigma; \Delta, x : \alpha \mid R)$ can be split up similarly.

$$\Phi(\sigma; x : \alpha, \Delta \mid R) = \Phi^{nc}(x : \alpha \mid R) + \Phi^{v}(\Delta \mid R) + \Phi^{\times_\Delta}(x : \alpha, \Delta \mid R) + \Phi(\varnothing \mid R) \tag{4.2}$$

By MIH we have

$$\forall w \in \mathsf{supp}(\nu).\ \Phi(\sigma[x \mapsto w]; x : \alpha, \Delta \mid R) \geq c_w + \mathbb{E}_\mu(\lambda v.\Phi(v \mid Q'))\ .$$

We then multiply the respective left- and right-hand sides by the corresponding probability $\nu(w)$ and sum the resulting inequalities, effectively applying $\mathbb{E}_\nu$ to both sides. Additionally we apply linearity of the expectation on the right-hand side, which yields

$$\mathbb{E}_\nu(\lambda w.\Phi(\sigma[x \mapsto w]; x : \alpha, \Delta \mid R)) \geq \mathbb{E}_\nu(\lambda w.c_w) + \mathbb{E}_\nu(\lambda w.\mathbb{E}_{\mu_w}(\lambda v.\Phi(v \mid Q')))\ .$$

So in order to prove the theorem, i.e.

$$\Phi(\sigma;\Gamma,\Delta \mid Q) \geq c_1 + \mathbb{E}_\mu(\lambda w.c_w) + \mathbb{E}_\mu(\lambda w.\mathbb{E}_{\mu_w}(\lambda v.\Phi(v \mid Q'))) \ ,$$

we are left to show

$$\Phi(\sigma;\Gamma,\Delta \mid Q) \geq c_1 + \mathbb{E}_\nu(\lambda w.\Phi(\sigma[x \mapsto w];x\colon\alpha,\Delta \mid R)) \ .$$

Depending on the type of α we distinguish whether (let : base) or (let) was applied.

Case. (let : base) was applied.

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \qquad \Phi^{nc}(\Delta \mid R) = \Phi^{nc}(\Delta \mid Q) \\ \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \\ \Gamma|P \vdash e_1\colon\alpha|P' \qquad \Delta, x\colon\alpha|R \vdash e_2\colon\beta|Q' \qquad \alpha \neq T\end{array}}{\Gamma,\Delta|Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\colon\beta|Q'}\ (\mathsf{let}:\mathsf{base})$$

Using the premises of the rule, we substitute the various terms of Equation 4.1 and arrive at the following equality.

$$\Phi(\sigma;\Gamma,\Delta \mid Q) = \Phi^{nc}(\Gamma \mid P) + \Phi^{v}(\Delta \mid R) + \Phi^{\times_\Delta}(\Gamma,\Delta \mid Q) + \Phi(\varnothing \mid Q)$$

Since α does not bear any potential, we get $\Phi^{nc}(x\colon\alpha \mid R) = \Phi^{nc}(e_1\colon\alpha \mid P') = 0$ and $\Phi^{\times_\Delta}(x\colon\alpha,\Delta \mid R) = 0$. For the remaining terms of Equation 4.2, we again apply the premises of the rule and obtain

$$\Phi(\sigma;x\colon\alpha,\Delta \mid R) = \Phi^{v}(\Delta \mid R) + (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \ .$$

From the MIH we have

$$\begin{array}{rl}\Phi(\sigma;\Gamma \mid P) \geq c_1 + \mathbb{E}_\nu(\lambda w.\Phi(w \mid P')) & \\ \Phi^{nc}(\sigma;\Gamma \mid P) + \Phi(\varnothing \mid P) \geq c_1 + \Phi(\varnothing \mid P') & \alpha \neq T\end{array}$$

and further

$$\begin{array}{rcl}\Phi^{nc}(\sigma;\Gamma \mid P) + \Phi(\varnothing \mid P) & \geq & c_1 + \Phi(\varnothing \mid P') \\ \Phi^{nc}(\sigma;\Gamma \mid P) + \Phi(\varnothing \mid Q) + \Phi(\varnothing \mid P) & \geq & c_1 + \Phi(\varnothing \mid Q) + \Phi(\varnothing \mid P') \\ \Phi^{nc}(\sigma;\Gamma \mid P) + \Phi(\varnothing \mid Q) & \geq & c_1 + \Phi(\varnothing \mid Q) \\ & & - \Phi(\varnothing \mid P) + \Phi(\varnothing \mid P') \\ \Phi^{nc}(\sigma;\Gamma \mid P) + \Phi^{v}(\sigma;\Delta \mid R) & \geq & c_1 + \Phi^{v}(\sigma;\Delta \mid R) + \Phi(\varnothing \mid R) \\ + \Phi(\varnothing \mid Q) & & \\ \Phi^{nc}(\sigma;\Gamma \mid P) + \Phi^{v}(\sigma;\Delta \mid R) & \geq & c_1 \\ + \Phi^{\times_\Delta}(\sigma;\Gamma,\Delta \mid Q) + \Phi(\varnothing \mid Q) & & + \mathbb{E}_v(\lambda w.\Phi^{v}(\sigma[x \mapsto w];\Delta \mid R) \\ & & + \Phi(\varnothing \mid R)) \\ \Phi(\sigma;\Gamma,\Delta \mid Q) & \geq & c_1 \\ & & + \mathbb{E}_\nu(\lambda w.\Phi(\sigma[x \mapsto w]; \\ & & x\colon\alpha,\Delta \mid R))\end{array}$$

Case. (let) was applied.

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \\ \Phi^{v}(\Delta \mid R) = \Phi^{v}(\Delta \mid Q) \qquad \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \\ \Phi^{nc}(x\!:\!\alpha \mid R) = \Phi^{nc}(e_1\!:\!\alpha|P') \qquad \Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x\!:\!\alpha, \Delta \mid R) \\ \Gamma|P \vdash e_1\!:\!\alpha|P' \qquad \Delta, x\!:\!\alpha|R \vdash e_2\!:\!\beta|Q'\end{array}}{\Gamma, \Delta|Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\!:\!\beta|Q'}\ (\mathsf{let})$$

We proceed in the same way as in the previous case with the exception that we now have to consider the terms $\Phi^{nc}(x\!:\!\alpha \mid R)$ and $\Phi^{\times\Delta}(x\!:\!\alpha, \Delta \mid R)$ as well.

$$\Phi(\sigma; x\!:\!\alpha, \Delta \mid R) = \Phi^{nc}(x\!:\!\alpha \mid R) + \Phi^{v}(\Delta \mid R) + \Phi^{\times\Delta}(x\!:\!\alpha, \Delta \mid R) + \Phi(\varnothing \mid R)$$

From the MIH we have

$$\begin{array}{c}\Phi(\sigma; \Gamma \mid P) \geq c_1 + \mathbb{E}_{\nu}(\lambda w.\Phi(w \mid P')) \\ \Phi^{nc}(\sigma; \Gamma \mid P) + \Phi(\varnothing \mid P) \geq c_1 + \mathbb{E}_{\nu}(\lambda w.\Phi^{nc}(w \mid P') + \Phi(\varnothing \mid P'))\end{array}$$

and further

$$\begin{array}{rcl}
\Phi^{nc}(\sigma; \Gamma \mid P) + \Phi(\varnothing \mid P) & \geq & c_1 + \mathbb{E}_{\nu}(\lambda w.\Phi^{nc}(w \mid P') \\
& & \quad + \Phi(\varnothing \mid P')) \\
\Phi^{nc}(\sigma; \Gamma \mid P) + \Phi(\varnothing \mid Q) + \Phi(\varnothing \mid P) & \geq & c_1 + \Phi(\varnothing \mid Q) \\
& & + \mathbb{E}_{\nu}(\lambda w.\Phi^{nc}(w \mid P') \\
& & \quad + \Phi(\varnothing \mid P')) \\
\Phi^{nc}(\sigma; \Gamma \mid P) + \Phi(\varnothing \mid Q) & \geq & c_1 + \Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P) \\
& & + \mathbb{E}_{\nu}(\lambda w.\Phi^{nc}(w \mid P') \\
& & \quad + \Phi(\varnothing \mid P')) \\
\Phi^{nc}(\sigma; \Gamma \mid P) + \Phi^{v}(\sigma; \Delta \mid R) + \Phi(\varnothing \mid Q) & \geq & c_1 + \Phi^{v}(\sigma; \Delta \mid R) \\
& & + \mathbb{E}_{\nu}(\lambda w.\Phi^{nc}(w \mid P') \\
& & \quad + \Phi(\varnothing \mid R)) \\
\Phi^{nc}(\sigma; \Gamma \mid P) + \Phi^{v}(\sigma; \Delta \mid R) + \Phi(\varnothing \mid Q) & \geq & c_1 + \mathbb{E}_{\nu}(\lambda w.\Phi^{v}(\sigma; \Delta \mid R) \\
& & + \Phi^{nc}(w \mid P') + \Phi(\varnothing \mid R))
\end{array}$$

At this point, we instantiate the key inequality

$$\Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x\!:\!\alpha, \Delta \mid R)\ ,$$

derived from the rule's premises, as

$$\Phi^{\times\Delta}(\sigma; \Gamma, \Delta \mid Q) \geq \max_{w \in \mathsf{supp}(\nu)} (\Phi^{\times\Delta}(\sigma[x \mapsto w]; x\!:\!\alpha, \Delta \mid R))\ .$$

From this fact we conclude

$$\begin{aligned}\Phi^{\times\Delta}(\sigma;\Gamma,\Delta \mid Q) &\geq \max_{w\in\mathsf{supp}(\nu)}(\Phi^{\times\Delta}(\sigma[x\mapsto w];x\colon\alpha,\Delta \mid R))\\ &\geq \sum_{w\in\mathsf{supp}(\nu)}\nu(w)\cdot\max_{w\in\mathsf{supp}(\nu)}(\Phi^{\times\Delta}(\sigma[x\mapsto w];x\colon\alpha,\Delta \mid R))\\ &\geq \sum_{w\in\mathsf{supp}(\nu)}\nu(w)\cdot\Phi^{\times\Delta}(\sigma[x\mapsto w];x\colon\alpha,\Delta \mid R)\\ &\geq \mathbb{E}_\nu(\lambda w.\Phi^{\times\Delta}(\sigma[x\mapsto w];x\colon\alpha,\Delta \mid R))\ ,\end{aligned}$$

which we apply to our previous derivation.

$$\begin{aligned}\Phi^{nc}(\sigma;\Gamma \mid P)+\Phi^{v}(\sigma;\Delta \mid R) &\geq c_1+\mathbb{E}_\nu(\lambda w.\Phi^{\times\Delta}(\sigma[x\mapsto w];\\ &\qquad x\colon\alpha,\Delta \mid R))\\ +\ \Phi^{\times\Delta}(\sigma;\Gamma,\Delta \mid Q)+\Phi(\varnothing \mid Q) &\quad +\mathbb{E}_\nu(\lambda w.\Phi^{v}(\Delta \mid R)\\ &\quad +\Phi^{nc}(w \mid P')+\Phi(\varnothing \mid R))\\ \Phi^{nc}(\sigma;\Gamma \mid P)+\Phi^{v}(\sigma;\Delta \mid R) &\geq c_1+\mathbb{E}_\nu(\lambda w.\Phi^{v}(\Delta \mid R)\\ +\ \Phi^{\times\Delta}(\sigma;\Gamma,\Delta \mid Q)+\Phi(\varnothing \mid Q) &\quad +\Phi^{nc}(\sigma[x\mapsto w];x\colon\alpha \mid R)\\ &\quad +\Phi^{\times\Delta}(\sigma[x\mapsto w];x\colon\alpha,\Delta \mid R)\\ &\quad +\Phi(\varnothing \mid R))\\ \Phi(\sigma;\Gamma,\Delta \mid Q) &\geq c_1+\mathbb{E}_\nu(\lambda w.\Phi(\sigma[x\mapsto w];\\ &\qquad x\colon\alpha,\Delta \mid R))\end{aligned}$$

Case. Finally we assume, Ξ ends in the following rule:

$$\frac{\begin{array}{cc}\Gamma|P\vdash^{\mathrm{cf}} e\colon\alpha|P' & \phi \text{ is monotone}\\ \Phi(\Gamma \mid P)=q_\phi\cdot\phi(x)+c & \Phi(\Gamma \mid Q)=q_\phi\cdot\phi(x+k)+c\\ \Phi(e\colon\alpha \mid P')=q_\phi\cdot\phi(y)+c & \Phi(e\colon\alpha \mid Q')=q_\phi\cdot\phi(y+k)+c\end{array}}{\Gamma|Q\vdash^{\mathrm{cf}} e\colon\alpha|Q'}\ (\mathsf{shiftm})$$

Note that, given that the rule only types a cost free judgment, we only need to proof the corresponding weaker part of theorem. By main MIH and the rule obligations we have

$$\begin{aligned}\Phi(\sigma;\Gamma \mid P)&\geq\Phi(v \mid P')\\ q_\phi\cdot\phi(x\sigma)+c&\geq q_\phi\cdot\phi(v)+c\\ \phi(x\sigma)&\geq\phi(v)\\ \phi(x\sigma+k)&\geq\phi(v+k) && \text{monotonicity of } \phi\\ q_\phi\cdot\phi(x\sigma+k)+c&\geq q_\phi\cdot\phi(v+k)+c\\ \Phi(\sigma;\Gamma \mid Q)&\geq\Phi(v \mid Q')\end{aligned}\ .$$

□

4.2 Potential Instances

With the definition of the abstract type system, we have established a flexible analysis framework. In this section, we demonstrate how to instantiate this framework with a specific potential function to obtain a complete analysis. Since the abstract type rules allow for one potential-bearing type, our goal is to specify the requirements for the potential function associated with this type. To formalize this concept, we introduce the notion of a *potential instance*, as defined below.

Definition 4.3. A potential instance is defined as the quadruple $(T, \Phi, \Sigma_{\leq}, C)$ where T is the data type on which the template potential function Φ operates. $\Sigma_{\leq}$ is a formal theory that specifies linear inequalities, denoted by $\leq$, over basic potential functions of Φ. Additionally C is a set of instance specific constraints, that implies the following abstract constraints over Φ.

For every constructor c of type T

$$\Phi(x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid Q) = \Phi(\mathsf{c}\, x_1, \ldots, x_n \mid P)$$
$$\Phi(\Gamma, \mathsf{c}\, x_1, \ldots, x_n \mid Q) = \Phi(\Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid P)$$

and

$$\exists J, \vec{P}, \vec{P'}.\ \forall i \in J.\ \Gamma | P_i \vdash^{\mathrm{cf}} e_1 : \alpha | P'_i \Rightarrow \Phi^{\times_\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times_\Delta}(x : \alpha, \Delta \mid R)$$

$\Sigma_{\leq}$ describes the specialized expert knowledge required for the application of the (w) rule. Based on these facts, we can apply the linearization technique described in Chapter 3 and derive an expert knowledge matrix A for a specific instance of Φ. In A, each row represents an inequality over the basic potential functions of Φ, derived from $\Sigma_{\leq}$. Specifically, every row $(a_1, \ldots, a_n)$ of A corresponds to the inequality $\sum_{1 \leq i \leq n} a_i \phi_i \leq 0$. Note that the indices of the basic potential functions are not actually natural numbers, as we already established in Chapter 3, but we can assume that they can be countably ordered.

The potential instance constraints C, are formulated in terms of potential function coefficients to guarantee the obligations of the (const) and (match) rules, as well as the abstract size constraint of the (let) rule. Note that this is only a subset of the obligations in the type system. The other requirements are potential agnostic. The following lemmas express these potential agnostic constraints, in terms of the individual coefficients of the corresponding resource annotations.

To begin with need to guarantee the equalities required by the $(\mathsf{app} : \mathsf{base})$ and (var) rules. Since the potential does not change in these instances, we only require the individual coefficients to be equal. Note that the annotations might contain different variables, which correspond to the same potential. In this case we apply a substitution to rename those variables.

Lemma 4.4. *Let* $\sigma : \mathsf{vars}(\Gamma) \to \mathsf{vars}(\Delta)$ *be a bijective substitution,* $S(\vec{x}) = \{\sigma(x)^a \mid x^a \in \vec{x}\}$, $x \in \mathsf{vars}(\Gamma)$, $\vec{x} \in I(\mathsf{vars}(\Gamma))$, $c \in \mathbb{N}$ *and* Q, P *resource annotations. If*

$$q_{(x)} = p_{(\sigma(x))} \qquad\qquad q_{(\vec{x},c)} = p_{(S(\vec{x}),c)}$$

then

$$\Phi(\Gamma \mid Q) = \Phi(\Delta \mid P)$$

Proof.

$$\begin{aligned}\Phi(\Gamma \mid Q) &= \sum_{x \in \mathsf{vars}(\Gamma)} q_{(x)} \cdot \phi_{(x)} + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), c \in \mathbb{N}} q_{(\vec{x},c)} \cdot \phi_{(\vec{x},c)} \\ &= \sum_{y \in \mathsf{vars}(\Delta)} p_{(y)} \cdot \phi_{(y)} + \sum_{\vec{y} \in I(\mathsf{vars}(\Delta)), c \in \mathbb{N}} p_{(\vec{y},c)} \cdot \phi_{(\vec{y},c)} \\ &= \Phi(\Delta \mid P)\end{aligned}$$

□

The following lemma provides the constraints necessary to invoke the (let : base) rule.

