Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis

Jan Hoffmann

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München

Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis

Jan Hoffmann

Dissertation

Fakultät für Mathematik, Informatik und Statistik
Ludwig-Maximilians-Universität München

Advisor

Prof. Martin Hofmann, Ph.D.
Ludwig-Maximilians-Universität München

Reviewers

Nick Benton, Ph.D.
Microsoft Research

Prof. Zhong Shao, Ph.D
Yale University

Submitted on August 4, 2011

Disputation on October 14, 2011

This work was set in Utopia Regular with Fourier math fonts using the LaTeX document preparation system.

Impressum

Druck und Verlag: epubli GmbH, Berlin, www.epubli.de
ISBN 978-3-8442-1516-8

Abstract

A primary feature of a computer program is its quantitative performance characteristics: the amount of resources such as time, memory, and power the program needs to perform its task. Concrete resource bounds for specific hardware have many important applications in software development but their manual determination is tedious and error-prone.

This dissertation studies the problem of automatically determining concrete worst-case bounds on the quantitative resource consumption of functional programs.

Traditionally, automatic resource analyses are based on recurrence relations. The difficulty of both extracting and solving recurrence relations has led to the development of type-based resource analyses that are compositional, modular, and formally verifiable. However, existing automatic analyses based on amortization or sized types can only compute bounds that are linear in the sizes of the arguments of a function.

This work presents a novel type system that derives polynomial resource bounds from first-order functional programs. As pioneered by Hofmann and Jost for linear bounds, it relies on the potential method of amortized analysis. Types are annotated with multivariate resource polynomials, a rich class of functions that generalize non-negative linear combinations of binomial coefficients. The main theorem states that type derivations establish resource bounds that are sound with respect to the resource-consumption of programs which is formalized by a big-step operational semantics.

Simple local type rules allow for an efficient inference algorithm for the type annotations which relies on linear constraint solving only. This gives rise to an analysis system that is fully automatic if a maximal degree of the bounding polynomials is given. The analysis is generic in the resource of interest and can derive bounds on time and space usage. The bounds are naturally closed under composition and eventually summarized in closed, easily understood formulas.

The practicability of this automatic amortized analysis is verified with a publicly available implementation and a reproducible experimental evaluation. The experiments with a wide range of examples from functional programming show that the inference of the bounds only takes a couple of seconds in most cases. The derived heap-space and evaluation-step bounds are compared with the measured worst-case behavior of the programs. Most bounds are asymptotically tight, and the constant factors are close or even identical to the optimal ones.

For the first time we are able to automatically and precisely analyze the resource consumption of involved programs such as quick sort for lists of lists, longest common subsequence via dynamic programming, and multiplication of a list of matrices with different, fitting dimensions.

Zusammenfassung

Eine der wichtigsten Eigenschaften eines Programms ist sein Ressourcenverbrauch, die Menge an Ressourcen wie Zeit, Speicher und Energie, die das Programm bei seiner Ausführung benötigt. Konkrete Ressourcenschranken für individuelle Hardware haben wichtige Anwendungen in der Softwareentwicklung. Die manuelle Bestimmung solcher Schranken ist jedoch aufwendig und fehleranfällig.

Diese Dissertation behandelt die automatische Bestimmung konkreter Schranken an den Ressourcenverbrauch von funktionalen Programmen.

Traditionell basieren automatische Methoden zur Ermittlung des Ressourcenverbrauchs auf Rekurrenzgleichungen. Die technischen Schwierigkeiten beim Ermitteln und Lösen von Rekurrenzgleichungen haben zur Entwicklung von typbasierten Methoden zur Ressourcenanalyse geführt, die formal verifizierbar sind sowie über ein hohes Maß an Kompositionalität und Modularität verfügen. Bestehende Ansätze, die auf Amortisierung oder Sized Types basieren, können jedoch lediglich Schranken berechnen, die linear in der Größe der Funktionsargumente sind.

Diese Arbeit präsentiert ein neuartiges Typsystem, das polynomielle Ressourcenschranken für erststufige funktionale Programme herleitet. Wie ein von Hofmann und Jost vorgeschlagenes System für lineare Schranken, beruht es auf der Potentialmethode und amortisierter Analyse. Typen werden mit multivariaten Ressourcenpolynomen annotiert, einer Klasse von Funktionen, die nichtnegative Linearkombinationen von Binomialkoeffizienten verallgemeinern. Der Hauptsatz der Arbeit besagt, dass Typherleitungen Schranken beweisen, die korrekt sind im Bezug auf den Ressourcenverbrauch, der durch eine operationale Semantik formalisiert ist.

Einfache, lokale Typregeln eröffnen die Möglichkeit eines effizienten Inferenzalgorithmus für die Typannotationen , der ausschließlich auf linearer Optimierung beruht. Dies führt zu einer Analysemethode, die vollkommen automatisch ist, falls der Grad der Polynome beschränkt ist. Die Analyse kann mit zahlreichen Ressourcenmetriken parametrisiert werden und ermittelt beispielsweise Schranken an den Zeit- und Speicherverbrauch. Die Schranken sind abgeschlossen unter Komposition und werden am Ende der Analyse in geschlossenen, leicht zu verstehenden Formeln zusammengefasst.