Lemma 4.5. *Let* $x \in \mathsf{vars}(\Gamma)$, $\vec{x} \in I(\mathsf{vars}(\Gamma))$, $\vec{x} \neq \emptyset$,$y \in \mathsf{vars}(\Delta)$, $\vec{y} \in I(\mathsf{vars}(\Delta))$, $\vec{y} \neq \emptyset$, $c \in \mathbb{N}$ *and* Q, P, R *resource annotations. If*

$$p_{(x)} = q_{(x)} \qquad p_{(\vec{x},c)} = q_{(\vec{x},c)} \qquad r_{(y)} = q_{(y)} \qquad r_{(\vec{y},c)} = q_{(\vec{y},c)} \qquad r_{(c)} = q_{(c)} - p_{(c)} + p'_{(c)}$$

then

$$\begin{aligned}&\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \\ &\Phi^{nc}(\Delta \mid R) = \Phi^{nc}(\Delta \mid Q) \\ &\Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P')\end{aligned}$$

Proof. The equations are easily proved by unfolding the definitions and applying the constraints.

$$\begin{aligned}\Phi^{nc}(\Gamma \mid Q) &= \sum_{x \in \mathsf{vars}(\Gamma)} q_{(x)} \cdot \phi_{(x)} + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), \vec{x} \neq \emptyset, c \in \mathbb{N}} q_{(\vec{x},c)} \cdot \phi_{(\vec{x},c)} \\ &= \sum_{x \in \mathsf{vars}(\Gamma)} p_{(x)} \cdot \phi_{(x)} + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), \vec{x} \neq \emptyset c \in \mathbb{N}} p_{(\vec{x},c)} \cdot \phi_{(\vec{x},c)} \\ &= \Phi^{nc}(\Gamma \mid P)\end{aligned}$$

$$\begin{aligned}\Phi^{nc}(\Delta \mid R) &= \sum_{y \in \mathsf{vars}(\Delta)} r_{(y)} \cdot \phi_{(y)} + \sum_{\vec{y} \in I(\mathsf{vars}(\Delta)), \vec{y} \neq \emptyset, c \in \mathbb{N}} r_{(\vec{y},c)} \cdot \phi_{(\vec{y},c)} \\ &= \sum_{y \in \mathsf{vars}(\Delta)} q_{(y)} \cdot \phi_{(y)} + \sum_{\vec{y} \in I(\mathsf{vars}(\Delta)), \vec{y} \neq \emptyset, c \in \mathbb{N}} q_{(\vec{y},c)} \cdot \phi_{(\vec{y},c)} \\ &= \Phi^{nc}(\Delta \mid Q)\end{aligned}$$

$$\begin{aligned}\Phi(\varnothing \mid R) &= \sum_{c\in\mathbb{N}} r_{(c)} \cdot \phi_{(c)} \\ &= \sum_{c\in\mathbb{N}} \left((q_{(c)} - p_{(c)}) + p'_{(c)}\right) \phi_{(c)} \\ &= \left(\sum_{c\in\mathbb{N}} q_{(c)} \cdot \phi_{(c)} - \sum_{c\in\mathbb{N}} p_{(c)} \cdot \phi_{(c)}\right) + \sum_{c\in\mathbb{N}} p'_{(c)} \cdot \phi_{(c)} \\ &= (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P')\end{aligned}$$

□

The constraint $\Phi^{nc}(x\colon\alpha \mid R) = \Phi^{nc}(e_1\colon\alpha|P')$, found in (let), is an equality constraint that possible requires renaming of variables and is covered by Lemma 4.4. Finally the only potential-agnostic abstract constraint, not yet addressed, is realized by the following lemma.

Lemma 4.6. *Let $\vec{y} \in I(\mathsf{vars}(\Delta))$, x be a variable, and Q, P, R resource annotations. If $p_{(\vec{y})} = q_{(\vec{y})}$ then $\Phi^v(\Delta \mid R) = \Phi^v(\Delta \mid Q)$.*

Proof. The theorem is easily verified by applying the definition of $\Phi^v(.)$ □

Given our new definition of potential instances, in the following sections we investigate examples of different template potential functions and demonstrate how they can serve as potential instances for our abstract analysis. The first two examples, logarithmic and polynomial potentials, have already been utilized for automated amortized analysis in previous literature and are only adapted for our framework, whereas the final example, log-linear potential, is introduced for the first time in this work.

4.2.1 Logarithmic Potential for Trees

Logarithmic potential functions where motivated by the analysis of splay trees, which have logarithmic amortized costs. Additionally the pen and paper proof of the amortized cost bound, required potential that reflects the shape of the search tree [42]. For this reason Hofmann et al. [19] employ a variant of Schoenmakers' potential, $rk(t)$ for a tree t, cf. [34, 36], which is inductively defined by

$$\begin{aligned}\mathsf{rk}(\texttt{leaf}) &:= 1 \\ \mathsf{rk}(\texttt{node}\ l\ d\ r) &:= \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)\end{aligned}$$

. Here the size $|t|$ of a tree t, denotes the number of leaves. As mentioned before, an important characteristic in the context of a syntax directed analysis is the closure under basic operations of the programming language, more specifically constructors and deconstructors. Since rk, as defined above, is not closed under node construction, the following additional basic potential functions are introduced to cover the logarithmic terms [19].

$$\phi_{(a,b)}(t) = \log_2(a|t| + b)$$

Here, in principle, we have $a, b \in \mathbb{N}$, but for the actual analysis to become feasible, we obviously can only consider a finite subset of the possible basic potential functions. Leutgeb et al. [29] found restricting $a \in \{0, 1\}$ and $b \in \{0, 1, 2\}$ to be sufficient for the analysis of their chosen examples. Our prototype implementation presented in Chapter 6, follows the same heuristic. With this basic template shape, it is also possible to express the relationship for trees s and t with $|t| = |s| + 1$ as $\phi_{(a,b)}(t) = \phi_{(a,b+a)}(s)$. This property is crucial for the analysis, as it ensures the potential can be extended when a node is added to a tree. Additionally the function allows to represent constants by setting $a = 0$ and $b = 2$. For the case of multiple tree arguments the basic definition is extended to

$$\phi_{(a_1,\ldots,a_n,b)}(t_1, \ldots, t_n) = \log_2(a_1 \cdot |t_1| + \cdots + a_n \cdot |t_n| + b) \ .$$

As we recall, a potential function is a linear combination of basic potential functions, which leads to the following definition of the multivariate logarithmic potential.

Definition 4.7. Let Q be a resource annotation. The multivariate logarithmic potential function Φ_{log} is defined as follows.

$$\Phi_{\mathsf{log}}(t_1, \ldots, t_n \mid Q) = \sum_{i=1}^{n} q_i \cdot \mathsf{rk}(t_i) + \sum_{a_1,\ldots,a_n,b \in \mathbb{N}} q_{(t_1^{a_1},\ldots,t_n^{a_n},b)} \cdot \log_2(a_1 \cdot |t_1| + \cdots + a_n \cdot |t_n| + b)$$

For the logarithmic potential Lorenz et al. [29] have demonstrated the necessity of incorporating the monotonicity of the logarithm and Lemma 4.8 as expert knowledge for the weakening rule. The latter expresses an identity for logarithms that allows the potential of a tree to be split for its two sub trees and a constant remainder potential.

Lemma 4.8. *Let* $x, y \geq 1$. *Then* $2 + \log_2(x) + \log_2(y) \leq 2\log_2(x + y)$

In the remainder of this section, we show how the potential function introduced in [19] can be reformulated as a potential instance within our abstract analysis. To this end, we define the corresponding constraints and then proceed to prove the potential instance axioms. Note that in the following definition and in the rest of the chapter, we assume that coefficients not mentioned explicitly in constraints are set to zero.

Definition 4.9. The potential specific constraints C_{log} are defined as follows.

Constructors :

`leaf`. Let $(Q + K)$ and Q' be resource annotations, where $\mathsf{vars}(Q + K) = \emptyset$ and $\mathsf{vars}(Q') = \{y\}$.

$$\forall c \geqslant 2 \ q_{(c)} = \sum_{a+b=c} q'_{(y^a,b)} \qquad\qquad K = q'_{(y)}$$

$q'_{(y^1)}$ is left unconstrained.

`node` $x_1\ x_2\ x_3$: Let Q and Q' be resource annotations with $\mathsf{vars}(Q) = \{x_1, x_3\}$ and $\mathsf{vars}(Q') = \{y\}$.

$$q_{(x_1)} = q_{(x_3)} = q'_{(y)} \qquad q_{(x_1^1)} = q_{(x_3^1)} = q'_{(y)} \qquad q_{(x_1^a, x_3^a, c)} = q'_{(y^a, c)}$$

Deconstructors :

`leaf`. Let Q and $(P + K)$ be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X$, with x being the deconstructed variable and $\vec{x} \in I(X)$.

$$p_{(\vec{x},c)} = \sum_{a+b=c} q_{(\vec{x},x^a,b)} \qquad p_{(x_i)} = q_{(x_i)} \qquad K = q_{(x)}$$

`node` $x_1\ x_2\ x_3$: Let Q and P be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X \cup \{x_1, x_3\}$, with x being the deconstructed variable and $\vec{x} = I(X)$.

$$p_{(x_1)} = p_{(x_3)} = q_{(x)} \qquad p_{(x_i)} = q_{(x_i)} \qquad p_{(x_1^1)} = p_{(x_3^1)} = q_{(x)}$$

$$p_{(\vec{x},x_1^a,x_3^a,c)} = q_{(\vec{x},x^a,c)}$$

Cost Free Annotations Let Q, R be resource annotations and x, y variables. Then let J be the following index set.

$$J = \{(\vec{y}, x^d, e) \mid \vec{y} \in I(\mathsf{vars}(R)), \vec{y} \neq \vec{0}, d + \max\{e, 0\} > 0\}$$

Let $\vec{x} \in I(X)$. We then define constraints over the sets of resource annotations $\vec{P}$, $\vec{P'}$ indexed by elements of J.

$$\forall \vec{y} \neq \vec{0}, \vec{x} \neq \vec{0} \vee c \neq 0 \left(q_{(\vec{x},\vec{y},c)} = \sum_{d,e} p^{(\vec{y},x^d,e)}_{(\vec{x},c+\max\{-e,0\})} \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(\sum_{(\vec{x},c)} p^{(\vec{y},x^d,e)}_{(\vec{x},c)} \geq p'^{(\vec{y},x^d,e)}_{(d,\max\{e,0\})} \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(\forall \vec{x} \neq \vec{0} \vee c \neq 0 \left(p^{(\vec{y},x^d,e)}_{(\vec{x},c)} \neq 0 \rightarrow p'^{(\vec{y},x^d,e)}_{(y^d,e)} \leq p^{(\vec{y},x^d,e)}_{(\vec{x},c)} \right) \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(\forall (d', e') \neq (d, e) \left(p'^{(\vec{y},x^d,e)}_{(d',e')} = 0 \right) \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(r_{(\vec{y},x^d,e)} = p'^{(\vec{y},x^d,e)}_{(y^d,e)} \right)$$

In contrast to the original formulation, for the `leaf` constructor constraints we add the condition that $q'_{(y^1)}$ is left unconstrained. In fact ATLAS contains the same insight, but

does not explicitly mention it in its formal presentation. Allowing the constraint solver to assign arbitrary potential to this coefficient, might seem problematic, but is sound and even crucial to the analysis. First note that $q'_{(y^1)}$ corresponds to the basic potential function $\log_2(|leaf|) = \log_2(1) = 0$, therefore no matter which value is assigned to $q'_{(y^1)}$, the potential remains zero. Next, to understand why we still have to include this zero potential, consider the following nested constructor expression, which presents a common occurrence in our benchmarks.

```
1   node t a leaf
```

We assume that the potential for `leaf` is constructed according to the corresponding constraints, and bound to variable x. In order to fulfill the constructor constraints for `node`, we have to guarantee the following equation, where the variable y represents the constructed tree.

$$q'_{(y)} = q_{(t)} = q_{(t^1)} = q_{(x^1)} = q_{(x)}$$

Assuming we do set $q_{(x^1)} = 0$, i.e. disregarding the zero potential, we would then be forced set $q'_{(y)} = 0$, as well, which would inhibit us of having potential for trees of this shape.

Before proceeding with the proof of the previous constraints, we introduce an additional lemma that plays a crucial role in the proof, particularly for the cost-free typing constraints.

Lemma 4.10. *Assume* $\sum_i q_i \log_2 a_i \geqslant q \log_2 b$ *for some rational numbers* $a_i, b > 0$ *and* $q_i \geqslant q$. *Then,* $\sum_i q_i \log_2(a_i + c) \geqslant q \log_2(b + c)$ *for all* $c \geqslant 1$.

For a proof of lemma 4.10 we refer to [19].

Lemma 4.11. $(\mathsf{T}, \Phi_{\mathsf{log}}, \Sigma_{\leq_{\mathsf{log}}}, C_{\mathsf{log}})$ *is a potential instance.*

Proof. To begin with, we verify that the constructor axioms are fulfilled.

Case. $\mathsf{c}\, x_1 \dots x_n = \texttt{leaf}$:

$$\begin{aligned}
\Phi(\varnothing \mid Q + K) &= K + \sum_c q_{(c)} \cdot \log_2(c) \\
&= K + \sum_{c \geq 2} q(c) \cdot \log_2(c) \\
&= q'_{(y)} + \sum_{a+b \geq 2} q'_{(y^a,b)} \cdot \log_2(a+b) + q'_{(y^1)} \cdot \log_2(1) \\
&= q'_{(y)}\, \mathsf{rk}(\texttt{leaf}) + \sum_{a+b \geq 2} q'_{(y^a,b)} \cdot \log_2(a+b) \\
&\quad + q'_{(y^1)} \cdot \log_2(|\texttt{leaf}|) \\
&= q'_{(y)}\, \mathsf{rk}(\texttt{leaf}) + \sum_{a,b} q'_{(y^a,b)} \phi_{(y^a,b)} \\
&= \Phi(\texttt{leaf} \mid Q')
\end{aligned}$$

Case. $\mathsf{c}\, x_1 \ldots x_n = \texttt{node}\ x_1\ x_2\ x_3$:

$$
\begin{aligned}
&\Phi(x_1 : \mathsf{T}, x_2 : \mathsf{B}, x_3 : \mathsf{T} \mid Q) \\
&= q_1 \cdot \mathsf{rk}(x_1) + q_2 \cdot \mathsf{rk}(x_3) \\
&\quad + \sum_{a_1,a_2,b} q_{(x_1^{a_1}, x_3^{a_2}, b)} \cdot \log_2(a_1 \cdot |x_1| + a_2 \cdot |x_3| + b) \\
&\geq q'_{(y)} \cdot \mathsf{rk}(x_1) + q'_{(y)} \cdot \mathsf{rk}(x_3) + q_{(x_1^1)} \cdot \log_2(|x_1|) + q_{(x_3^1)} \cdot \log_2(|x_3|) \\
&\quad + \sum_{a,b} q_{(x_1^a, x_3^a, b)} \cdot \log_2(a \cdot |x_1| + a \cdot |x_3| + b) \\
&\geq q'_{(y)} \cdot (\mathsf{rk}(x_1) + \mathsf{rk}(x_1) + \log_2(|x_1|) + \log_2(|x_3|)) \\
&\quad + \sum_{a,b} q'_{(y^a,b)} \cdot \log_2(a \cdot (|x_1| + |x_3|) + b) \\
&= q'_y \cdot \mathsf{rk}(\texttt{node}\ x_1\ x_2\ x_3) + \sum_{a,b} q'_{(y^a,b)} \cdot \phi_{(y^a,b)}(\texttt{node}\ x_1\ x_2\ x_3) \\
&= \Phi(\texttt{node}\ x_1\ x_2\ x_3 \mid Q')
\end{aligned}
$$

Next we verify the deconstructor axioms.

Case. $\mathsf{c}\, x_1 \ldots x_n = \texttt{leaf}$:

$$
\begin{aligned}
\Phi(\Gamma, x : T \mid Q) &= \sum_i q_{(x_i)}\, \mathsf{rk}(x_i) + q_{(x)}\, \mathsf{rk}(\texttt{leaf}) \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), a, b} q_{(\vec{x}, x^a, c)} \log_2(\vec{a}|x| + a|\texttt{leaf}| + b) \\
&= \sum_i q_{(x_i)}\, \mathsf{rk}(x_i) + q_{(x)}\, \mathsf{rk}(\texttt{leaf}) \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), a, b} q_{(\vec{x}, x^a, c)} \log_2(\vec{a}|\vec{x}| + a + b) \\
&= \sum_i p_{(x_i)}\, \mathsf{rk}(x_i) + q_{(x)} \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), c} \Big(\sum_{a+b=c} p_{(\vec{x}, c)} \Big) \log_2(\vec{a}|\vec{x}| + c) \\
&= \Phi(\Gamma \mid P) + q_{(x)} = \Phi(\Gamma \mid P + K)
\end{aligned}
$$

Case. $\mathsf{c}\, x_1 \ldots x_n = \texttt{node}\ x_1\ x_2\ x_3$:

$$\begin{aligned}
\Phi(\Gamma, x\colon T \mid Q) &= \sum_i q_{(x_i)}\, \mathsf{rk}(x_i) + q_{(x)}\, \mathsf{rk}(\texttt{node}\ x_1\ x_2\ x_3) \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), a, c} q_{(\vec{x}, x^a, c)} \log_2(\vec{a}|\vec{x}| + a|\texttt{node}\ x_1\ x_2\ x_3| + c) \\
&= \sum_i q_{(x_i)}\, \mathsf{rk}(x_i) + q_{(x)}(\mathsf{rk}(x_1) + \log_2(|x_1|) + \log_2(|x_3|) + \mathsf{rk}(x_3)) \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), a, c} q_{(\vec{x}, x^a, c)} \log_2(\vec{a}|\vec{x}| + a(|x_1| + |x_3|) + c) \\
&= \sum_i p_{(x_i)}\, \mathsf{rk}(x_i) + p_{(x_1)}\, \mathsf{rk}(x_1) + p_{(x_1^1)} \log_2(|x_1|) \\
&\quad + p_{(x_3^1)} \log_2(|x_3|) + p_{(x_3)}\, \mathsf{rk}(x_3) \\
&\quad + \sum_{\vec{x} \in I(\mathsf{vars}(\Gamma)), a, c} p_{(\vec{x}, x_1^a, x_3^a, c)} \log_2(\vec{a}|\vec{x}| + a|x_1| + a|x_3| + c) \\
&= \Phi(\Gamma, x_1\colon \mathsf{T}, a\colon \mathsf{B}, x_3\colon \mathsf{T} \mid R)
\end{aligned}$$

Lastly we show

$$\exists J, \vec{P}, \vec{P'}.\ \forall i \in J.\ \Gamma | P_i \vdash^{\mathrm{cf}} e_1\colon \alpha | P_i' \Rightarrow \Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x\colon \alpha, \Delta \mid R)\ .$$

Let Q and R be resource annotations and Γ, Δ typing contexts, where $\vec{x} \in I(\mathsf{vars}(\Gamma))$ and $\vec{y} \in I(\mathsf{vars}(\Delta))$. Also let x be a variable. Assume we have $\vec{P}$ and $\vec{P'}$, indexed by the set J, defined as specified in the potential instance constraints. We then assume

$$\forall i \in J.\ \Gamma | P_i \vdash^{\mathrm{cf}} e_1\colon \alpha | P_i'$$

From the soundness theorem we then obtain obtain the following inequalities.

$$\forall i \in J.\ \Phi(\Gamma \mid P_i) \geq \Phi(e_1\colon \alpha \mid P_i')$$

By unfolding the definitions we have

$$\sum_{\vec{x}, c} p_{(\vec{x}, c)}^{(\vec{y}, x^d, e)} \log_2(\vec{a}|\vec{x}| + c) \geq p'^{(\vec{y}, x^d, e)}_{(d, \max\{e, 0\})} \log_2(d|x| + \max\{e, 0\})$$

The instance provides the following constraints.

$$\forall (\vec{y}, x^d, e) \in J \left(\sum_{(\vec{x}, c)} p_{(\vec{x}, c)}^{(\vec{y}, x^d, e)} \geq p'^{(\vec{y}, x^d, e)}_{(d, \max\{e, 0\})} \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(\forall \vec{x} \neq \vec{0} \vee c \neq 0 \left(p_{(\vec{x}, c)}^{(\vec{y}, x^d, e)} \neq 0 \rightarrow p'^{(\vec{y}, x^d, e)}_{(y^d, e)} \leq p_{(\vec{x}, c)}^{(\vec{y}, x^d, e)} \right) \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(\forall (d', e') \neq (d, e) \left(p'^{(\vec{y}, x^d, e)}_{(p', e')} = 0 \right) \right)$$

Using these as premises, we can instantiate lemma 4.10 with the previous inequality and obtain

$$\sum_{\vec{x},c} p^{(\vec{y},x^d,e)}_{(\vec{x},c)} \log_2(\vec{a}|\vec{x}| + \vec{b}|\vec{y}| + c - \max\{-e,0\}) \geq p'^{((\vec{y},x^d,e))}_{(d,\max\{e,0\})} \log_2(\vec{b}|\vec{y}| + d|x| + e)$$

In order to understand why we can subtract $\max\{-e,0\}$ on the left-hand-side and move from $\max\{e,0\}$ to e on the right-hand-side, we distinguish two cases for e. If $e \geq 0$ then $\max\{-e,0\} = 0$. In this case we instantiate c from lemma 4.10 with $\vec{b}|\vec{y}|$. On the other hand, if $e < 0$ then $\max\{-e,0\} = e$, then we instantiate c with $\vec{b}|\vec{y}| + e$.

We can now continue to sum up the inequalities for all $(\vec{y}, x^d, e) \in J$ and using the remaining instance constraints

$$\forall \vec{y} \neq \vec{0}, \vec{x} \neq \vec{0} \vee c \neq 0 \left(q_{(\vec{x},\vec{y},c)} = \sum_{d,e} p^{(\vec{y},x^d,e)}_{(\vec{x},c+\max\{-e,0\})} \right)$$

$$\forall (\vec{y}, x^d, e) \in J \left(r_{(\vec{y},x^d,e)} = p'^{(\vec{y},x^d,e)}_{y^d,e} \right)$$

we finally obtain

$$\sum_{\vec{x}\neq\vec{0},\vec{y}\neq\vec{0},c\neq 0} q_{(\vec{x},\vec{y},c)} \log_2(\vec{a}|\vec{x}| + \vec{b}|\vec{y}| + c) \geq \sum_{(\vec{y},x^d,e)\in J} r_{(\vec{y},x^d,e))} \log_2(\vec{b}|\vec{y}| + d|x| + e)$$

which by definition is exactly

$$\Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x\colon\alpha, \Delta \mid R)$$

□

4.2.2 Polynomial Potential for Lists

The next potential function, we present, is a variation of the multivariate polynomial potential devised by Hoffmann et al. [9]. The formulation given here is a simplification of the former, in the sense that we only consider lists with basic elements, i.e. of type B. In the original version an elaborate indexing scheme is used to allow the nesting of trees and lists. Since we do not need to support these cases, we instead employ a simpler version that can be represented with our common indexing scheme introduced in Chapter 3.

We start this section by recapitulating the basic concepts of polynomial potential, as presented in previous works [9], while adapting them to our simplified setting.

The basic potential functions $\mathcal{BF}(\alpha)$ are defined as follows:

$$\mathcal{BF}(\mathsf{B}) = \{v \mapsto 1\}$$
$$\mathcal{BF}(\mathsf{L}) = \{v \mapsto \binom{|v|}{k} \mid k \in \mathbb{N}\}$$
$$\mathcal{BF}(\alpha_1, \ldots, \alpha_n) = \sum\{(a_1, \ldots, a_n) \mapsto \prod_{i=1}^{n} \phi_i(a_i) \mid \phi_i \in \mathcal{BF}(\alpha_i)\}$$

Note that this definition is closed under list construction, i.e $\phi_k(x : \ell) = \binom{|\ell+1|}{k} \in \mathcal{BF}(\mathsf{L})$. In this approach coefficients and basic potential functions are then linked with an index set defined as

$$I_k(x_1, \ldots, x_n) = \{(i_1, \ldots, i_n) \mid \sum_{j=1}^{n} i_j \leq k\}$$

This set is used to restrict the basic potential functions in the linear combination and thereby give the potential function a maximum polynomial degree of k.

$$\Phi(x_1, \ldots, x_n \mid Q) = \sum_{(i_1,\ldots,i_n) \in I_k(x_1,\ldots,x_n)} q_{\vec{i}} \prod_{j=1}^{n} \phi_{i_j}(x_j)$$

Example 4.12. For a pair of integer lists ℓ_1, ℓ_2 with lengths m, n, the potential function for maximum degree k is given by

$$\Phi(l_1, l_2 \mid Q) = \sum_{0 \leq i+j \leq k} q_{(i,j)} \cdot \binom{|l_1|}{i} \binom{|l_2|}{j}$$

One can easily translate this restricted formulation, to our definition of resource annotations. The basic potential functions are then defined as follows.

$$\phi_{(l_1^{a_1},\ldots,l_n^{a_n})}(l_1, \ldots, l_n) = \prod_{i=1}^{n} \binom{|l_i|}{a_i}$$

The potential function Φ is then defined as the usual linear combination with coefficients from Q.

Definition 4.13. Let Q be a resource annotation. The multivariate polynomial potential function of degree k Φ_{n^k} is defined as follows.

$$\Phi_{\mathsf{n}^k}(l_1, \ldots, l_n \mid Q) = \sum_{i=0}^{k} \sum_{a_1+\cdots+a_n=i} q_{(l_1^{a_1},\ldots,l_n^{a_n})} \cdot \prod_{j=1}^{n} \binom{|l_j|}{a_j}$$

Just like the original formulation we define an additive shift to handle the `cons` constructor.

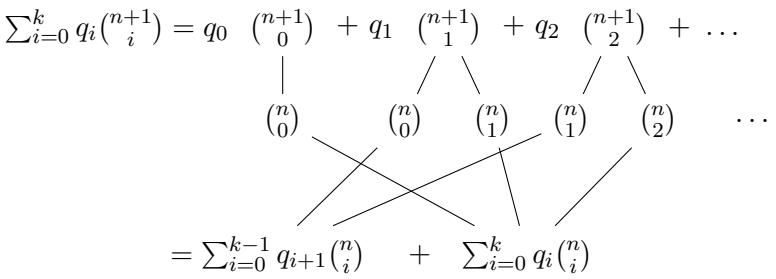
Definition 4.14. Let Q and Q' be resource annotations and k be the polynomial degree. The additive shift $Q' = \triangleright(Q)$ is then defined by

$$q'_{(\vec{l},xs^a)} = \begin{cases} q_{(\vec{l},l^a)} + q_{(\vec{l},l^{(a+1)})} & a < k \\ q_{(\vec{l},l^a)} & a = k \end{cases}$$

The mathematical foundation of this shift operation is the following identity for binomial coefficients.

$$\sum_{i=0}^{k} q_i \binom{n+1}{i} = \sum_{i=0}^{k-1} q_{i+1} \binom{n}{i} + \sum_{i=0}^{k} q_i \binom{n}{i} \tag{4.3}$$

It is based on pascal's triangle $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$ and its validity is illustrated in the following diagram.



Next we demonstrate that this identity can be extended for our multivariate basic potential functions. Lemma 4.15 allows us to read of the definition of the additive shift.

Lemma 4.15. *Let $q_* \in \mathbb{Q}_0^+, k \in \mathbb{N}$. Then, for a sum involving multinomial coefficients, the following identity holds:*

$$\sum_{i=0}^{k} \sum_{k_1+\dots+k_p+j=i} q_{(k_1,\dots,k_p,j)} \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m+1}{j} =$$
$$\sum_{i=0}^{k-1} \sum_{k_1+\dots+k_p+j=i} (q_{(k_1,\dots,k_p,j)} + q_{(k_1,\dots,k_p,j+1)}) \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m}{j}$$
$$+ \sum_{k_1+\dots+k_p+j=k} q_{(k_1,\dots,k_p,j)} \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m}{k}$$

Proof.

$$
\begin{aligned}
&\sum_{i=0}^{k} \sum_{k_1+\cdots+k_p+j=i} q_{(k_1,\ldots,k_p,j)} \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m+1}{j} \\
&= \sum_{i=0}^{k} \sum_{j=0}^{k-i} q_{(k_1,\ldots,k_p,j)} \prod_{k_1+\cdots+k_p=i} \binom{n_p}{k_p} \binom{m+1}{j} \\
&= \sum_{i=0}^{k} \prod_{k_1+\cdots+k_p=i} \binom{n_p}{k_p} \sum_{j=0}^{k-i} q_{(k_1,\ldots,k_p,j)} \binom{m+1}{j} \\
&= \sum_{i=0}^{k} \prod_{k_1+\cdots+k_p=i} \binom{n_p}{k_p} \\
&\quad \left(\sum_{j=0}^{k-i-1} (q_{(k_1,\ldots,k_p,j)} + q_{(k_1,\ldots,k_p,j+1)}) \binom{m}{j} + q_{(k_1,\ldots,k_p,j)} \binom{m}{k-i} \right) \qquad \text{Equation 4.3} \\
&= \sum_{i=0}^{k-1} \sum_{k_1+\cdots+k_p+j=i} (q_{(k_1,\ldots,k_p,j)} + q_{(k_1,\ldots,k_p,j+1)}) \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m}{j} \\
&\quad + \sum_{k_1+\cdots+k_p+j=k} q_{(k_1,\ldots,k_p,j)} \binom{n_1}{k_1} \cdots \binom{n_p}{k_p} \binom{m}{k}
\end{aligned}
$$

□

For the polynomial potential, we do not require any expert knowledge. In the original formulation, it is shown that it suffices to compare the coefficients of resource annotations, for further details we refer the reader to [9]. The weakening in such an analysis, can often be resolved at the leaves of the derivation, i.e. through the (app) rule and a suitable cost free typing.

In the following we present the potential specific constraints for the polynomial potential and proceed with a proof of the potential axioms.

Definition 4.16. The potential specific constraints C_{nk} are defined as follows.

Constructors :

`nil`. Let Q and Q' be resource annotations, where $\mathsf{vars}(Q) = \emptyset$ and $\mathsf{vars}(Q') = \{y\}$.

$$q_{()} = q'_{()}$$

`cons` x xs: Let Q and Q' be resource annotations with $\mathsf{vars}(Q) = \{x, xs\}$ and $\mathsf{vars}(Q') = \{y\}$.

$$Q = \triangleright(Q')$$

Deconstructors :

`nil`. Let Q and $(P + K)$ be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X$, with x being the deconstructed variable and $\vec{x} \in I(X)$.

$$p_{()} = q_{()} \qquad\qquad p_{(\vec{x})} = q_{(\vec{x})}$$

`cons` x xs: Let Q and P be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X \cup \{xs\}$, with x being the deconstructed variable and $\vec{x} = I(X)$.

$$P = \triangleright(Q)$$

Cost Free Annotations Let Q, R be resource annotations and x, y variables. Then let I be the following index set.

$$J = \{(\vec{y}) \mid \vec{y} \in I(\mathsf{vars}(R)), \vec{y} \neq \vec{0}\}$$

We then define constraints over the sets of resource annotations $\vec{P}$, $\vec{P'}$ indexed by elements of I.

$$p^{(\vec{y})}_{(\vec{x})} = q_{(\vec{x},\vec{y})} \qquad\qquad r_{(\vec{y},x^a)} = p'^{(\vec{y})}_{(x^a)}$$

Lemma 4.17. $(\mathsf{L}, \Phi_{\mathsf{n}^k}, \Sigma_{\leq_{\mathsf{n}^k}}, C_{\mathsf{n}^k})$ *is a potential instance.*

Proof. First we verify that the constructor axioms hold. For this we distinguish between the two possible constructors for the list type.

Case. $\mathsf{c}\, x_1 \dots x_n =$ `nil`:

$$\begin{array}{rcl} \Phi(\varnothing \mid Q) & = & \sum_{i=0}^{k} \sum_{0=k} q_{()} \cdot 1 \\ & = & q_{()} = q'_{()} \\ & = & \Phi(\texttt{nil} \mid Q') \end{array}$$

Case. $\mathsf{c}\, x_1 \dots x_n =$ `cons` x xs:

$$\begin{array}{ll} & \Phi(x\colon \mathsf{B}, xs\colon \mathsf{L} \mid Q) \\ = & \displaystyle\sum_{i=0}^{k} q_{(xs^i)} \binom{|xs|}{i} \\ = & \displaystyle\sum_{i=0}^{k-1} q_{(xs^i)} \binom{|xs|}{i} + q_{(xs^k)} \binom{|xs|}{k} \\ = & \displaystyle\sum_{i=0}^{k-1} (q'_{(l^i)} + q'_{(l^{(i+1)})}) \binom{|xs|}{i} + q'_{(xs^k)} \binom{|xs|}{k} & \text{Definition 4.14} \\ = & \displaystyle\sum_{i=0}^{k} q'_{(l^i)} \binom{|xs|+1}{i} & \text{Lemma 4.15} \\ = & \Phi(\texttt{cons}\ x\ xs \mid Q') \end{array}$$

Next we show the deconstructor axioms in a similar fashion.

Case. The `nil` case follows from the same argument as presented above.