Eine frei verfügbare Implementierung und eine reproduzierbare experimentelle Auswertung belegen die Praxistauglichkeit dieser automatischen amortisierten Analyse. Die Experimente mit einer Vielzahl von funktionalen Programmen zeigen, dass die

Berechnung der Schranken in vielen Fällen nur wenige Sekunden dauert. Die hergeleiteten Schranken an den dynamischen Speicher und die Anzahl der Auswertungsschritte wurden mit dem gemessenen maximalen Ressourcenverbrauch der Programme verglichen. Die meisten Schranken sind asymptotisch exakt und die konstanten Faktoren liegen dicht an den optimalen Faktoren oder entsprechen diesen sogar.

Zum ersten Mal sind wir in der Lage komplexe Programme vollständig automatisch zu analysieren: Die Analyse liefert beispielsweise präzise Schranken für Quicksort für Listen von Listen, für die Berechnung der längsten gemeinsamen Teilfolge mit dynamischer Programmierung und für die Multiplikation einer Liste von Matrizen mit unterschiedlichen Dimensionen.

Contents

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

DOUGLAS HOFSTADTER
Gödel, Escher, Bach: an Eternal Golden Braid (1979)

Preface

Writing a doctoral dissertation differs in several ways from writing research papers for conferences and journals. For one thing, your work is not aimed at a specific audience such as the attendees of a particular conference. For another thing, you do not directly compete with other papers and you have an unlimited number of pages at your command. As a result, you enjoy unusual liberties.

My intention is to use these liberties in this dissertation to make my work more accessible to non-experts. At the same time, I try to keep the text short and concise. If you are a researcher, you shall be able to quickly get an inkling of the basic ideas and concepts to decide if they are relevant for your own work. If you are a student, you shall find enough explanations to fully understand the technical details.

In any case, I hope to convey some of the excitement and pleasure I had during the work on my thesis.

Content and Structure

My dissertation deals with the problem of *automatic quantitative resource analysis:* Given a program P; automatically compute a bound on the resource consumption of P as a function of the sizes of its inputs.

A *resource* can be every quantity that is consumed by a program during its execution by a computer. This includes time, memory, and power, but also more specific quantities such as data exchange over a network or the number calls to a particular system function.

Automatic quantitative analysis of algorithms is a non-trivial problem which has been the subject of extensive research. In this work, I follow a line of research that is known as *automatic amortized resource analysis*. In a nutshell, I present an analysis that automatically computes *polynomial resource bounds for first-order functional programs*.

The main concepts I use are functional programming, type systems, big-step operational semantics, linear programming, and basic mathematics. If you are not familiar with these concepts or if you struggle with some of the imperfect explanations in my thesis then you find excellent guidance in the books *Types and Programming Languages* [Pie02], *Concrete Mathematics* [GKP94], and *Introduction to Linear Optimization* [BT97].

I describe novel results in Chapters 3, 5, 6, and 7. In Chapters 1, 2, 4, and 9, I explain and summarize the results and relate them to existing research.

Chapter 1 introduces in detail the area of research. It formulates the problem of (quantitative) resource analysis. It describes applications of resource analysis, difficulties of manual analysis, and the need of automatic methods. I also discuss the theoretical limitations of automatic resource analysis systems and the problems you face in designing them. Finally, I give a high-level description of the contents of this dissertation and informally explain the achievements of my work.

Chapter 2 introduces *amortized resource analysis* and outlines the technical contributions of this thesis. I first explain the idea of manual amortized analysis and show how it can be automated to statically predict the resource consumption of programs. I then informally present the main innovation of my work, the first automatic amortized analysis that derives *polynomial* resource bounds. Finally, I summarize the technical contributions of this thesis.

Chapter 3 presents *Resource Aware ML (RAML)*, a first-order fragment of the functional programming language SML. RAML programs are the objects that I study in my dissertation. I define their syntax and state the reasons behind my design decisions. To reason about resource consumption of RAML programs, I introduce a big-step operational semantics that formalizes terminating and non-terminating evaluations. It is parametric in the resource of interest and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. I use it later to prove the soundness of the analysis system.

Chapters 4, 5, and 6 formally describe automatic amortized resource analysis systems. I present them in form of three type systems for RAML. In Chapter 4, I recapitulate the automatic amortized analysis introduced by Hofmann and Jost. This system is able to derive *linear* resource bounds. Thereafter, Chapter 5 and Chapter 6 describe my main contributions, that is, automatic amortized resource analyses that compute *polynomial* resource bounds. More precisely, Chapter 5 presents a type system that is able to derive resource bounds that are sums of *univariate* polynomials, functions such as $3+5n^2+m$. Chapter 6 contains a type system that also computes *multivariate* polynomial resource bounds as, for instance, $10n^2+5nm$.