Case. $\mathsf{c}\, x_1 \dots x_n = \texttt{cons}\ x\ xs$:

$$
\begin{aligned}
&\Phi(\Gamma, \texttt{cons}\ x\ xs \mid Q) \\
&= \sum_{i=0}^{k} \sum_{a_1+\dots+a_n+j=i} q_{(\vec{x},l^j)} \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|+1}{j} \\
&= \sum_{i=0}^{k-1} \sum_{a_1+\dots+a_n+j=i} (q_{(\vec{x},l^j)} + q_{(\vec{x},l^{(j+1)})}) \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|}{j} \\
&\quad + \sum_{a_1+\dots+a_n+j=i} q_{(\vec{x},l^k)} \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|}{k} && \text{Lemma 4.15} \\
&= \sum_{i=0}^{k-1} \sum_{a_1+\dots+a_n+j=i} p_{(\vec{x},xs^j)} \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|}{j} \\
&\quad + \sum_{a_1+\dots+a_n+j=i} p_{(\vec{x},l^k)} \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|}{k} && \text{Definition 4.14} \\
&= \sum_{i=0}^{k} \sum_{a_1+\dots+a_n+j=i} p_{(\vec{x},xs^j)} \binom{|x_1|}{a_1} \dots \binom{|x_n|}{a_n} \binom{|xs|}{j} \\
&= \Phi(\Gamma, xs : \mathsf{L} \mid P)
\end{aligned}
$$

Finally, using the cost free constraints, we proof $\exists J, \vec{P}, \vec{P'}.\ \forall i \in J.\ \Gamma | P_i \vdash^{\text{cf}} e_1 : \alpha | P'_i \Rightarrow \Phi^{\times\triangle}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\triangle}(x : \alpha, \Delta \mid R)$. Let $\vec{P}, \vec{P'}$ be vectors of resource annotations indexed by J as defined in the instance constraints. Then let us assume

$$\forall \vec{y} \in J.\ \Phi(\Gamma \mid P_i) \geq \Phi(x : \mathsf{L} \mid P'_i)$$

By unfolding the definitions and instance constraints we obtain

$$
\begin{aligned}
\forall \vec{y} \in J.\ \sum_{\vec{x}} p^{(\vec{y})}_{(\vec{x})} \prod_{j=1}^{n} \binom{|x_j|}{a_j} &\geq \sum_{i=0}^{k} p^{(\vec{y})}_{(x^i)} \binom{|x|}{i} \\
\forall \vec{y} \in J.\ \sum_{\vec{x}} q_{(\vec{x},\vec{y})} \prod_{j=1}^{n} \binom{|x_j|}{a_j} &\geq \sum_{i=0}^{k} r_{(\vec{y},x^i)} \binom{|x|}{i}\ .
\end{aligned}
$$

Since $\binom{|l|}{k}$ is non-negative for any list l and every $k \geq 0$, we obtain by monotonicity of

multiplication

$$\forall \vec{y} \in J.\ \sum_{\vec{x}} q_{(\vec{x},\vec{y})} \prod_{j=1}^{n} \binom{|x_j|}{a_j} \prod_{l=1}^{m} \binom{|y_l|}{a_l} \geq \sum_{i=0}^{k} r_{(\vec{y},x^i)} \prod_{l=1}^{m} \binom{|y_l|}{a_l} \binom{|x|}{i}$$

$$\sum_{\vec{x},\vec{y}} q_{(\vec{x},\vec{y})} \prod_{j=1}^{n} \binom{|x_j|}{a_j} \prod_{l=1}^{m} \binom{|y_l|}{a_l} \geq \sum_{i=0}^{k} \sum_{\vec{y}} r_{(\vec{y},x^i)} \prod_{l=1}^{m} \binom{|y_l|}{a_l} \binom{|x|}{i}$$

$$\sum_{i=0}^{k} \sum_{a_1+\cdots+a_n=i} q_{(\vec{x},\vec{y})} \prod_{j=0}^{n} \binom{|z_n|}{a_n} \geq \sum_{i=0}^{k} \sum_{a_1+\cdots+a_n+j=i} r_{(\vec{y},x^i)} \prod_{l=1}^{n} \binom{|y_l|}{a_l} \binom{|x|}{j},$$

which is exactly the definition of the theorem. □

4.2.3 Log-Linear Potential for Lists

As we have seen in Chapter 2, in order to express amortized costs of the form $n \cdot \log_2 n$, we require a log-linear potential function. To define its basic potential functions, we draw inspiration from both the polynomial and the logarithmic analysis and arrive at the following formulation.

$$\phi_{(l^{(a,b)},c)}(l) = |l|^a \cdot \log_2(b \cdot |l| + c)$$

Note that in contrast to the previous examples, here we have tuples as variable degrees, since indices need to track multiple parameters per variable. To keep the templates tractable we restrict to $a, b \in \{0, 1\}$ and $c \in \{0, 1, 2\}$. Note that we define the size of lists in the following slightly unusual way.

$$|\texttt{nil}| := 1$$
$$|\texttt{cons}\ x\ xs| := 1 + |xs|$$

This definition aligns with how we specify the size of a tree as the number of leaves. In this way, we can express that an empty list has the same potential as an empty tree. This fact, in turn, allows us to "type convert" potential between lists and trees, in an inductive fashion.

The log-linear potential function is restricted to the univariate setting, so we do not provide basic potential functions with multiple arguments. We define this new potential again in the usual way, as the linear combination of its basic potential functions.

Definition 4.18. Let Q be a resource annotation. The log-linear potential function Φ_{nlog} is defined a follows.

$$\Phi_{\mathsf{nlog}}(l_1, \ldots, l_n) = \sum_{\substack{1 \leq i \leq n,\\ a,b \in \{0,1\},\\ c \in \{0,1,2\}}} q_{(l_i^{(a,b)},c)} \cdot |l_i|^a \cdot \log_2(b \cdot |l_i| + c)$$

The constructor and deconstructor constraints in C_{nlog} are based on the following simple identity, between a list l and its tail xs.

$$\begin{aligned} |l| \cdot \log_2(|l|) &= (|xs| + 1) \cdot \log_2(|xs| + 1) \\ &= |xs| \cdot \log_2(|xs| + 1) + \log_2(|xs| + 1) \end{aligned}$$

Building on this intuition, we define a shift function over the indices of resource annotations. This function will play a key role in specifying the constraints for the `cons` constructor.

$$\triangleright(q_(x^{(a,b)}, c)) := \begin{cases} \{(x^{(a,b)}, c+b), (x^{(a-1,b)}, c+b)\} & a > 0 \wedge (c+b \leq 2) \\ \{(x^{(a,b)}, c+b)\} & c+b \leq 2 \\ \{\} & \text{otherwise} \end{cases}$$

With respect to expert knowledge, for the log-linear potential, it is sufficient to include the inequalities presented in Lemma 4.19. We note that this potential function, to the best of our knowledge, does not lend itself to a more smooth or general definition. Nonetheless, this underscores the advantage of highly specialized potential instances, as they allow to express such edge cases.

Lemma 4.19. *Let $n \in \mathbb{N}, n \geq 1$, then*

$$\log_2(n) \leq \log_2(n+1) \tag{4.4}$$

$$\log_2(n) \leq n \cdot \log_2(n) \tag{4.5}$$

$$n \cdot \log_2(n) \leq n \cdot \log_2(n+1) \tag{4.6}$$

$$\log_2(n) \leq n \tag{4.7}$$

Proof. The inequalities 4.4 to 4.6 are instances of the monotonicity of the logarithm and multiplication. For the inequality 4.7 we proceed as follows. First we introduce an equivalent recursive definition $\log_{2\mathbb{N}_1}$ for $\log_2$ for the strictly positive natural numbers $\mathbb{N}_1$.

$$\log_{2\mathbb{N}_1}(n) := \begin{cases} \log_{2\mathbb{N}_1}(\frac{n}{2}) + 1 & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

We then proceed to prove the inequality by strong induction over n.

Base . $\log_{2\mathbb{N}_1}(1) = 0 \leq 1$

Step .

$$\begin{aligned} \log_{2\mathbb{N}_1}(n) &= \log_{2\mathbb{N}_1}(\frac{n}{2}) + 1 \\ &\leq \frac{n}{2} + 1 && \text{IH} \\ &\leq n \end{aligned}$$

□

For the rest of this section we proceed by introducing and verifying the potential specific constraints.

Definition 4.20. The potential specific constraints C_{nlog} are defined as follows.

Constructors :

`nil`. Let Q and Q' be resource annotations, where $\mathsf{vars}(Q) = \emptyset$ and $\mathsf{vars}(Q') = \{y\}$.

$$q_{(2)} = q'_{(x^{(0,1)},1)} + q'_{(x^{(1,0)},2)} + q'_{(x^{(1,1)},1)} + q'_{(2)} \qquad q_{(1)} = q'_{(x^{(0,1)})}$$

`cons` x xs: Let Q and Q' be resource annotations with $\mathsf{vars}(Q) = \{x, xs\}$ and $\mathsf{vars}(Q') = \{y\}$.

$$q_i = \sum_{i \in \triangleright(j)} q'_j$$

Deconstructors :

`nil`. Let Q and $(P + K)$ be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X$, with x being the deconstructed variable and $\vec{x} \in I(X)$.

$$p_{(2)} = q_{(x^(0,1),1)} + q_{(x^(1,0),2)} + q_{(x^(1,1),1)} + q_{(2)} \qquad p_{(1)} = q_{(x^{(0,1)})}$$

`cons` x xs: Let Q and P be resource annotations, where $\mathsf{vars}(Q) = X \cup \{x\}$ and $\mathsf{vars}(P) = X \cup \{xs\}$, with x being the deconstructed variable and $\vec{x} = I(X)$.

$$p_i = \sum_{i \in \triangleright(j)} q_j$$

Cost Free Annotations Let Q, R be resource annotations and x, y variables. We define $J = \emptyset$ and the following constraints over Q and R.

$$r_{(\vec{y},c)} = q_{(\vec{y},c)}$$

Lemma 4.21. $(\mathsf{L}, \Phi_{\mathsf{nlog}}, \Sigma_{\leq_{\mathsf{nlog}}}, C_{\mathsf{nlog}})$ *is a potential instance.*

Proof. First we verify that the constructor axioms hold. For this we distinguish between the two possible constructors for the list type.

Case. $\mathsf{c}\, x_1 \dots x_n =$ `nil`:

$$\begin{aligned}
\Phi(\varnothing \mid Q) &= q_{(2)} \cdot \log_2(2) + q_{(1)} \cdot \log_2(1) \\
&= q_{(2)} \cdot 1 + q_{(1)} \cdot \log_2(1) \\
&= (q'_{(x^{(0,1)},1)} + q'_{(x^{(1,0)},2)} + q'_{(x^{(1,1)},1)} + q'_{(2)}) \cdot 1 + q'_{(x^{(0,1)})} \cdot \log_2(1) \\
&= q'_{(x^{(0,1)},1)} \cdot \log_2(|\texttt{nil}| + 1) \\
&\quad + q'_{(x^{(1,0)},2)} \cdot |\texttt{nil}| \cdot \log_2(2) \\
&\quad + q'_{(x^{(1,1)},1)} \cdot |\texttt{nil}| \cdot \log_2(|\texttt{nil}| + 1) \\
&\quad + q'_{(2)} \cdot \log_2(2) \\
&\quad + q'_{(x^{(0,1)})} \cdot \log_2(1) \\
&= \Phi(\texttt{nil} \mid Q')
\end{aligned}$$

Case. $\mathsf{c}\, x_1 \dots x_n = \texttt{cons}\ x\ xs$:

$$
\begin{aligned}
&\Phi(x : \mathsf{B}, xs : \mathsf{L} \mid Q) \\
&= \sum_{\substack{a,b \in \{0,1\} \\ c \in \{0,1,2\}}} q_{(xs^{(a,b)},c)} \cdot |xs|^a \cdot \log_2(b \cdot |xs| + c) \\
&= \sum_{\substack{a,b \in \{0,1\} \\ c \in \{0,1,2\}}} \left(\sum_{(xs^{(a,b)},c) \in \triangleright (xs^{(a',b')},c')} q'_{(xs^{(a',b')},c')} \right) \cdot |xs|^a \cdot \log_2(b \cdot |xs| + c) \\
&= \sum_{\substack{a'=1, \\ b' \in \{0,1\}, c' \in \{0,1,2\}, \\ c'+b' \le 2}} q'_{(y^{(a',b')},c')} \cdot |xs| \cdot \log_2(b' \cdot |xs| + c' + b') \\
&\quad + q'_{(y^{(a',b')},c')} \cdot \log_2(b' \cdot |xs| + c' + b') \\
&\quad + \sum_{\substack{a'=0, \\ b' \in \{0,1\}, c' \in \{0,1,2\}, \\ c'+b' \le 2}} q'_{(y^{(a',b')},c')} \cdot \log_2(b' \cdot |xs| + c' + b') \\
&= \sum_{\substack{a',b' \in \{0,1\} \\ c' \in \{0,1,2\}}} q'_{(y^{(a',b')},c')} \cdot |\texttt{cons}\ x\ xs|^{a'} \cdot \log_2(b' \cdot |\texttt{cons}\ x\ xs| + b') \\
&= \Phi(\texttt{cons}\ x\ xs : \mathsf{L} \mid Q)
\end{aligned}
$$

When examining the deconstructor constraints, we observe that they follow the same structure as the constructor constraints. Therefore, rather than providing a rigorous proof, we outline how a proof can be derived from the constructor proofs by substituting the annotations and reversing the steps of the argument.

Lastly we confirm

$$\exists J, \vec{P}, \vec{P'}.\ \forall i \in J.\ \Gamma | P_i \vdash^{\mathrm{cf}} e_1 : \alpha | P'_i \Rightarrow \Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x : \alpha, \Delta \mid R)\ .$$

Let Q and R be resource annotations and Γ, Δ typing contexts, where $\vec{x} \in I(\mathsf{vars}(\Gamma))$ and $\vec{y} \in I(\mathsf{vars}(\Delta))$. Also let x be a variable. Since $J = \varnothing$, the antecedent is trivially satisfied. In addition since, the log-linear potential has only univariate basic potential functions, the consequent

$$\Phi^{\times\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times\Delta}(x : \alpha, \Delta \mid R)$$

simplifies to

$$q_{(\vec{y},c)} \cdot \phi_{(\vec{y},c)} \geq r_{(\vec{y},c)} \cdot \phi_{(\vec{y},c)}$$

, which is guaranteed by the instance constraints.

□

4.3 Mixed Potential Functions

In the following section we build upon the intuition of an analysis with mixed potential functions, given in Chapter 2. We start with a formal definition of mixed potentials.

In order to allow multiple potential functions for different data types in our type system, we again represent each of them with a resource annotation. In contrast to before these resource annotations do not annotate the whole typing context, but rather a subset of variables with the same data type as the potential represented by the annotation. This leads us to a new notion of a typing context that allows to group variables of the same type together with the corresponding annotation.

Definition 4.22. We define an annotated typing context Γ as a sequence of typing contexts containing only variables of a single type together with a resource annotation for that type.

$$\Gamma := \Gamma_\alpha, \Gamma_\beta, \cdots = (x_1, \ldots, x_n : \alpha \mid Q), (y_1, \ldots, y_m : \beta \mid P), \ldots$$

Bearing this definition in mind, we are now ready to define a mixed potential function. A mixed potential function is the sum of individual potential functions, specifically the template potential functions we have encountered thus far.

Definition 4.23. A mixed potential function Φ, consisting of the individual potential functions $\Phi_1, \ldots, \Phi_n$ is defined as follows.

$$\Phi(\Gamma) := \Phi_{\alpha_1}(\Gamma_{\alpha_1}) + \cdots + \Phi_{\alpha_n}(\Gamma_{\alpha_n})$$

A multi potential analysis is now parameterized by a mapping from types to potential instances that dictates which potential function is used for each data type in the program. When a data type is not assigned a potential function, it is implicitly mapped to the constant zero function. In an annotated typing context, we omit the annotation of such types. Additionally in our analysis we assume that booleans and other primitive data types never bear potential.

Since the resource annotations are now part of the context, we lift the arithmetic operations defined on resource annotations to annotated contexts. Recall that adding a constant to a resource annotation, is defined by increasing the coefficient of the basic potential function $\phi(x_1, \ldots, x_n) = 1$. Since every potential function in the annotated context can have such a constant basic potential function, it is not immediately clear to which one we should add the constant. In fact it could even be split between those functions. We encode this fact in the following definition and let the constraint solver choose how to split up the constant.

Definition 4.24. We define $\Gamma' := \Gamma + K$ as

$$\Gamma' = (\vec{x_1} : \alpha_1 \mid Q_{\alpha_1} + K_1), \ldots, (\vec{x}_n : \alpha_n \mid Q_{\alpha_n} + K_n)$$

where $K = K_1 + \cdots + K_n$

For multiplication and addition of annotated contexts, we just lift the point-wise definition on resource annotations.

Definition 4.25. Let $\Gamma = (\vec{x_1}:\alpha_1 \mid Q_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid Q_{\alpha_n})$ and $\Delta = (\vec{x_1}:\alpha_1 \mid P_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid P_{\alpha_n})$. We define $E := \Gamma \cdot \Delta$ as

$$E = (\vec{x_1}:\alpha_1 \mid Q_{\alpha_1} \cdot P_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid Q_{\alpha_n} \cdot P_{\alpha_n})$$

Definition 4.26. Let $\Gamma = (\vec{x_1}:\alpha_1 \mid Q_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid Q_{\alpha_n})$ and $\Delta = (\vec{x_1}:\alpha_1 \mid P_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid P_{\alpha_n})$. We define $E := \Gamma + \Delta$ as

$$E = (\vec{x_1}:\alpha_1 \mid Q_{\alpha_1} + P_{\alpha_1}), \ldots, (\vec{x}_n:\alpha_n \mid Q_{\alpha_n} + P_{\alpha_n})$$

The key observation for mixed potential functions is that the constraints over the individual functions, defined in the instances, can be used to generate constraints for a mixed potential function. Assume in our abstract type system we have a constraint of the form $\Phi(\Gamma) = \Phi(\Delta)$, where Φ is a mixed potential function and Γ, Δ are annotated contexts. We observe that a mixed potential function is the sum of some individual potential functions and the arguments can be separated into ordinary typing contexts as well.