The two polynomial type systems extend the respective preceding type system. However, Chapter 5 does not depend on Chapter 4. Similarly, Chapter 6 does not depend on Chapter 4 and Chapter 5. In fact, I included Chapters 4 and 5 for didactic reasons only. Each chapter is devoted to a different purpose. Chapter 4 explains the general idea of automatic amortized analysis. Chapter 5 shows how you can use automatic amortized analysis to derive super-linear bounds. Finally, Chapter 6 describes how amortized analysis can take into account relations between different parts of the input. So if you are familiar with linear amortized analysis then you can skip Chapter 4 and start with Chapter 5. Similarly, if you are an expert in the field, you can skip Chapter 4 and Chapter 5, and directly read Chapter 6.

Chapter 7 presents the experimental evaluation of the analysis system. Klaus Aehlig and I jointly implemented the multivariate analysis system from Chapter 6 using the programming language Haskell and the Glasgow Haskell Compiler. I briefly describe

the implementation, report the running times of the analysis on standard desktop computers, and compare the computed bounds with the measured worst-case behavior of several example programs.

In Chapter 8, I give an overview of existing approaches to automatic resource analysis and relate my work to similar research.

Finally, Chapter 9 summarizes the results and states possible future research directions.

Acknowledgments

This work results from countless insightful discussions that I had with my bright and inspiring colleagues from LMU and TU Munich.

Martin Hofmann was an unerring source for sorting out the good ideas from the, say, not so good ideas. He always pushed for simpler and more general solutions, especially at times in which I was prematurely satisfied with my work. I profited greatly from his broad knowledge and his ability to quickly understand, narrow down, and solve problems.

Klaus Aehlig coauthored the paper on multivariate amortized resource analysis [HAH11] and also influenced my earlier work. He proved theorems and wrote Haskell code in a high-level way that seems to be reserved for genuine mathematicians. Max Jakob set up the server for the presentation of the prototype implementation on the web.

Helmut Seidl supported me constantly with advice and valuable suggestions. I am profoundly indebted to Robert Grabowski who shared a noisy office with me for several years. Ulrich Schöpp patiently answered my frequent questions on type theory, category theory, and functional programming.

I was lucky enough to be able to discuss my work with Andreas Abel, Nick Benton, Lennart Beringer, Andreas Gaiser, Jan Johannsen, Steffen Jost, Andrew Kennedy, Martin Lange, Markus Latte, Luke Ong, Dulma Rodriguez, Zhong Shao, and many others.

I thank you all.

Funding Acknowledgment

I wrote this dissertation as a scholar of the DFG Graduiertenkolleg 1480 (PUMA).

We often are faced with several algorithms for the same problem, and we must decide which is best. This leads us to the extremely interesting and all-important field of algorithmic analysis: Given an algorithm, we want to determine its performance characteristics.

DONALD KNUTH
The Art of Computer Programming Vol. 1 (1968)

1 Introduction

The analysis of the quantitative resource behavior of algorithms and programs is a classic domain of computer science. In Section 1.1, I explain the term and the reasons why it is an important problem in software development.

The identification of a problem and the desire to solve it automatically with a computer usually goes hand in hand in computer science. Quantitative resource analysis is not an exception. In Section 1.2, I describe why automatic methods for quantitative resource analysis are desirable and investigate the theoretical possibilities and limitations of automatic resource analyses of programs. Finally, I give a survey of the research on automatic resource analysis.

In Section 1.3, I state the goals of my research and outline the contents of my dissertation. I then describe the achievements of my work in a non-technical way.

1.1 Quantitative Resource Analysis

The *(quantitative) analysis of algorithms* has been described in a great number of textbooks and is studied by most computer-science students in undergraduate courses. According to the popular textbook *Introduction to Algorithms* [CSRL01] "analyzing an algorithm has come to mean predicting the resources that the algorithm requires".

The need of quantitative analysis of algorithms naturally arises when you program a computer.

- You have to compare the efficiency of different algorithms for the same problem to decide which one you should implement.
- You have to take into account the complexity of algorithms to design efficient programs.

- You have to find bottlenecks in a present software system to improve its performance.

Sometimes we want to determine the behavior of an algorithm in the *average-case* with respect to some distribution over the input. Most often—like in this work—we are however interested in the *worst-case* behavior of algorithms. The reason is that a worst-case bound guaranties a certain resource consumption for every input.

Quantitative analysis is a non-trivial problem. It may require sophisticated mathematics and is often challenging even for experts. The result of the analysis has to be summarized in closed expressions, which usually involve the sizes of the inputs.

Asymptotic Behavior

The analysis of an algorithm is only meaningful with respect to a machine model that describes how algorithms are executed. Many papers and textbooks, like *Introduction to Algorithms*, use informal machine models to abstract from implementation details and to make the analysis "as machine-independent as possible" [CSRL01]. The analyses then usually focus on the *asymptotic* resource behavior of algorithms.