Further assume that the instance constraints for the components assure the following.

$$\forall i \leq n. \Phi(\Gamma_{\alpha_i}) = \Phi(\Delta_{\alpha_i})$$

This implies that the desired constraint over the mixed potentials holds as well.

$$\begin{aligned}\Phi(\Gamma) &= \Phi_{\alpha_1}(\Gamma_{\alpha_1}) + \cdots + \Phi_{\alpha_n}(\Gamma_{\alpha_n}) \\ &= \Phi_{\alpha_1}(\Delta_{\alpha_1}) + \cdots + \Phi_{\alpha_n}(\Delta_{\alpha_n}) \\ &= \Phi(\Delta)\end{aligned}$$

In the following we redefine our previous typing rules, by integrating annotated typing contexts.

The typing rule (app : base) is a straight-forward extension of its non-mixed counterpart.

$$\frac{\alpha \text{ and } \beta \text{ do not bear potential}}{x_1, \ldots, x_n:\alpha|Q \vdash f(x_1, \ldots, x_n):\beta|Q'} \text{ (app : base)}$$

Since none of the involved types bear potential, we can omit the trivial constraint $\Phi(\Gamma, x_1:\alpha_1, \ldots, x_n:\alpha_n \mid 0) = \Phi(f(x_1, \ldots, x_n):\beta \mid 0)$.

In order to lift the rule (var) to the mixed setting we only need to change the ordinary typing contexts to annotated contexts, which again is a straightforward extension.

$$\frac{x \text{ a variable}}{x:\alpha|Q_\alpha \vdash x:\alpha|Q_\alpha} \text{ (var)}$$

Examining the judgment for the adapted rule (const), we have a typing context for the variables $x_1, \ldots, x_m$ of constructor type α and an annotated context for variables of other types. This is necessary for constructors like `node` $l : \mathsf{T}\ a : \mathsf{B}\ r : \mathsf{T}$, which include some non-recursive types (e.g. B). We note that this definition relies on the assumption that potential bearing constructors do not include other potential bearing types. For example, if in our analysis we assign potential to both lists and trees, we do not allow trees that have lists as their element type. The constraint over the potential function, relies only on annotated contexts of the constructor type, so it can be guaranteed by the corresponding instance axiom.

$$\frac{\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \qquad \Phi(x_1, \ldots, x_m : \alpha \mid Q) = \Phi(\mathsf{c}\, x_1\ \ldots\ x_n : \alpha \mid Q')}{\Gamma, (x_1, \ldots, x_m : \alpha \mid Q) \vdash \mathsf{c}\, x_1\ \ldots\ x_n : \alpha | Q'} \ (\mathsf{const})$$

The (match) rule is adapted in similar way as the (const) rule. The premise of the rule assures again, that the potential for the deconstructed data type is split correctly according to the pattern. Note that the potential of all other data types is left untouched. For this reason we can again decompose and assure the premises with existing potential instance constraints.

$$\frac{\Phi(\vec{x}, \mathsf{c}_i\, \vec{x_i} : \alpha \mid Q) = \Phi(\vec{x}, \vec{x_i} : \alpha \mid P) \qquad \Gamma, (\vec{x}, \vec{x_i} : \alpha \mid P) \vdash e_i : \beta | Q'}{\Gamma, (\vec{x}, x : \alpha \mid Q) \vdash \texttt{match}\ x\ \texttt{with}\ \mid\ \mathsf{c}_1\, \vec{x_1}\ \texttt{->}\ e_1 \mid\ \ldots\ \mid\ \mathsf{c}_m\, \vec{x_m}\ \texttt{->}\ e_m : \beta | Q'} \ (\mathsf{match})$$

The rules (ite), (ite : coin) and (app) follow the same principle as before, except that we now deploy arithmetic over annotated contexts, rather than individual resource annotations.

Finally we discuss the most interesting rule, (let). In the previous type system we have seen two different variants of this rule. First the more involved version (let), which distributes the potential that comes from the binding expression and a simpler version (let : base), that only deals with constant potential in the binding expression. The simpler version is applied when the binding does not have a potential bearing type. For the mixed potentials we observe that the initial annotated context, contains multiple types, of which at most one matches the type of the binding, yet all of the potential must be distributed. In the definition of the adapted let rule we combine our previous definition for (let) and (let : base), into single rule. For the non-binding types we apply constraints corresponding to the base case and for the binding type we have constraints that respect the result potential of the binding, in the same way as the previous (let) rule. We note that this new definition eliminates the need for two distinct rules, even in the case of a mono potential analysis.

$$\frac{\begin{array}{c} Z = (\Delta_\alpha, x{:}\alpha \mid R_\alpha), Z_{\beta_1}, \ldots, Z_{\beta_n} \qquad \beta \neq \alpha \qquad \Phi^{nc}(E_\beta) = \Phi^{nc}(\Gamma_\beta) \\ \Phi^{nc}(Z_\beta) = \Phi^{nc}(\Delta_\beta) \qquad \Phi^{c}(Z) = (\Phi^{c}(\Gamma) - \Phi^{c}(E)) + \Phi^{c}(e_1{:}\alpha \mid P') \\ \Phi^{nc}(E_\alpha) = \Phi^{nc}(\Gamma_\alpha) \qquad \Phi^{v}(Z_\alpha) = \Phi^{v}(\Delta_\alpha) \qquad \Phi^{nc}(x{:}\alpha \mid R) = \Phi^{nc}(x{:}\alpha|P') \\ \Phi^{\times\Delta}(\Gamma_\alpha, \Delta_\alpha) \geq \Phi^{\times\Delta}(x{:}\alpha, \Delta_\alpha \mid R) \qquad E \vdash e_1{:}\alpha|P' \qquad Z \vdash e_2{:}\gamma|Q' \end{array}}{\Gamma, \Delta \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2{:}\gamma|Q'}\ (\mathsf{let})$$

A complete account of the typing rules for mixed potential is given in Figure 4.3 and Figure 4.4. Next we present an adapted version of the soundness theorem for the new type system.

Theorem 4.27. *Soundness Theorem. Let P be well-typed and let σ be a substitution. Suppose $\Gamma \vdash e{:}\alpha|Q'$ and $\sigma \overset{c}{\vdash} e \Rightarrow \mu$. Then, we have $\Phi(\sigma;\Gamma) \geq c + \mathbb{E}_\mu(\lambda v.\Phi(v \mid Q'))$. Further, if $\Gamma \vdash^{cf} e{:}\alpha|Q'$, then $\Phi(\sigma;\Gamma) \geq \Phi(v \mid Q')$*

Proof. We can, in principle, prove this again via induction. However, we will only provide a proof sketch, as the reasoning is essentially the same as in the previous soundness proof. For example, in the case of (const), which involves a single resource annotation, soundness follows directly from the argument presented earlier. For (match), although more than one type is involved, only one annotation is modified, while the other potential remains unchanged. This ensures soundness through the monotonicity of addition. Finally, in the (let) case, we combine the sub-proofs of (let) and use the monotonicity of addition once again to establish the theorem. □

4.3.1 Inter-Potential Weakening

Extending the weakening rule is not as straightforward as the other structural rules. The main challenge lies in adapting the symbolic comparison method, used for template potential functions, to accommodate mixed potentials, as illustrated by the following equation.

$$\Phi(\Gamma) = \Phi_{\alpha_1}(\Gamma_{\alpha_1}) + \cdots + \Phi_{\alpha_n}(\Gamma_{\alpha_n}) \leq \Phi_{\alpha_1}(\Delta_{\alpha_1}) + \cdots + \Phi_{\alpha_n}(\Delta_{\alpha_n}) = \Phi(\Delta)$$

Intuitively it might seem like we can avoid any extra considerations and just compare the individual potential functions separately, as described in Chapter 3. In principle this approach is valid, due to the monotonicity of addition, but it is limited in that potential cannot be transferred between the individual potential functions. The attentive reader might have already noticed that we allow constants in every potential function, which is redundant since these constants could be represented by a single coefficient for all potential functions. However, such a representation would compromise the modularity we have maintained in the typing rules so far. As we will demonstrate in Chapter 5

the capability of transferring constant potential between different functions is crucial for enabling type conversion for potentials. For this reason, we require a combined linearization approach over the individual potential functions.

As described in Chapter 3, the symbolic comparison of template potential functions is achieved through Farkas' Lemma, which relies on expert knowledge specific to the potential instance, as detailed in Section 4.2. Let Γ be an annotated context, and let α be a type in this context. We then denote the expert knowledge for α as $(A_\alpha, \vec{b}_\alpha)$. For Γ have $\{(A_{\alpha_1}, \vec{b}_{\alpha_1}), \ldots, (A_{\alpha_n}, \vec{b}_{\alpha_n})\}$ as the expert knowledge provided by the individual potential instances. The idea is now to combine the individual matrices into a combined matrix (A, b). This enables us to apply Farkas' Lemma to the combined expert knowledge and obtain a linearization of $\Phi(\Gamma) \geq \Phi(\Delta)$ directly. (A, b) is constructed as follows.

$$(A, b) := \left(\left(\begin{array}{ccc} A_{\alpha_1} & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & A_{\alpha_n} \\ \hline & A_0 & \\ & A_1 & \end{array} \right), \left(\begin{array}{c} b_{\alpha_1} \\ \vdots \\ b_{\alpha_n} \\ \hline 0 \\ 0 \end{array} \right) \right)$$

The upper rows correspond to our previous intuition and integrate the existing expert knowledge without modification. These rows only represent knowledge about basic potential function concerning one data type. The lower rows represent inter-potential expert knowledge and allow to move constant potential from one data type to another. We remember that each potential instance has a constant one potential function, which is involved in A_1 and possibly a constant zero potential function, which is part of A_0. In the following we only demonstrate how A_1 is defined, since A_0 follows the same principle. For every pair of types α, β in Γ we define two rows in A_1, representing the following inequalities.

$$\mathsf{one}(Q_\alpha) \leq \mathsf{one}(Q_\beta) \qquad\qquad \mathsf{one}(Q_\beta) \leq \mathsf{one}(Q_\alpha)$$

This effectively leaves us with $\mathsf{one}(Q_\alpha) = \mathsf{one}(Q_\beta)$, which after the application of Farkas' Lemma, allows the constraint solver to transfer potential between the two. Note that if an instance does not define a zero basic potential function, we just omit the corresponding rows in A_0.

$$\frac{a_i \neq T \qquad \beta \neq T \qquad \Phi(x_1:\alpha_1,\ldots,x_n:\alpha_n \mid Q) = \Phi(f(x_1,\ldots,x_n):\beta \mid Q')}{x_1:\alpha_1,\ldots,x_n:\alpha_n|Q \vdash f(x_1,\ldots,x_n):\beta|Q'}\ (\mathsf{app}:\mathsf{base})$$

$$\frac{x \text{ a variable} \qquad \Phi(x:\alpha \mid Q) = \Phi(x:\alpha \mid Q')}{x:\alpha|Q \vdash x:\alpha|Q'}\ (\mathsf{var})$$

$$\frac{\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \qquad \Phi(x_1:\alpha_1,\ldots,x_n:\alpha_n \mid Q) = \Phi(\mathsf{c}\,x_1\ \ldots\ x_n:\alpha \mid Q')}{x_1:\alpha,\ldots,x_n:\alpha_n|Q \vdash \mathsf{c}\,x_1\ \ldots\ x_n:\alpha|Q'}\ (\mathsf{const})$$

$$\frac{\Gamma|Q \vdash e_1:\alpha|Q' \quad \Gamma|Q \vdash e_2:\alpha|Q'}{\Gamma, x:\mathsf{Bool}|Q \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\alpha|Q'}\ (\mathsf{ite})$$

$$\frac{\Gamma|Q_1 \vdash e_1:\alpha|Q' \quad \Gamma|Q_2 \vdash e_2:\alpha|Q' \quad p = {}^{a}\!/_{b} \quad Q = p\cdot Q_1 + (1-p)\cdot Q_2}{\Gamma|Q \vdash \texttt{if coin}\ a/b\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\alpha|Q'}\ (\mathsf{ite}:\mathsf{coin})$$

$$\frac{\Phi(\Gamma, \mathsf{c}_i\,\vec{x_i} \mid Q) = \Phi(\Gamma, \vec{x_i} \mid Q_i) \qquad \Gamma, \vec{x_i}|Q_i \vdash e_i:\beta|Q'}{\Gamma, x:\alpha|Q \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \mathsf{c}_1\,\vec{x_1}\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathsf{c}_m\,\vec{x_m}\ \texttt{->}\ e_m:\beta|Q'}\ (\mathsf{match})$$

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \qquad \Phi^{nc}(\Delta \mid R) = \Phi^{nc}(\Delta \mid Q) \\ \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \\ \Gamma|P \vdash e_1:\alpha|P' \qquad \Delta, x:\alpha|R \vdash e_2:\beta|Q' \qquad \alpha \neq T\end{array}}{\Gamma, \Delta|Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2:\beta|Q'}\ (\mathsf{let}:\mathsf{base})$$

$$\frac{\begin{array}{c}\Phi^{nc}(\Gamma \mid Q) = \Phi^{nc}(\Gamma \mid P) \qquad \Phi^{v}(\Delta \mid R) = \Phi^{v}(\Delta \mid Q) \\ \Phi(\varnothing \mid R) = (\Phi(\varnothing \mid Q) - \Phi(\varnothing \mid P)) + \Phi(\varnothing \mid P') \qquad \Phi^{nc}(x:\alpha \mid R) = \Phi^{nc}(e_1:\alpha|P') \\ \Phi^{\times_\Delta}(\Gamma, \Delta \mid Q) \geq \Phi^{\times_\Delta}(x:\alpha, \Delta \mid R) \qquad \Gamma|P \vdash e_1:\alpha|P' \qquad \Delta, x:\alpha|R \vdash e_2:\beta|Q'\end{array}}{\Gamma, \Delta|Q \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2:\beta|Q'}\ (\mathsf{let})$$

$$\frac{\alpha_1 \times \cdots \times \alpha_n|P \to \beta|P' \in \mathcal{F}(f) \quad \alpha_1 \times \cdots \times \alpha_n|Q \to \beta|Q' \in \mathcal{F}^{\mathrm{cf}}(f) \quad K \in \mathbb{Q}_0^+}{x_1:\alpha_1,\ldots,x_n:\alpha_n|(P + K\cdot Q) \vdash f(x_1,\ldots,x_n):\beta|(P' + K\cdot Q')}\ (\mathsf{app})$$

$$\frac{\Gamma|Q \vdash e:\alpha|Q'}{\Gamma|Q \vdash e^{\checkmark\, a/b}:\alpha|Q' - {}^{a}\!/_{b}}\ (\mathsf{tick}:\mathsf{defer})$$

Figure 4.1: Syntax directed rules of the abstract type system. T denotes the potential bearing type of the analysis.

$$\frac{\Phi(\Gamma \mid Q) = \Phi(\Gamma \mid R) \quad \Gamma|R \vdash e\colon\beta|Q'}{\Gamma, x\colon\alpha|Q \vdash e\colon\beta|Q'}\ (\mathsf{w}:\mathsf{var}) \qquad \frac{\Phi(\Gamma, x\colon\alpha, y\colon\alpha \mid P) = \Phi(\Gamma, z\colon\alpha \mid Q) \quad \Gamma, x\colon\alpha, y\colon\alpha|P \vdash e[x,y]\colon\beta|Q'}{\Gamma, z\colon\alpha|Q \vdash e[z,z]\colon\beta|Q'}\ (\mathsf{share})$$

$$\frac{\Gamma|P \vdash e\colon\alpha|P' \quad \Phi(\Gamma \mid P) \leq \Phi(\Gamma \mid Q) \quad \Phi(e\colon\alpha \mid P') \geq \Phi(e\colon\alpha \mid Q')}{\Gamma|Q \vdash e\colon\alpha|Q'}\ (\mathsf{w})$$

$$\frac{\Gamma|Q \vdash e\colon\alpha|Q' \quad K \geqslant 0}{\Gamma|Q + K \vdash e\colon\alpha|Q' + K}\ (\mathsf{shift})$$

$$\frac{\begin{array}{c}\Gamma|P \vdash^{\mathrm{cf}} e\colon\alpha|P' \qquad \phi \text{ is monotone} \\ \Phi(\Gamma \mid P) = q_\phi \cdot \phi(x) + c \qquad \Phi(\Gamma \mid Q) = q_\phi \cdot \phi(x+k) + c \\ \Phi(e\colon\alpha \mid P') = q_\phi \cdot \phi(y) + c \qquad \Phi(e\colon\alpha \mid Q') = q_\phi \cdot \phi(y+k) + c\end{array}}{\Gamma|Q \vdash^{\mathrm{cf}} e\colon\alpha|Q'}\ (\mathsf{shiftm})$$

Figure 4.2: Structural rules of the abstract type system.