To give an impression of such an analysis, I sketch the determination of the asymptotic worst-case behavior of the classic *insertion sort* algorithm as described in *Introduction to Algorithms*. In this book, Insertion sort is specified in pseudo code as follows.

```
Insertion-Sort(A)                          cost          times
  for j ← 2 to length[A]                    c1            n
      do key ← A[j]                         c2            n-1
         (*Insert A[j]*)                    c3            n-1
         i ← j - 1                          c4            n-1
         while i > 0 and A[i] > key         c5            Σ_{j=2}^{n} t_j
             do A[i+1] ← A[i]               c6            Σ_{j=2}^{n} (j-1)
                i ← i - 1                   c7            Σ_{j=2}^{n} (j-1)
         A[i+1] ← key                       c8            n-1
```

Like in the textbook, we assume a machine that consumes a constant amount of resources c_i in line i of the preceding code. If A is an array of length n then the last column states the number of times each line is executed in the worst-case, that is, when the array is in reverse sorted order. We use the identity $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ to summarize the worst-case resource consumption $T(n)$ of *Insertion−Sort(A)* as

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right) + (c_6 + c_7)\frac{n(n-1)}{2} \\ &= \left(\frac{c_5 + c_6 + c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

The values of the constants c_i depend on both the resource of interest and the actual implementation in a computer. For the running-time of the algorithm we assume that

$c_i > 0$ for $i \neq 3$ (comments do not influence the running-time and thus $c_3 = 0$). Then the quadratic term in the formula is dominating. We therefore say that *Insertion−Sort* has a quadratic running-time and write $T(n) = O(n^2)$.

Precise Bounds

An abstract, informal machine model may be favorable to convey algorithmic ideas and to analyze asymptotic behavior. However, it can lead to subtle problems and to disagreements on how to account for certain operations in the analysis. In some cases it is clearly problematic:

- It is sometimes hard to compare algorithms that have the same asymptotic behavior.
- You can not directly determine a concrete number that bounds the resource consumption for a given input.
- The asymptotic behavior is not meaningful if you are interested in *small inputs*.

To rigorously argue about the number of steps that an algorithm needs, you have to define a formal machine model and to implement algorithms in a programming language whose commands correspond to concrete steps of the formal machine.

Donald Knuth follows this approach in his seminal book *The Art of Computer Programming* [Knu97]. He formulates algorithms in a machine language for the MIX architecture and plays close attention to concrete and best possible values of constants in the analyses. Knuth implements insertion sort as follows.

```
START ENT1  2-N            1          S1. Loop on j. j ← 2.
2H    LDA   INPUT+N,1      N-1        S2. Set up i, K, R.
      ENT2  N-1,1          N-1        i ← j-1.
3H    CMPA  INPUT,2        B+N-1-A    S3. Compare K : K_i.
      JGE   5F             B+N-1-A    To S5 if K ≥ K_i.
4H    LDX   INPUT,2        B          S4. Move R_i, decrease i.
      STX   INPUT+1,2      B          R_{i+1} ← R_i.
      DEC2  1              B          i ← i-1.
      J2P   3B             B          To S4 if i > 0.
5H    STA   INPUT+1,2      N-1        S5. R into R_{i+1}.
      INC1  1              N-1
      J1NP  2B             N-1        2 ≤ j ≤ N.
```

The locations `INPUT+1` through `INPUT+N` are the array to be sorted. The first column contains the MIX program and the third column contains comments. In the second column you find the number of times each instruction is executed, where N is the size of the input, A is the number of times i decreases to zero in step S4, and B is the number of moves. The running time of the program on the MIX machine is $9B + 10N - 3A - 9$ units. A thorough analysis shows that $A = N - 1$ and $B = \frac{N^2-N}{2}$ in the worst-case.

Software Development

In *The Art of Computer Programming*, Knuth derives precise bounds on the worst-case number of execution steps of programs for the MIX architecture mainly to explain and understand the implemented algorithms. For different reasons, such bounds are of growing interest in software development.

For many practical applications it is insufficient to determine the asymptotic behavior of program only. You rather need *concrete upper bounds for specific hardware* to safely predict the resource consumption for a specific input or to compare two programs with the same asymptotic behavior. That is to say, you have to determine closed functions in the sizes of the inputs of the program that bound the number of clock cycles or memory cells on a given system—bounds as developed for insertion sort for the MIX architecture.

Concrete worst-case bounds are particularly useful in the development of embedded systems and hard real-time systems. In the former, you want to use hardware that is *just good enough* to accomplish a task in order to produce a large number of units at lowest possible cost. In the latter, you need to guarantee specific worst-case running times to ensure the safety of the system.

Another area of application of concrete bounds is cloud and grid computing. *In the cloud*, a program is often simply terminated if it exceeds the resources—such as memory and computing time—that a client reserved for it in advance. Consequently, clients can save time and money by knowing a non-asymptotic bound on the resource consumption of the program. On the other side, the operator of the cloud could use resource bounds for better load balancing and scheduling.

1.2 Automatic Computation of Bounds

Even for basic programs, a manual analysis of the specific (non-asymptotic) resource cost of a program is cumbersome, error-prone, and time consuming. Not everyone commands the mathematical ease of Knuth and even he would run out of steam if he had to do these calculations over and over again while going through the debugging loops of program development. In short, derivation of precise bounds by hand appears to be unfeasible in practice in all but the simplest cases.

As a result, *automatic methods for static resource analysis* are highly desirable and have been the subject of extensive research. Of course, one can not expect the full automation of a manual analysis that involves creativity and sophisticated mathematics. But in most resource analyses in software development the greater part of the complexity arises from the glut of detail and the program size rather than from conceptual difficulty.

In recent years, the resource analysis community made great advances in the development of automatic computation and formal verification of resource bounds. Nevertheless, the automation of resource analysis entails inherent theoretical limitations.

Limits of Automatic Methods

Assume we have formally defined what programs are, how they are executed by a machine, and what the resource consumption during their execution is. We are now interested in the following problem. Given a program P, compute a function of the sizes of P's inputs that bounds the resource consumption of P.

If you work on methods that compute the resource consumption of programs then you need means to measure the quality of such methods. The first question that you have to explore is how precise a concrete (non-asymptotic) bound on the resource consumption, as a function of the sizes of the inputs, can be in general.

Sometimes it is already hard to even describe a resource bound that exactly describes the worst-case resource behavior of a program for inputs of size n. Consider for example the well-known algorithm *Sieve of Eratosthenes*. It takes a list of integers *[2,3,...,n]* as input and computes a list of primes that is included in this list. The time consumption of this algorithm depends on the number of primes in the list. Since there is an asymptotically tight upper bound on this number (namely $O(\frac{n}{\log n})$) it is possible to give an *asymmetrically tight* upper on the time consumption of the algorithm (namely $O(n(\log n)(\log\log n))$). But in order to give an *exact* description of the worst-case time consumption as a function of the length of the input, it seems to be hard to do so without using a term like "the number of primes less or equal to n". Such a description is unsatisfying in two ways. First, it is maybe not meaningful to a user and, second, the actual resource consumption for a given input of length n is not immediately computable.

This example shows that it seems that we have to be satisfied with automatic computed bounds that only *asymptotically* match the worst-case resource behavior. For the Sieve of Eratosthenes it is however the case that an asymptotically tight bound on its resource behavior relies on deep results on the density of the primes. So it seems to be hopeless that it could be *automatically computed* from the code of the program.

That is why we can not even expect an automatic method to compute asymptotically tight upper bounds on the worst-case resource behavior in general. This leads to the question what we expect of automatically generated resource bounds.

Undecidability

The rule of thumb in automatic resource analysis is: *if you have nothing but a minimal requirement on the quality of bounds then the computation of the bounds is already impossible in general.* I illustrate this with two examples.

A first minimal requirement on computed resource bounds would be to demand that they should be a polynomial if the worst-case resource behavior of the program is polynomially bounded. Every algorithm that would compute such bounds, could also be used to decide if a given program runs in polynomial time. But as the following reduction from the *halting problem* shows, the latter problem is undecidable. The input program f is transformed to f' such that f' first deletes its input and then behaves like f. It is then the case that f' runs in polynomial time (in fact in constant time) if and

only if f terminates on the empty input.

A second minimal requirement on computed bounds would be to demand that they should bound the resource usage of a program for a given input with a finite number if the resource usage for that input is finite. But every algorithm that would compute such bounds on the running time of programs would directly solve the *halting problem*. So it is an undecidable problem to compute such bounds.

Valuation of Resource Analyses

Even though the problem is undecidable, we can still develop algorithms that compute resource bounds. However, the best we can achieve are algorithms that are not complete. This means that they may terminate for some input programs without providing bounds.

The means of measurement of the quality of automatic resource analyses are the following.

- Range: Which programs can be successfully analyzed?
- Precision: How close are the bounds to optimal ones?
- Efficiency: How long does it take to compute bounds?
- Verifiability: How easy is it to check whether a computed bound is sound?

The main challenge in automatic resource analysis is to develop analysis methods that provide good bounds for as many programs as possible. Theoretically it is even achievable to find a method that works for nearly all programs that appear in practice. An analogy are the non-measurable functions in physics: it is well-known that many functions over the real numbers are not measurable (i.e., do not have a Lebesgue integral). In practice, however, non-measurable functions hardly ever appear in physical calculations.

Tour d'Horizon

The state of the art in automatic resource analysis relies on various techniques of program analysis. On the one hand there is the large field of worst-case execution time (WCET) analysis, which is mainly focused on the run-time analysis of sequential code without loops taking into account low-level features like hardware caches and instruction pipelines [WEE+08].

On the other hand there is an active research community that employs type systems and abstract interpretation to analyze loops, recursion and data structures. My work falls within this area of research, which I sketch in this small tour d'horizon. Please refer to Chapter 8 for a detailed comparison of my work with existing techniques.