$$\frac{\alpha \text{ and } \beta \text{ do not bear potential}}{x_1, \ldots, x_n : \alpha | Q \vdash f(x_1, \ldots, x_n) : \beta | Q'} \text{ (app : base)} \qquad \frac{x \text{ a variable}}{x : \alpha | Q_\alpha \vdash x : \alpha | Q_\alpha} \text{ (var)}$$

$$\frac{\mathsf{c} \text{ is an } n\text{-ary constructor for type } \alpha \quad \Phi(x_1, \ldots, x_m : \alpha \mid Q) = \Phi(\mathsf{c}\, x_1 \ \ldots \ x_n : \alpha \mid Q')}{\Gamma, (x_1, \ldots, x_m : \alpha \mid Q) \vdash \mathsf{c}\, x_1 \ \ldots \ x_n : \alpha | Q'} \text{ (const)}$$

$$\frac{\Gamma \vdash e_1 : \alpha | Q'_\alpha \quad \Gamma \vdash e_2 : \alpha | Q'_\alpha}{\Gamma, x : \mathsf{Bool} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \alpha | Q'_\alpha} \text{ (ite)}$$

$$\frac{\Gamma_1 \vdash e_1 : \alpha | Q' \quad \Gamma_2 \vdash e_2 : \alpha | Q' \quad p = {}^a\!/_b \quad \Gamma = p \cdot \Gamma_1 + (1-p) \cdot \Gamma_2}{\Gamma \vdash \texttt{if coin}\ a/b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \alpha | Q'} \text{ (ite : coin)}$$

$$\frac{\Phi(\vec{x}, \mathsf{c}_i\, \vec{x_i} : \alpha \mid Q) = \Phi(\vec{x}, \vec{x_i} : \alpha \mid P) \qquad \Gamma, (\vec{x}, \vec{x_i} : \alpha \mid P) \vdash e_i : \beta | Q'}{\Gamma, (\vec{x}, x : \alpha \mid Q) \vdash \texttt{match}\ x\ \texttt{with}\ \mid\ \mathsf{c}_1\, \vec{x_1}\ \texttt{->}\ e_1 \mid\ \ldots\ \mid\ \mathsf{c}_m\, \vec{x_m}\ \texttt{->}\ e_m : \beta | Q'} \text{ (match)}$$

$$\frac{\begin{array}{c} Z = (\Delta_\alpha, x : \alpha \mid R_\alpha), Z_{\beta_1}, \ldots, Z_{\beta_n} \qquad \beta \neq \alpha \qquad \Phi^{nc}(E_\beta) = \Phi^{nc}(\Gamma_\beta) \\ \Phi^{nc}(Z_\beta) = \Phi^{nc}(\Delta_\beta) \qquad \Phi^c(Z) = (\Phi^c(\Gamma) - \Phi^c(E)) + \Phi^c(e_1 : \alpha \mid P') \\ \Phi^{nc}(E_\alpha) = \Phi^{nc}(\Gamma_\alpha) \qquad \Phi^v(Z_\alpha) = \Phi^v(\Delta_\alpha) \qquad \Phi^{nc}(x : \alpha \mid R) = \Phi^{nc}(x : \alpha | P') \\ \Phi^{\times\Delta}(\Gamma_\alpha, \Delta_\alpha) \geq \Phi^{\times\Delta}(x : \alpha, \Delta_\alpha \mid R) \qquad E \vdash e_1 : \alpha | P' \qquad Z \vdash e_2 : \gamma | Q' \end{array}}{\Gamma, \Delta \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \gamma | Q'} \text{ (let)}$$

$$\frac{\Gamma \to \Gamma' \in \mathcal{F}(f) \quad \Delta \to \Delta' \in \mathcal{F}(f) \in \mathcal{F}^{\mathrm{cf}}(f) \quad K \in \mathbb{Q}_0^+}{(\Gamma + K \cdot \Delta) \vdash f(x_1, \ldots, x_n) : \beta | (\Gamma' + K \cdot \Delta')} \text{ (app)}$$

$$\frac{\Gamma | Q \vdash e : \alpha | Q'}{\Gamma | Q \vdash e^{\checkmark\, a/b} : \alpha | Q' - {}^a\!/_b} \text{ (tick : defer)}$$

Figure 4.3: Abstract typing rules for mixed potentials.

$$\frac{\Phi(\Gamma, (\vec{x}, x\colon\alpha \mid Q)) = \Phi(\Gamma, (\vec{x}\colon\alpha \mid R)) \quad \Gamma, (\vec{x}\colon\alpha \mid R) \vdash e\colon\beta|Q'}{\Gamma, (\vec{x}, x\colon\alpha \mid Q) \vdash e\colon\beta|Q'}\ (\mathsf{w} : \mathsf{var})$$

$$\frac{\Delta \vdash e\colon\alpha|P' \quad \Phi(\Gamma) \leq \Phi(\Delta) \quad \Phi(e\colon\alpha \mid P') \geq \Phi(e\colon\alpha \mid Q')}{\Gamma \vdash e\colon\alpha|Q'}\ (\mathsf{w})$$

$$\frac{\Gamma \vdash e\colon\alpha|Q' \quad K \geqslant 0}{\Gamma + K \vdash e\colon\alpha|Q' + K}\ (\mathsf{shift})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathrm{cf}} e\colon\beta|P' \qquad \phi \text{ is monotone} \\ \Phi(\Gamma) = q_\phi \cdot \phi(x) + c \qquad \Phi(\Gamma \mid Q) = q_\phi \cdot \phi(x+k) + c \\ \Phi(e\colon\beta \mid P') = q_\phi \cdot \phi(y) + c \qquad \Phi(e\colon\beta \mid Q') = q_\phi \cdot \phi(y+k) + c\end{array}}{\Gamma \vdash^{\mathrm{cf}} e\colon\alpha|Q'}\ (\mathsf{shiftm})$$

Figure 4.4: Structural rules of the for mixed potentials.

5 Case Study: Splay Tree from a List

In this chapter, we present a detailed demonstration of the mixed potential analysis through a case study focusing on the function `fromList`, introduced in Chapter 2. As with the example of the purely functional queue, we present the type derivation of the function step-by-step. With this description we highlight the necessity of the (shiftm) rule and inter-potential weakening. Figure 5.1 gives the code, we will analyze in the following, with the necessary transformations applied.

5.1 Type Derivation

As hinted in Chapter 2, the amortized cost of `fromList` is $2 \cdot |l| \cdot \log_2(|l|) + 1/2|l|$, which is exactly $|l|$ times the cost of `insert`. In order to verify this claim we perform type checking with the mixed potential function we have seen earlier. We start with an application of (match), for the match on the tree argument.

$$\dfrac{\dfrac{}{\varnothing \mid 1 \vdash \texttt{error} : \mathsf{T}|1/2\,\mathsf{rk}(t') + 1}\,(\mathsf{const})}{\begin{array}{l} l : \mathsf{L} \mid 2 \cdot |l| \cdot \log_2(|l|) + 1/2|l|, \\ t : \mathsf{T} \mid 1/2\,\mathsf{rk}(x_1) + \log_2(x_1) + \log_2(x_2) + 1/2\,\mathsf{rk}(x_2) + 1 \\ \vdash \texttt{error} \\ : \mathsf{T}|1/2\,\mathsf{rk}(t') + 1 \end{array}\,(\mathsf{w} : \mathsf{var})^* \quad (5.1)}{\begin{array}{l} l : \mathsf{L} \mid 2 \cdot |l| \cdot \log_2(|l|) + 1/2|l|, t : \mathsf{T} \mid 1/2\,\mathsf{rk}(t) + 1 \\ \vdash \texttt{match}\ t\ \texttt{with} \mid \texttt{node}\ x_1\ \_\ x_2 \rightarrow\ e_1 \mid \texttt{leaf} \rightarrow\ e_2 \\ : \mathsf{T}|1/2\,\mathsf{rk}(t') + 1 \end{array}\,(\mathsf{match})}$$

The left branch types the case in which t is not a leaf and we return the error constant. The potential of t is divided into two sub-trees according to the definition of rk, reflected in the instance constraints of the logarithmic potential. As described in the previous chapter, with (match), only the potential of the matched data type is modified, while the other potentials remain unchanged. Consequently, the full potential of the list l is still preserved after the match statement. Next, we observe a sequence of variable eliminations denoted by $(\mathsf{w} : \mathsf{var})^*$. The rule (var) eliminates one redundant variable at a time, i.e. a variable not present in the expression. The corresponding potential is subsequently set to zero and only the constant potential remains. The application of (const), might seem counter intuitive at first, but stems from the fact that with standard types `error` unifies with every other type. When we extend this notion to annotated types, `error` covers any potential difference. One could also argue that, in the error case, no value is

Figure 5.1: Code for `SplayTree.fromList`, in let normal form.

```
1 fromList :: (List Base * Tree Base) → Tree Base
2 fromList l t = match t
3   | node x1 _ x2 → error
4   | leaf   → match l
5     | nil  → leaf
6     | cons x xs → let x3 = leaf in
7       let x4 = fromList xs x3 in
8         insert x x4
```

The given code is the result of preprocessing the code of function `fromList` for the cost analysis, which transforms it into let-normal form.

actually returned; instead, the program execution halts, rendering the resulting potential irrelevant.

$$\dfrac{\dfrac{\dfrac{}{\varnothing_{\mathsf{L}} \mid 0, \varnothing_{\mathsf{T}} \mid 3/2 \vdash \texttt{leaf} : \mathsf{T}|1/2\,\mathsf{rk}(\texttt{leaf}) + 1}\,(\mathsf{const})}{\varnothing_{\mathsf{L}} \mid 1/2, \varnothing_{\mathsf{T}} \mid 3/2 \vdash \texttt{leaf} : \mathsf{T}|1/2\,\mathsf{rk}(\texttt{leaf}) + 1}\,(\mathsf{w}) \qquad (5.2)}{\begin{array}{l} l : \mathsf{L} \mid 2 \cdot |l| \cdot \log_2(|l|) + 1/2|l|, \varnothing_{\mathsf{T}} \mid 3/2 \\ \vdash \texttt{match}\ l\ \texttt{with} \mid \texttt{nil} \to\ e_1 \mid \texttt{cons}\ x\ xs \to\ e_2 \\ :\ \mathsf{T}|1/2\,\mathsf{rk}(t') + 1 \end{array}}\,(\mathsf{match}) \qquad (5.1)$$

Next, we examine the `leaf` case, which includes another match statement to deconstruct the list l. In the `nil` case, the list potential is $1/2$. This can be understood by recalling that the length of lists is defined such that $|\texttt{nil}| = 1$. The application of `const` requires the potentials of types other than the constructor to equal the right-hand side potential. This explains why, in this case, the list potential is weakened to zero. The equality for the tree potential holds, since $\mathsf{rk}(\texttt{leaf}) = 1$.

$$\dfrac{\dfrac{}{\varnothing_{\mathsf{T}} \mid 3/2 \vdash \texttt{leaf} : \mathsf{T}|1/2\,\mathsf{rk}(\texttt{leaf}) + 1}\,(\mathsf{const}) \qquad (5.3)}{\begin{array}{l} l : \mathsf{L} \mid 2 \cdot |xs| \cdot \log_2(|xs| + 1) + 2 \cdot \log_2(|xs| + 1) + 1/2|xs| + 1/2, \\ \varnothing_{\mathsf{T}} \mid 3/2 \\ \vdash \texttt{let}\ x_3 = \texttt{leaf}\ \texttt{in}\ e_3 \\ :\ \mathsf{T}|1/2\,\mathsf{rk}(t') + 1 \end{array}}\,(\mathsf{let}) \qquad (5.2)$$

For the `cons` case, observe that the potential of l shifted to its tail xs, and some constant potential. The constant tree potential is subsequently transferred to the binding

expression, to pay the for `leaf` constructor.

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{l} l\colon \mathsf{L} \mid 2\cdot|xs|\cdot\log_2(|xs|)+2\cdot\log_2(|xs|)+1/2|xs|+1/2,\\ x_3\colon \mathsf{T} \mid 1/2\,\mathsf{rk}(x_3)+1 \\ \vdash \texttt{fromlist xs x3} \\ :\ \mathsf{T}|1/2\,\mathsf{rk}(t')+2\log_2(|t'|)+3/2 \end{array}}\ (\mathsf{app})}{\begin{array}{l} l\colon \mathsf{L} \mid 2\cdot|xs|\cdot\log_2(|xs|+1)+2\cdot\log_2(|xs|+1)+1/2|xs|+1/2,\\ x_3\colon \mathsf{T} \mid 1/2\,\mathsf{rk}(x_3)+1 \\ \vdash \texttt{fromlist xs x3} \\ :\ \mathsf{T}|1/2\,\mathsf{rk}(t')+2\log_2(|t'|)+3/2 \end{array}}\ (\mathsf{w})\qquad (5.4)}{\begin{array}{l} l\colon \mathsf{L} \mid 2\cdot|xs|\cdot\log_2(|xs|+1)+2\cdot\log_2(|xs|+1)+1/2|xs|+1/2,\\ x_3\colon \mathsf{T} \mid 1/2\,\mathsf{rk}(x_3)+1 \\ \vdash \texttt{let } x_4 = \texttt{fromlist xs x3 in insert x x4} \\ :\ \mathsf{T}|1/2\,\mathsf{rk}(t')+1 \end{array}}\ (\mathsf{let})\qquad (5.3)$$

The body of the let expression contains another let binding for the recursive call to `fromList`. This call forms the tree argument of `insert`. Consequently, the potential of the bound variable aligns with the signature of `insert`, which is given by the expression $1/2\,\mathsf{rk}(t')+2\log_2(|t|)+3/2$. Through an application of the rule (w), the logarithmic terms are shifted from $\log_2(|xs|+1)$ to $\log_2(|xs|)$, to match the signature of `fromList`.

At this point, the terms $2\cdot\log_2(|xs|)+1/2$ and $2\cdot\log_2(|t'|)+1/2$ still deviate from the signature of the recursive call. We recall that the (app) rule allows to add a multiple of the cost-free signature to the regular signature for typing the call. This is where the cost-free typings become relevant. We need to proof that a call to `fromList` preserves the potential $\log_2(|xs|)+1/2$, in the form $\log_2(|t'|)+1/2$. From the soundness theorem, we know that a cost-free typing $\Phi \vdash^{\mathrm{cf}} \Psi$ implies $\Phi \geq \Psi$. For `fromList` specifically, this means the size of the input list is greater than or equal to the size of the resulting tree. The cost-free derivation resembles an inductive proof of the given property over the structure of `fromList`, and enables, in this case, a "type cast" of the potential, from a list to a tree. For the time being we assume that we can derive such a typing and therefore conclude this branch. We will however return to the cost free derivation in the next part of this section.

$$\dfrac{}{x_4\colon \mathsf{T} \mid 1/2\,\mathsf{rk}(t')+2\log_2(|t|)+3/2 \vdash \texttt{insert x x4}\colon \mathsf{T}|1/2\,\mathsf{rk}(t')+1}\ (\mathsf{app})\qquad (5.4)$$

Finally the body of the previous let expression holds the call to `insert`. The potential aligns correctly with the corresponding signature, so the rule (app) can be applied directly.

5.2 Cost-Free Derivation

For the cost-free derivation of `fromList`, we have the same proof tree, but we apply the cost free variants of the typing rules. Note that this really only makes a difference for

the (tick) and (app) rules.

$$\dfrac{\dfrac{\dfrac{}{\varnothing_{\mathsf{T}} \mid 0 \vdash^{\mathrm{cf}} \texttt{error} : \mathsf{T} \mid \log_2(|t'|) + 1/2}\,(\mathsf{const})}{l : \mathsf{L} \mid \log_2(|l|) + 1/2, t : \mathsf{T} \mid 0 \vdash^{\mathrm{cf}} \texttt{error} : \mathsf{T} \mid \log_2(|t'|) + 1/2}\,(\mathsf{w} : \mathsf{var})^*}{\begin{array}{l} l : \mathsf{L} \mid \log_2(|l|) + 1/2, t : \mathsf{T} \mid 0 \\ \vdash^{\mathrm{cf}} \texttt{match}\ t\ \texttt{with} \mid \texttt{node}\ x_1\ \_\ x_2 \rightarrow\ e_1 \mid \texttt{leaf} \rightarrow\ e_2 \\ :\ \mathsf{T} \mid \log_2(|t'|) + 1/2 \end{array}}\,(\mathsf{match})\quad (5.5)$$

Again we start with an application of the match rule. The derivation of the `leaf` case follows the same principle as before.

$$\dfrac{\dfrac{\dfrac{}{\varnothing_{\mathsf{L}} \mid 0, \varnothing_{\mathsf{T}} \mid 1/2 \vdash^{\mathrm{cf}} \texttt{leaf} : \mathsf{T} \mid 1/2}\,(\mathsf{const})}{\varnothing_{\mathsf{L}} \mid 1/2, \varnothing_{\mathsf{T}} \mid 0 \vdash^{\mathrm{cf}} \texttt{leaf} : \mathsf{T} \mid 1/2}\,(\mathsf{w})\quad (5.6)}{\begin{array}{l} l : \mathsf{L} \mid \log_2(|l|) + 1/2, \varnothing \mid 0 \\ \vdash^{\mathrm{cf}} \texttt{match}\ l\ \texttt{with} \mid \texttt{nil} \rightarrow\ e_1 \mid \texttt{cons}\ x\ xs \rightarrow\ e_2 \\ :\ \mathsf{T} \mid \log_2(|t'|) + 1/2 \end{array}}\,(\mathsf{match})\quad (5.5)$$

Another application of (match) is required for the deconstruction of the list l. However, in this case, the weakening step in the `nil` case does not nullify the remaining constant potential of l. Instead, it is transferred to the constant tree potential. In the type system, this corresponds to a type conversion, enforced through inter-potential weakening, as described in Chapter 4. This step is clearly valid, as both potential functions represent the same constant, $1/2$. When viewing the derivation as an inductive proof, we have now shown, $|l| = |t'|$, for the `nil` case.

$$\dfrac{\dfrac{}{\varnothing_{\mathsf{T}} \mid 0 \vdash^{\mathrm{cf}} \texttt{leaf} : \mathsf{T} \mid 0}\,(\mathsf{const})\quad (5.7)}{l : \mathsf{L} \mid \log_2(|xs| + 1) + 1/2, \varnothing_{\mathsf{T}} \mid 0 \vdash^{\mathrm{cf}} \texttt{let}\ x_3 = \texttt{leaf}\ \texttt{in}\ e_3 : \mathsf{T} \mid \log_2(|t'|) + 1/2}\,(\mathsf{let})\quad (5.6)$$