Classic methods for automatic or semi-automatic resource analysis are based on *recurrence relations* or *recurrences*. It seems to have been common knowledge since the earliest days of algorithmic analysis that the resource consumption of recursive

programs can be naturally described by such recurrence relations [Knu97]. The worst-case time consumption $T(n)$ of an implementation of insertion sort might for instance be described by the following recurrence where c_0, c_1 and c_2 are constants.

$$\begin{aligned} T(0) &= c_0 \\ T(n) &= c_1 + c_2(n-1) + T(n-1) \end{aligned}$$

As early as 1975, Wegbreit [Weg75] proposed an automatic analysis that consists of two phases. First, derive the recurrence relations from the program. Second, compute closed forms for recurrence relations that can be easily understood and further processed. Wegbreit implemented this analysis idea for LISP programs but notes that it "can only handle simple programs" [Weg75]. The most complicated examples that he provides are a reverse function for lists and a union function for sets represented by lists. Nevertheless, Wegbreit's technique remained predominant in automatic resource analysis for the next 25 years [Ram79, Coh82, Mét88, HC88, ZZ89, Ros89, FSZ91, DL93, Ben01, Gro01, BPZZ05, AAGP08, AGM11]. Benzinger [Ben01] notices in 2001:

> "Automated complexity analysis is a perennial yet surprisingly disregarded aspect of static program analysis. The seminal contribution to this area was Wegbreit's METRIC system, which even today still represents the state-of-the-art in many aspects."

In consideration of the substantial work on Wegbreit's method, it might be surprising that comparatively little progress in the area was made. There are two reasons.

1. It is a hard problem to compute recurrence relations from a program.
2. It is a hard problem to find closed forms for recurrence relations.

In general, it is already difficult to *manually* determine closed forms for recurrence relations. Admittedly, there exist powerful tools such as the well-known *master method* [CSRL01] and its generalizations [AB98, Rou01, EP08]. But these methods only determine *asymptotic* bounds and ignore base cases and constant factors. Additionally, the master theorem only applies to divide-and-conquer recurrences with one variable.

More fundamental approaches for solving recurrence relations build on sophisticated analytic methods such as *generating functions* [GKP94, FS09].

Such methods are the basis of solvers for automatically computing closed forms of recurrences that are implemented in computer algebra systems such as *Mathematica* and *Maple*. However, they have limitations that make them less suitable for solving recurrence relations that originate from programs. For instance, *RSolve*—the built-in solver of *Mathematica*—does not support functions of multiple variables [Ben01].

Obtaining the recurrence relations from a program in the first place is anything but straightforward, even for simple functional programs. One of the difficulties is that you need to infer size relations between different program variables. This is an undecidable problem that is sometimes as difficult as the resource analysis itself. For instance it is

already non-trivial to infer that a reverse function for lists produces a list of the same length as the input. That is why many modern automatic resource analyses are still restricted to simple programs like functions with primitive recursion [Ben01, Ben04].

Recently there has been a lot of progress in both deriving and solving recurrence relations, especially with techniques that only approximate closed forms with upper bounds [Ben01, Gro01, Ben04, AAG+07, AAGP08, AGM11]. However, insertion sort is still at the frontier of the class of programs this technique can handle [Ben01, AGM11] while slightly more involved programs like quick sort for lists of lists still seem to be beyond its scope.

The intrinsic problems with the classic methods for automatic resource analysis caused a renewed research interest in novel approaches to the problem in recent years.

A successful method to estimate time bounds for C++ procedures with loops and recursion was recently developed by Gulwani et al. [GMC09, GG08] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. A recent innovation for non-recursive programs is the combination of disjunctive invariant generation via abstract interpretation with proof rules that employ SMT-solvers [GZ10].

Another approach is the use of sized types [HPS96, HP99, CK01, Vas08] which provide a general framework to represent the size of the data in its type.

Most closely related to the work I present in my dissertation is the work on automatic amortized analysis [HH10a, HH10b, HJ03, HJ06, HR09, JLH+09, JHLH10]. While having appealing features (see Section 2.2 for an informal introduction) these analyses are restricted to linear resource bounds and can thus not infer a time bound for a program like insertion sort.

1.3 Resource-Aware Programming

The aim of my research is to understand, formally describe, and predict the complexity of computations to simplify the development of reliable software systems. In this dissertation, I present the first automatic amortized analysis that computes polynomial resource bounds.

The techniques I present provide foundations for designing and implementing full-featured programming languages that enable software engineers to work with quantitative resource bounds in the same way they work with usual type information. Like types, resource bounds should be inferred in most cases. But if the inference fails it should be simple and natural to enrich parts of programs with resource information and to formally reason about soundness in a flexible way.

My work rests upon great achievements in the research on programming languages, program analysis, and linear optimization. The tools I use include amortized complexity analysis, linear type systems, operational semantics, and LP solving.

This Dissertation

In this work, I present *Resource Aware ML (RAML)*, a programming language that supports automatic computation and verification of resource bounds without sacrificing natural and succinct programming. RAML is a first-order ML-like language that features integers, lists, binary trees, and recursion. The language is small enough to keep proofs and definitions readable but expressive enough to hint at the treatment of other language features.

I *formalize the resource consumption of the evaluation of RAML programs* in a realistic and parametric way that allows for both, different hardware architectures and a wide range of resource metrics. To this end, I define a big-step operational semantics that is parametrized with resource metrics that can be directly related to the compiled assembly code for a specific system architecture [JLH+09]. The semantics formalizes the resource consumption of both terminating and non-terminating computations.

I develop elaborated *resource-parametric type systems* whose type judgments establish concrete worst-case bounds in terms of closed, easily understood polynomials. The type systems allows for an efficient and completely automatic inference algorithm, which is based on linear programming. As any type systems, they are naturally compositional and lend themselves to the smooth integration of components whose implementation is not available. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [BHMS04].

I *prove* the non-trivial soundness of the derived resource bounds with respect to the formalized resource-consumption of programs by the operational semantics. The proof is technically involved but relies on standard techniques from program analysis and type systems.

I verify the practicability of the approach with a publicly available implementation and a reproducible *experimental evaluation*. Experiments show that the analysis works for realistic examples and that the constant factors in the computed bounds are reasonably precise and even match the measured worst-case running times of many functions.

To the best of my knowledge, the proposed technique is the first that allows the fully automatic computation of evaluation-step bounds for involved programs such as quick sort for lists of lists, the computation of the length of the longest common subsequence via dynamic programming, and the multiplication of a list of matrices with matching but possibly different dimensions.

Achievements

Resource Aware ML enables a natural programming style and it can be used without understanding the built-in automatic resource analysis. Additionally, the amortized method provides an intuition that guides programmers in writing code that can be analyzed.

To give you a concrete idea of the analysis from a users' point of view, I demonstrate

the resource analysis of the sorting algorithm insertion sort in RAML. Insertion sort can be implemented like in a textbook on functional programming as follows.

```
insert: (int,L(int)) → L(int)

insert (x,l) = match l with | nil → [x]
                 | y::ys → if x <= y then x::y::ys
                                     else y::insert(x,ys);
isort : L(int) → L(int)

isort l = match l with | nil → nil
             | x::xs → insert(x, isort xs);
```

At the press of a button, the prototype implementation produces the following output. The computation takes less than 0.04 seconds on my laptop[1].

```
The number of evaluation steps consumed by insert is at most:
         12.0*n + 5.0
where n is the length of the second component of the input

The number of evaluation steps consumed by isort is at most:
         6.0*n^2 + 6.0*n + 3.0
where n is the length of the input
```

We manually identified worst-case inputs for insertion sort (namely reversely ordered lists) and compared the measured running time with the computed bound. The results show that RAML computes a tight evaluation-step bound for insertion sort. To the best of my knowledge, no other paper reported an automatically computed bound for insertion sort that exactly matches its measured run-time cost.

It is possible to link the resource metric to a compiler and to specific system architectures [JLH+09] to bound the number of clock cycles on that architecture. In this way, a programmer can compare the performance guaranties for different implementations and different systems in a couple of seconds while developing a program.

As any automatic method for resource bound computation, the technique we developed for RAML has limitations. The computed bounds are polynomials and the user has to provide a maximal degree of the bounding polynomials. The larger the maximal degree is, the larger is the search space of the bounds. The result of a successful analysis does however not depend on the maximal degree.

It is technically convenient to work with polynomials since they are closed under composition, multiplication and addition. Furthermore, polynomially bounded functions are considered to be the class of efficient computation by many computer scientists.

In the following, I summarize the features of my analysis technique by applying the means of measurement of the quality of automatic resource analyses from Section 1.2.

[1] A 2010 MacBook Air with a 2.13 GHz Intel Core 2 Duo.

Range Our method is restricted to polynomial bounds and there is still a large number of polynomially bounded programs that cannot be analyzed in our system. An example is the sorting algorithm *bubble sort*, which is often implemented such that the function is recursively called if the input list is *not sorted already*. Furthermore, the bounds are functions of sizes of inductive data structures such as lists and trees but not functions of integer or floating-point inputs.

It is not easy to abstractly characterize the class of programs that can be analyzed because the analysis does not impose any syntactic restrictions on RAML programs. For instance, it is entirely possible to compute a resource bound for a non-terminating program if its resource consumption is polynomial with respect to a given resource (e.g. heap space).

However, the experiments with the prototype indicate that the analysis scales well for larger programs that are written in a programming style that is usually used by functional programmers. Our method performs particularly well on programs with nested data structures, non-structural recursion, and composed functions. For instance, RAML computes a evaluation-step bound for a program that takes a tree of matrices (lists of lists) with matching but arbitrary many dimensions and multiplies the matrices in breadth-first order using a functional queue implemented with two lists.

Precision An automatic analysis can of course not always achieve the same accuracy as a careful manual analysis. Since RAML computes polynomial bounds, it can not infer an asymptotically tight evaluation-step bound for a function such as merge sort. It has an asymptotic worst-case running time of $O(n \log n)$ but the analysis computes a quadratic bound.

RAML infers however asymptotically tight bounds for most examples with a polynomial worst-case behavior that we implemented. We manually identified worst-case inputs of several sizes for some of the examples and compared the measured resource consumptions to the computed bounds. Our experiments show that the constant factors in the bounds are surprisingly precise and even exactly match measured resource consumption for many programs, including quick sort and insertion sort for lists of lists.