For the `cons` case, we again end up with the shifted potential $\log_2(|xs| + 1)$ for the tail of the list. The binding of the let expression is typed trivially by (const), as $0 = 0$.

$$\dfrac{\dfrac{\dfrac{}{l:\mathsf{L} \mid \log_2(|xs|) + 1/2, x_3:\mathsf{T} \mid 0 \vdash^{\mathrm{cf}} \texttt{fromlist xs x3}:\mathsf{T}\mid \log_2(|t'|) + 1/2}\,(\mathsf{app})}{l:\mathsf{L} \mid \log_2(|xs|+1) + 1/2, x_3:\mathsf{T} \mid 0 \vdash^{\mathrm{cf}} \texttt{fromlist xs x3}:\mathsf{T}\mid \log_2(|t'|+1) + 1/2}\,(\mathsf{shiftm})}{\begin{array}{l} l:\mathsf{L} \mid \log_2(|xs|+1) + 1/2, x_3:\mathsf{T} \mid 0 \\ \vdash^{\mathrm{cf}} \texttt{let } x_4 = \texttt{fromlist xs x3 in insert x x4} \\ :\ \mathsf{T}\mid \log_2(|t'|) + 1/2 \end{array}}\,(\mathsf{let}) \tag{5.7}$$

Before examining the recursive call to `fromList`, we must consider a valid cost-free signature for `insert`. Intuitively, it makes sense that the size of the resulting tree is exactly one greater than the size of the input tree, and therefore $\log_2(|t|+1) \geq \log_2(|t'|)$. In fact there exists a corresponding a cost-free derivation for `insert`, which we will not discuss here. Unfortunately utilizing this signature results in an issue, illustrated by the following equation. It demonstrates that the type-free signatures of `fromList` and `insert` do not align, when composing the two functions.

$$\underbrace{\log_2(|xs|) \geq \log_2(|t|)}_{\texttt{fromList}} \neq \underbrace{\log_2(|t|+1) \geq \log_2(|t'|)}_{\texttt{insert}}$$

The issue can be solved by shifting the argument of the logarithms in the first inequality.

$$\log_2(|xs|+1) \geq \log_2(|t|+1) = \log_2(|t|+1) \geq \log_2(|t'|)$$

This would correspond to the signature $\log_2(|x|+1) \vdash^{\mathrm{cf}} \texttt{fromList} : \log_2(|t|+1)$, which contradicts our current assumption for this signature. We could in principle try and derive this new signature in addition the current one, as we can have an arbitrary number of cost-free signatures for any function in the analysis. However, this is problematic since we would need yet another one for that derivation and so on. Instead we can apply the (shiftm) rule just before the call to `fromList`. This allows to shift the arguments of a potential function consisting of a single monotone basic potential function by a constant. Since we have logarithmic terms on both sides, the preconditions for this rule hold, allowing us to derive the correct potential for the recursive call. When viewing this fact again from our inductive proof perspective, the application of (shiftm) enables us to exploit the specific properties of the logarithm to apply the induction hypothesis, i.e. the type of the recursive call.

$$\dfrac{}{x_4:\mathsf{T} \mid \log_2(|x_4|+1) + 1/2 \vdash^{\mathrm{cf}} \texttt{insert x x4}:\mathsf{T}\mid \log_2(|t'|) + 1/2}\,(\mathsf{app}) \tag{5.8}$$

The final call to `insert` is again trivial, as the correct potential is present.

This concludes the type derivation of `fromList`. Not only did we witness the necessity of a mixed potential analysis and techniques like inter-potential weakening, we also observed how potential is formally converted between different data types.

6 Implementation

In this chapter, we describe the prototype implementation of the analysis framework introduced in Chapter 4. At its core is a type inference algorithm for the abstract type system supporting mixed potentials. We refer to this new tool as atlas-2, as it represents an evolution of the existing tool, ATLAS. Rather than extending the previous Java implementation, we chose a complete reimplementation in Haskell. This decision enabled a more generic design from the outset and simplified the tool wherever possible, removing some of the experimental features present in the earlier version.

To describe the operation of this tool, we distinguish between the following stages:

1. *Frontend*: Parsing the input program into an abstract syntax tree (AST), inferring types, applying program transformations, and contextualizing the program.
2. *Resource Analysis*: Deriving types for resource annotations, generating and solving the constraint system, and computing amortized bounds.

At this level, the two stages are also evident in the original tool. The first stage is more generic and closely resembles the structure of a compiler, which is why we adopt the term "frontend stage". The second stage, however, is specific to our analysis and encapsulates a type inference algorithm that automates the potential analysis.

In the following, we revisit how these stages operate in principle, present our generalizations, and highlight some subtleties of the analysis that were left implicit in the previous description.

6.1 Frontend

This stage can again be divided into the following steps:

1. *Parsing*: Parse the input program into an AST.
2. *Type Inference*: Infer non-annotated types for every (sub-) expression.
3. *Program Transformation*: Transform the given program into let-normal form.
4. *Contextualization*: Add context clues for automated tactic generation.

All of the above mentioned steps operate on and extend the AST of the given program. In order support adding additional fields to our AST data structure at each stage, we implement the "tress that grow" pattern [32]. This means that we let every expression carry an annotation indexed by a type family. This annotation holds additional information,

like the position in the source or the inferred type of the expression. Type families, in Haskell, provide type indexed types and named functions on types [38]. Here we utilize them to define different expression annotations for every stage, i.e. the stage acts as the index for the type family.

Take as an example our abbreviated definition of expressions.

```
type family XExprAnn a

data Expr a
  = VarAnn (XExprAnn a) Id
    | ...
```

Every (sub-) expression is annotated with the family `XExprAnn a`. We then define a concrete AST type for every stage, and instantiate the annotation type. For example, every expression in the parsed AST is annotated with a source position, that stores the line and column number of the expression.

```
data Parsed
type ParsedExpr = Expr Parsed

-- type SourcePos = (Int, Int)
type instance XExprAnn Parsed = SourcePos
```

Similarly, we define a representation for typed expressions. This representation includes not only the source position information from parsing but also the inferred type of the expression. Here is an example of a typed expression:

```
data Typed
type Expr Typed

type instance XExprAnn Typed = TypedExprAnn {
  teSrc :: ExprSrc, -- Source position
  teType :: Type}   -- Inferred type
  deriving (Eq, Show)
```

By applying the "trees that grow" pattern, we enable the AST to evolve naturally through the stages of compilation (e.g., parsing, typing). Each stage adds its own specific annotations without affecting the core structure of the AST, making the implementation modular and extensible.

The first step is parsing the input program. We implement the abstract grammar described in Chapter 3 with the parser combinator library megaparsec [1]. Once parsing is complete, we infer the plain types for every expression in the program using a variant of Hindley-Milner type inference. These inferred types are stored in the AST and are crucial for determining resource annotations in later stages.

Next, the program is transformed into let-normal form (LNF), as required by the given semantics and type system. LNF is a variant of A-normal form (ANF) [8], which imposes restrictions on arguments in function applications: they must be variables, constants, or λ-terms. Any more complex expression must be decomposed and assigned to a variable by a let binding. Furthermore, as specified by the abstract grammar, we impose an additional restriction: only variables are allowed as function arguments. Take as an example the following expression.

[1] `https://github.com/mrkkrp/megaparsec`

```
node (∼ delete_min ta) a (node tb b tc)
```

It transforms into the following LNF.

```
let x0 = ∼(delete_min ta) in
  let x1 = (node tb b tc) in
    (node x0 a x1)
```

As the final step of the frontend stage, we add a set of context clues to every expression. While the details of automated tactic generation are covered in the next section, it is important to note that this process requires information about an expression's position within the AST. Specifically, it has to identify the parent of a given expression.

In the previous Java implementation, this was handled ad hoc using pointers within the AST data structure. However, in our functional implementation, we instead gather and store the necessary information in a single top-to-bottom traversal of the AST.

6.2 Resource Analysis

This section presents a concrete type inference algorithm for the type system presented in Chapter 4. The following pseudo code gives a high level description this algorithm.

Algorithm 1 Resource Analysis Algorithm

Input: program $\mathcal{P}$, analysis mode
Output: proof tree and coefficients or proof tree and unsat core
for each potential bearing data type in the analysis **do**
    create a fresh resource annotation representing a potential function Ψ
for each function f in program $\mathcal{P}$ **do**
    create a fresh annotated context as LHS for f's signature,
            and set RHS to the corresponding potential function
    create cost-free signatures for f.
    for each signature of f **do**
        generate constraints and proof tree via syntax-directed type derivation (†)
    add signature constraints based on the analysis mode
coefficients ← solve and optimize the constraints
if the constraints are satisfiable **then**
    return proof tree, coefficients, and compute bounds
else
    return proof tree and unsat core

The core of the algorithm is the subroutine (†). This subroutine generates constraints by following the rules of our type system and traversing the AST of the given function's definition. In line with the main soundness theorem, there exists a cost-free variant of this routine, which applies the corresponding cost-free versions of the type rules, when deriving a cost-free signature.

To ensure well-typedness, we require each function to have one signature with costs and any number of cost-free signatures. In practice, we limit the default number of

cost-free signatures to one. However, this limit can be increased for specific functions using annotations in the source code, e.g., `{-# NUMCF 2 #-}`.

A purely syntax-directed approach, however, is insufficient for deciding on the application of structural rules. For this reason, (†) relies on heuristics to generate tactics. Tactics provide a high-level outline of the type derivation, including all rule applications. Based on a tactic, a complete proof is then constructed.

As in the previous implementation, we support loading tactics from an external file for debugging purposes. However, the primary focus of the analysis is automated tactic generation, which we describe in the following section.

In addition to the constraints required by typing rules, we add *signature constraints*. These constraints allow us to incorporate type annotations from the source code into the analysis. How such annotations are linked to the coefficients of function signatures, depends on the chosen analysis mode. These modes are described in Section 6.2.2.

Note that the analysis needs to know which potential instance should be applied for each data type. We provide some defaults for this mapping, but also allow to specify it with source annotations on a per-file basis.

Example 6.1. The following listing gives an explicit type-potential mapping for the module `SplayTree`.

```
1  {-# POTENTIAL (Tree Base: logarithmic, List Base: linlog) #-}
```

6.2.1 Automated Tactic Generation

As mentioned before, the tactic generation, is mostly syntax directed, but requires heuristics for the application of structural rules. As observed by Leutgeb et al. [29], a sparse application of these rules, especially (w), is essential in keeping the resulting constraint system tractable. In particular they showed that applying (w), with all the available expert knowledge, at every possible point in the derivation, makes type inference infeasible. Through experimentation with hand-crafted tactics, they were able to derive common patterns in the application of these rules, which they implemented as heuristics for automation. We discovered that these existing techniques also suffice for our extended set of benchmarks.

At the heart of automated tactic generation lies a set of rules that determine, for each expression in the program, whether the (w) rule should be applied and what kind of expert knowledge should be used, based on the context clues provided. The different context clues are listed in the following.

- `pseudo leaf`: An expression that corresponds to a leaf in the resulting proof tree, but is not bound in a let expression.
- `binds (rec) app`: A let expression that binds a (recursive) function application, possible wrapped in a tick expression.
- `first after app`: First expression after a function application, i.e. the body of let expression binding a function application.

Table 6.1: Rules for introducing a weakening step.

Context Clues	Weakening Arguments
`pseudo leaf`	`mono`
`bind rec app`	`neg`
`bind app`	`mono`, `l2xy`
`first after app`, `outermost let`	`mono`, `l2xy`
`first after match`	`mono`
`ite coin`	`l2xy`

- `first after match`: First expression of a match arm.
- `outermost let`: Expression is the outermost let in a nested let expression.
- `ite coin`: If-then-else expression with a coin expression as condition.

Table 6.1 lists the rules that are applied to determine whether a weakening step for a certain expression is added to the tactic. The context clues serve as the preconditions and the consequence of each rule is a set of arguments for applying (w). These arguments correspond either to broader categories of expert knowledge (e.g., `mono` applies monotonicity if it is supported by the corresponding potential instance) or refer to specific lemmas (e.g., `l2xy` represents Lemma 4.8).

In addition to the rules for (w), (shift) is applied before every function application and (w : var) is used to eliminate redundant variables at the leafs of the derivation.

6.2.2 Analysis Modes

In principle when conducting an amortized analysis, we are interested in minimizing the the amortized costs. As was shown by Leutgeb et al. [29], this can be automated, by utilizing a constraint solver, capable of minimizing a cost-function that complements the linear constraint system produced by the analysis. The most prominent example for such a solver is z3 [7]. For this reason it also serves as the backend for our implementation. The optimization of an objective function that minimizes the amortized costs, forms the primary analysis mode, of both ATLAS and our tool. This objective function forms a set of constraints, involving the signature of the function under analysis. For the formulation of such constraints we define the symbolic cost of a signature consisting of LHS P and RHS P', as $Q = P - P'$. This difference is defined pointwise and requires a substitution of variables, that corresponds to the reasoning given in Chapter 2.

For our generalization, we define a cost function for each potential instance separately, as is the case with the (w) rule, expert knowledge about the basic potential functions is required to define such a function. For details about the cost function for logarithmic potential we refer the reader to [29]. For the polynomial potential instance and a given symbolic cost Q_f for signature of function f, we define the following simple objective function, which just penalizes the absolute value of the coefficients of the cost Q_f.

$$J_f = \sum_{q \in Q_f} q$$

For the log-linear potential we utilize a similar approach, that additionally penalizes more complex terms.

$$J_f = \sum_{q_{(l_i^{(a,b)},c)} \in Q_f} q_{(l_i^{(a,b)},c)} \cdot (a+b)$$

The described approach operates on the level of resource annotations, but is extend to annotated contexts by applying the corresponding cost functions per type and summing them up to obtain a compound cost function. Similarly we sum up the objectives for all function signatures in a program. We refer to this optimization based analysis mode as *full inference*. It is the most powerful mode, as it does not require any kind of type annotations and infers both a suitable potential function and the amortized costs.

In this work, we additionally introduce a less powerful variation of full inference, referred to as *improve costs*. This mode enables users to specify an upper bound on the amortized costs of a function as a type annotation. The tool then searches for amortized costs that are smaller than the provided bound. As with full inference, this process leverages optimization techniques to solve the constraints.

Example 6.2. Cost annotation for `SplayTree.splay`, corresponding to the cost $3/2 \log_2 |t|$.

```
1 splay :: (Base * Tree Base) → Tree Base @ Tree Base [(t^1) |→ 3/2]
```

Note that standard type annotations are supplemented by the cost annotation with the separator `@` and that our custom annotations are always grouped per type, to represent annotated contexts.

We believe this mode adds significant value, particularly when the user already has an intuition about the costs of a function but cannot define a suitable potential function and at the same time, running full inference is too expensive. In such cases, *improve costs* infers the potential function and provides guidance, helping steer the user toward a better understanding of the function's behavior. For completeness we also provide the mode *check costs*, which performs type checking for the exact given cost. In this case no optimization is needed, which makes a big difference in execution time.

Another addition in our reimplementation is the ability to mark a function for a worst case analysis. As we mentioned before we require every function that returns a certain type to be analyzed with the corresponding potential function, which is unique per analysis. This makes sure that we obtain consistent results when analyzing a complete program. There are however cases in which this is not possible. One example is the function `PairingHeap.merge`. Its definition does not contain any ticks as it does not require any recursive calls, therefore its amortized costs are clearly zero. In such a case the potential usually should just be preserved. For `PairingHeap.merge` this is not possible, however it is always valid to account for the worst case costs of function. In this case we set the potential function to zero, so it can not use up any potential for its computation, nor can it return any potential.

Example 6.3. We provide the following variant of cost annotations, that represent worst case costs.

```
1 merge :: (Tree Base * Tree Base) → Tree Base @> Tree Base []
```

In this case we have cost of 0.

Lastly we have the mode *check coefficients* that type checks the given coefficient values for the function signature. This option is mainly interesting while developing a new potential instance, but never the less crucial.

Example 6.4. Coefficient annotation for `SplayTree.splay`, including a cost free signature given in curly brackets.

```
1   splay :: (Base * Tree Base) → Tree Base
2         | Tree Base [t |→ 1/2, (2) |→ 1, (t^1) |→ 3/2]
3           → Tree Base [e1 |→ 1/2, (2) |→ 1]
4           {Tree Base [(t^1) |→ 1/2] → Tree Base [(e1^1) |→ 1/2]}
```

6.3 Evaluation

In the following section, we present experimental results obtained with our prototype implementation. To ensure that our generalized implementation, denoted atlas-2 in the following, serves as a faithful reimplementation of the original tool ATLAS, it should reproduce the previous results. To this end, we distinguish between the deterministic benchmarks [29] and probabilistic benchmarks [30], demonstrating that our approach either matches or improves upon the previous bounds while maintaining comparable performance, particularly in terms of execution time.

The results for deterministic examples are shown in Table 6.3. They include the execution times for both type checking and inference, corresponding to the check coefficients and full inference modes described above. As the two tools slightly diverge in how the analysis is conducted, we give the command line flags applied for this benchmark in Table 6.2. To begin with, we observe that both tools produce the same bounds, for splay trees and splay heaps. We were able to improve on the constants of `del_min`, `insert` and `merge` for pairing heaps, see Table 6.4, by introducing a refinement of the monotonicity expert knowledge. For our new benchmark `SplayTree.fromList`, we observe that inference is faster than for `SplayTree.insert`. This is counter intuitive, since `fromList`, depends on the analysis of `insert`. We suspect that the additional context, i.e. how the function is called, in particular the constraints for its cost free signature, improve the solver performance for `insert`.

When it comes to execution times, neither of the two tools outperforms the other across all benchmarks; their times generally are of a similar scale. We therefore conclude that the tools have comparable performance. In general it should be noted that the main bottleneck, as with the previous version, is the time spent by the constraint solver.

For the probabilistic benchmarks we arrive at similar results, as shown in Table 6.5, since both implementations, again produce the exact same bounds with similar execution

Table 6.2: The command line flags for ATLAS and atlas-2 used to benchmark type checking and inference. The Annotations column indicates whether resource annotations were provided in the source file.

Mode	ATLAS	atlas-2	Annotations
check coefficients	-	`--analysis-mode check-coeffs`	no
full inference	`--infer --equal-ranks=true`	`--analysis-mode infer`	no

times. For these examples we omit inference times, as both implementations take more than 24 hours to complete.

Lastly, we compare our analysis of the functional queue with RaML. We clarify upfront that while we do compare to RaML, we limit this comparison to a specific example that, according to the literature, requires amortized analysis. We do not aim to match RaML across its diverse set of benchmarks, which rely on different advanced analysis features. Although we modeled the potential function required for this example after the multivariate polynomial potential function for lists employed in RaML, the latter does not perform an amortized analysis. As a result, its bound cannot be smaller than $O(n)$. The derived bounds from both approaches are compared in the following table.

Function	RaML[10]	atlas-2
`moveToFront` (`copyover`)	n	0
`snoc` (`enqueue`)	1	2
`tail` (`dequeue`)	$2n + 1$	1

For RaML we used the "Functional queues and BFS" example provided in the RaML's web interface [2]. In this benchmark we configured "ticks" as metric, a degree of two and the "upper bounds" mode. In the results we see our thesis confirmed, as the bound for `tail` is linear for RaML, whereas our amortized analysis, gives the expected constant bound.

Further we implemented the classical example of `MultiPopStack`. This is a stack that supports the operation `push`, to add an element to the stack and an operation `pop`, that takes, in addition to the stack, an arguement k and pops k elements from stack, as long as k is at most the size of the stack. An amortized analysis allows to derive constant bounds for both operations. In particular our analysis correctly chooses the size of the stack as the potential function and gives a bound of 0 for `pop` and 1 for `push`.

[2] `https://www.raml.co/interface/`

Table 6.3: Comparison of execution times of type checking / inference, for the deterministic self-adjusting data structures `SplayTree` (`ST`), `SplayHeap` (`SH`) and `PairingHeap` (`PH`). Analysis of a function includes the analysis of its dependencies and t/o denotes a time out.

		ATLAS[29],[30]	atlas-2
Function	Bound	Time(s) (h:mm:ss)	Time(s) (h:mm:ss)
`ST.splay`	$3/2 \log_2(\|t\|)$	27.57 / 10:56.37	8.538 / 9:04.13
`ST.splay_max`	$3/2 \log_2(\|t\|)$	5.58 / 36.43	1.057 / 18.17
`ST.insert`	$2 \log_2(\|t\|) + 1/2$	26.89 / 19:45.55	17.27 / 26:13.68
`ST.delete`	$5/2 \log_2(\|t\|) + 2$	26.62 / 1:26:22	9.306 / 41:20.96
`ST.fromList`	$2\|l\| \log_2(\|l\|) + 1/2\|l\|$	- / -	9.21 / 9:04.25
`ST.*`	n.a.	42.90 / t/o	13.93 / 36:29.05
`SH.partition`	$1/2 \log_2(\|t\|)$ $+ \log_2(\|t\| + 1)$	25.05 / 13:00.61	9.874 / 14:50.02
`SH.insert`	$1/2 \log_2(\|t\|)$ $+ \log_2(\|t\|+1)+3/2$	22.63 / 16:42.67	11.12 / 20:02.31
`SH.delete_min`	$\log_2(\|t\|)$	1.56 / 11.83	0.62 / 1:23.59
`SH.*`	n.a.	20.99 / 16:48.27	12.30 / 28:23.95

Table 6.4: Comparison of the amortized complexity bounds for the operations of `PairingHeap` (`PH`) and the execution times of type checking / inference. Analysis of a function includes the analysis of its dependencies.

	ATLAS[29],[30]	
Function	Bound	Time(s) (h:mm:ss)
`PH.merge_pairs`	$3/2 \log_2(\|h\|)$	10.85 / 20.886
`PH.del_min`	$\log_2(\|h\|) + 1$	10.5 / 47.39
`PH.insert`	$1/2 \log_2(\|h\| + 1$	10.85 / 0.88
`PH.merge`	$1/2 \log_2(\|h_1\| + \|h_2\|) + 1$	3.08 / 2.843
`PH.*`	n.a.	11.72 / 1:10.89
	atlas-2	
	$3/2 \log_2(\|h\|)$	8.926 / 14.307
	$\log_2(\|h\|) + 1/2$	11.407 / 22.367
	$1/2 \log_2(\|h\|) + 1/2$	0.385 / 0.365
	$1/2 \log_2(\|h_1\| + \|h_2\|)$	0.855 / 0.916
	n.a	14.63 / 1:21.68

Table 6.5: Comparison of the execution times for type checking the probabilistic variants of `SplayTree` (`ST`) and `SplayHeap` (`SH`), as well as `MeldableHeap` (`MH`) and `CoinSearchTree` (`CST`). Analysis of a function includes the analysis of its dependencies.

		[29],[30] ATLAS	atlas-2
Function	Bound	Time(s) (mm:ss)	Time(s) (mm:ss)
`ST.insert`	$^3/_4 \log_2(\lvert t\rvert)$ $+ {}^3/_4 \log_2(\lvert t\rvert + 1)$	2:18.97	1:49.05
`ST.delete`	$^3/_4 \log_2(\lvert t\rvert)$ $+ {}^3/_4 \log_2(\lvert t\rvert + 1)$	2:06.75	1:28.03
`ST.splay`	$^9/_8 \log_2(\lvert t\rvert)$	1:29.49	46.22
`ST.*`	n.a.	3:05.91	6:17.64
`SH.insert`	$^3/_4 \log_2(\lvert t\rvert)$ $+ {}^3/_4 \log_2(\lvert t\rvert + 1)$	1:59.65	52.63
`SH.delete_min`	$^3/_4 \log_2(\lvert t\rvert)$	3.484	0.527
`SH.*`	n.a.	1:48.72	1:26.08
`MH.insert`	$\log_2 \lvert h\rvert + 1$	12.18	2.68
`MH.delete_min`	$2 \log_2 \lvert h\rvert$	11.57	2.97
`MH.meld`	$\log_2 \lvert h\rvert + 1$	11.39	2.64
`MH.*`	n.a.	8.89	2.976
`CST.insert`	$^3/_2 \log_2 \lvert t\rvert + {}^1/_2$	0.88	0.30
`CST.delete`	$^3/_2 \log_2 \lvert t\rvert + 1$	3.92	1.15
`CST.delete_max`	$^3/_2 \log_2 \lvert t\rvert$	4.24	0.94
`CST.*`	n.a.	3.98	1.47

7 State of the Art and Related Work

Our work builds upon the well-established research tradition in automatic amortized resource analysis (AARA), a field pioneered by Hofmann et al. and recently recapitulated in Hoffmann's retrospective study [12].

This research tradition originated in the Hofmann's interest in the field of implicit computation (ICC). ICC, a popular research topic, at the time, aims to characterize computational classes implicitly with program languages. Hofmann in particular was able to show that the class PTIME corresponds to higher-order programs with structural recursion, when the computation is *non-size-increasing* [12]. He showed that this property could be enforced with local type rules, forming a linear type system [15, 16].

In his 2000 article Hofmann presents the first connection of these theoretical results with practical programs. In this work, considered by Hoffmann as the start of this research discipline, linear typing is used to obtain functional programs which can be compiled into "malloc-free C code", i.e. modify heap allocated-data structures in place. In such programs all the heap space needed must be provided to a function via its arguments, which means that a bound on the required heap space could be read of the type signature. The next advancement from this point was to infer these types and was achieved by Jost in his diploma thesis [23]. Subsequently, they were able to move from the $\diamond$ type, that represents a singular heap space, to natural numbers, which made the inference less cumbersome. Finally with their 2003 POPL article they introduced constant AARA [17], as a method for determining heap-space bounds in first-order functional programs.

Over the course of the following two decades this initial idea was extended in various directions, one of them being the addition of different language features, including higher-order functions [24]. The analysis has also been adapted for object oriented languages. Hofmann et al. [18] first presented a type system for heap space analysis of Java-like programs, successfully treating the issues of inheritance, downcast, update and aliasing. Later more general subtyping, sharing relations and an efficient type checking algorithm were added [22]. Finally type inference was achieved by Bauer et al. [2], by introducing a decidable procedure for solving linear tree constraints.

There has also been efforts to analyze lazily evaluated functional programs with this method. Simões et al. [40], for the first, time were able to define an automated analysis extending the Hofmann and Jost type system. They base their cost model on a version of Launchbury's natural semantics [28, 39] and introduce they notion of lazy potential, to deal with the evaluation of thunks. Further Vasconcelos et al. [45], introduced automated type inference for co-recursion of lazy functional languages.

Hoffmann et al. [13] also introduced an extension for analyzing the parallel evaluation of functional programs. Their approach separates the type derivation into two distinct components: one for the work, representing the sequential evaluation time, and another

for the depth, which corresponds to the parallel evaluation time.

Atkey et al. [1] translated AARA from the functional setting to programs with side effects by extending separation logic with suitable representation of consumable resources. This extended logic allows to define constraints both on heap shape and the consumption of resources. Additionally they define a restricted subset of this logic that lends itself to efficient proof search and inference of resource annotations. Similarly Hoare logic can be modified to support reasoning about resource bounds. [3, 4]. Carbonneaux et al. apply a quantified version of Hoare logic, in which predicates map states of non-negative numbers, representing potential. Here a Hoare triple represents the same resource invariant as a type judgment in the Hofmann and Jost system.

With the Absynth prototype, Ngo et al. [33] presented the first automated expected resource analysis of imperative probabilistic programs. They combine reasoning based on the weakest-precondition calculus with the automation techniques of AARA. Wang et al. [46] in term provided a similar analysis, based on AARA, for probabilistic functional programs. Leutgeb et al. [30] also provide an expected cost analysis, which in contrast to the previous example, derives amortized logarithmic bounds.

Das et al. [6] adapt AARA for concurrent message-passing programs, by introducing binary session types into the analysis. A similar technique is applied in a later work which presents a programming language for digital contracts, which includes the capability of controlling resource usage [5].

One major and often stated, limitation of original analysis of Hofmann and Jost is the restriction to linear resource bounds. Overcoming this limitation has inspired a number of works. The first extension of this kind was univariate polynomial potential and was proposed by Hoffmann et al. [11]. The work demonstrated the counter-intuitive fact that these non-linear bounds, can be produced, without requiring a non-linear constraint system. Subsequently Hoffmann et al.[9] introduced multivariate polynomial potential, to derive bounds of the form $m \cdot n$, which with univariate potential had to approximated by $m^2 + n^2$. AARA was also extended to exponential potential by Kahn and Hoffmann [26].

The line of research applying AARA to splay trees and other self-adjusting data structures [19, 29], which forms the foundation of this thesis, contrasts with the works mentioned above by focusing not on worst-case or best-case bounds, but rather on amortized cost bounds. Moreover, this approach leverages non-linear potential functions to derive the logarithmic bounds required for analyzing these data structures. Compared to other approaches [17, 9], their method is specifically tailored to the given use case rather than aiming to provide a general analysis. This is evident in their choice of potential function, which draws inspiration from previous pen-and-paper proofs [36], rather than relying on a universal function designed for broad applicability.

Our method follows the same line of thought while introducing a level of generalization that allows the analysis to be modular in its choice of potential function, thereby lifting the previous restriction to tree data types. At same time we preserve the use case specific reasoning, enabled by linearization of expert knowledge. Our work can be seen as a strict extension of the probabilistic analysis by Leutgeb et al. [30] that adds multivariate polynomial potential [9], and univariate log-linear potential for an amortized cost analysis.

8 Conclusion

In this thesis we introduced a potential function agnostic framework that generalizes the automated expected amortized cost analysis by Leutgeb et al. [29, 30]. The framework can be instantiated easily with data type specific constraints and expert knowledge. We demonstrated this technique on the logarithmic potential function devised for the analysis of splay trees [19, 29], and on a restricted version of the polynomial potential introduced by Hoffmann et al. [11, 9]. In addition to the adaptation of these existing potential functions to our framework, we provided a novel new potential function for AARA, that allows us to express log-linear resource bounds.

Additionally we introduced the novel concept of mixed potentials, which enables the combination of different potential functions within a single analysis. This new method allowed us to address the outstanding challenge of automatically deriving the amortized cost of inserting a sequence of items into a splay tree by combining a log-linear potential for lists with the existing logarithmic potential for trees.

We presented atlas-2—the successor to ATLAS—as a prototype implementation developed in Haskell. Our experimental results demonstrated that our tool is a strict extension of the work by Leutgeb et al. [29, 30], since we were able to derive the same or improved bounds for both their deterministic and probabilistic benchmarks, all while maintaining comparable runtime. By leveraging our polynomial potential instance for lists and the amortization capabilities of our framework, we improved upon the bounds for a purely functional queue established by the tool RaML, achieving the well-known optimal bounds for the first time.

We believe that this approach to automated amortized complexity analysis—while not universally applicable—offers a high degree of specialization tailored to specific use cases. This specialization enhances automated reasoning techniques, bringing them closer to the effectiveness of manual analysis. By modularizing different potential functions, we aim to accelerate future research in this area. More specifically, our framework could serve as a foundation for analyzing more complex data structures, such as skew heaps [43, 27] and other variations of leftist heaps [37].

Bibliography

[1] R. Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming*, pages 85–103. Springer, 2010.

[2] S. Bauer, S. Jost, and M. Hofmann. Decidable inequalities over infinite trees. In *LPAR*, pages 111–130, 2018.

[3] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for c programs. *ACM SIGPLAN Notices*, 49(6):270–281, 2014.

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

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

[6] A. Das, J. Hoffmann, and F. 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.

[7] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM Sigplan Notices*, 28(6):237–247, 1993.

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

[10] J. Hoffmann, K. Aehlig, and M. 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.

[11] J. Hoffmann and M. 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.

[12] J. Hoffmann and S. Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, 32(6):729–759, 2022.

[13] J. Hoffmann and Z. 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.

[14] M. 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.

[15] M. Hofmann. The strength of non-size increasing computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 260–269, 2002.

[16] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices*, 38(1):185–197, 2003.

[18] M. Hofmann and S. 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.

[19] M. Hofmann, L. Leutgeb, D. Obwaller, G. Moser, and F. Zuleger. Type-based analysis of logarithmic amortised complexity. *Mathematical Structures in Computer Science*, 32(6):794–826, 2022.

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

[21] M. Hofmann and G. Moser. Analysis of logarithmic amortised complexity. *arXiv preprint arXiv:1807.08242*, 2018.

[22] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Computer Science Logic: 23rd international Workshop, CSL 2009,*

18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings 23, pages 317–331. Springer, 2009.

[23] S. Jost. Static prediction of dynamic space usage of linear functional programs. *Master's thesis, Technische Universität Darmstadt, Fachbereich Mathematik*, 2002.

[24] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–236, 2010.

[25] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2*, pages 354–369. Springer, 2009.

[26] D. M. Kahn and J. 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.

[27] A. Kaldewaij and B. Schoenmakers. The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, 37(5):265–271, 1991.

[28] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, 1993.

[29] L. Leutgeb, G. Moser, and F. Zuleger. Atlas: automated amortised complexity analysis of self-adjusting data structures. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*, pages 99–122. Springer, 2021.

[30] L. Leutgeb, G. Moser, and F. Zuleger. Automated expected amortised cost analysis of probabilistic data structures. In *International Conference on Computer Aided Verification*, pages 70–91. Springer, 2022.

[31] G. Moser and M. Schneckenreither. Automated amortised resource analysis for term rewrite systems. *Science of Computer Programming*, 185:102306, 2020.

[32] S. Najd and S. P. Jones. Trees that grow. *arXiv preprint arXiv:1610.04799*, 2016.

[33] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: resource analysis for probabilistic programs. *ACM SIGPLAN Notices*, 53(4):496–512, 2018.

[34] T. Nipkow and H. Brinkop. Amortized complexity verified. *Journal of Automated Reasoning*, 62:367–391, 2019.

[35] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[36] B. Schoenmakers. A systematic analysis of splaying. *Information processing letters*, 45(1):41–50, 1993.

[37] B. Schoenmakers. Amortized analysis of leftist heaps. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part I*, pages 73–84. Springer, 2024.

[38] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008.

[39] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[40] H. Simoes, P. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 165–176, 2012.

[41] H. M. O. R. Simoes. *Amortised resource analysis for lazy functional programs*. PhD thesis, Universidade do Porto (Portugal), 2014.

[42] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[43] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[44] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[45] P. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. 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 787–811. Springer, 2015.

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