Efficiency The inference of the resource bounds is performed in two steps. First, RAML computes a set of linear constraints from the program text. Second, the constraints are solved by an off-the-shelf LP solver. The number of linear constraints grows exponentially in both the size of the program and the maximal degree of the bounds that the analyzer is trying to find.

In practice, the analysis works fast and efficient. If the maximal degree is low then you can compute bounds for programs with several hundred lines of code in a few seconds. Even for more complicated examples that require a higher maximal degree the analysis is reasonably efficient. For example, the computation of the

evaluation-step bound for the breadth-first traversal with matrix multiplication has about 80 lines of code, requires a maximal degree of 5, and runs in 35 seconds on a 2010 MacBook Air with a 2.13 GHz Intel Core 2 Duo.

One reason for the efficiency of the analysis is that the linear constraints that the type system admits have a very simple form. It is similar to LPs that are derived from network flow problems. Such *network problems* can be solved by LP solvers extremely fast without using floating point arithmetic. In fact, the computation of the constraints takes often longer than the actual constraint solving[2].

Note that we focused on soundness rather than efficiency in our Haskell prototype implementation. There is a lot of room for improvement by writing more efficient code, reducing the number of constraints, better integrating the LP solver, and using a commercial, industrial-strength LP solver.

Verifiability A great advantage of the type-based approach is that our analysis not only computes a worst-case resource bound but also a type derivation for that bound. This type derivation can be seen as a proof for the bound that can be easily checked.

A type derivation of a bound can be automatically translated into formalized proofs in program logic [BHMS04]. These proofs can be shipped with the program to certify its resource consumption.

In general, our automatic analysis copes gracefully with failure. Our type-based approach enables the seamless integration of manually analyzed portions of code by expressing the derived bounds in our resource-parametric types. This enables the manual improvement of automatically generated bounds and the automatic analysis of code that uses the manually analyzed parts.

Functional Programming

There are several reasons why I decided to analyze functional rather than imperative or object-oriented programs. For one thing, I favour functional programming languages because they inspire programmers to strive for elegance and beauty. For another thing, I find it beneficial to study my analysis method on purely functional programs first because I can focus on the actual resource analysis without dealing with the notorious pitfalls of the imperative world:

- The resource consumption of RAML functions depends only on their arguments rather than on the global program state.
- Data structures such as lists and trees are guaranteed to be acyclic.

[2] We use the fantastic open-source LP-solver CLP from the COIN-OR project.

- RAML programs are well-typed and *can't go wrong* at run time if they have enough resources.

I have often been criticized for analyzing functional programs and a critical reader might raise the following objection:

> "Functional programming might be great in theory but it is too slow and not used in practice—especially not in embedded system and real-time systems, which require exact control of resources."

It is true that the majority of program code is written in object-oriented and imperative languages. It is also true that some problems such as maintaining large hash tables should be solved in an imperative style for performance reasons. That is why I find it important to develop automatic resource analysis for imperative languages. However, many new concepts have been studied carefully for functional languages first before they have been transferred to imperative programs. Examples are static type systems and type inference, polymorphism, and function closures.

Moreover, functional programming languages are more and more used in practice. A popular example is Microsoft's F# on the .NET platform.

In safety-critical embedded systems, functional programming has a long and successful history. The synchronous data-flow language Lustre was introduced in 1987 [CPHP87] and is now the core of Scade, a commercial software suit for the development of safety-critical embedded software. Scade is used by many well-known companies, including Airbus, Eurocopter, and Siemens[3]. Examples of modern functional languages for embedded systems are Lucid Synchrone [Pou06] and Hume [HM03].

The developers of Hume integrated a linear automatic amortized analysis into a compiler for Hume [HDF+06]. It has been successfully used in concrete embedded system to compute memory and clock-cycle bounds for 32 MHz Renesas M32C/85U embedded micro-controllers [JLH+09]. Also note that many embedded systems are developed by using graphical modeling tools that generate C code [KSLB03]. I think that resource analyses are best integrated in these high-level modeling tools and that my approach could be of interest there [CCM+03].

[3] See `http://www.esterel-technologies.com`

Computers are getting smarter all the time. Scientists tell us that soon they will be able to talk to us. (And by "they", I mean "computers". I doubt scientists will ever be able to talk to us.)

DAVE BARRY

Dave Barry's Bad Habits: A 100% Fact-Free Book (1987)

2 Informal Account

In this chapter, I introduce the idea of automatic amortized analysis and informally present the main contributions of my work.

First, Section 2.1 presents *amortized analysis with the potential method*, a technique for quantitative analysis that has been introduced by Tarjan [Tar85] to manually analyze the efficiency of data structures and algorithms.

By presenting illustrative examples, I then show in Section 2.2 how this technique can be automated to statically analyze programs. I follow the chronological order of the development of this automatic amortized resource analysis. That is, I move from the analysis of programs with *linear* resource consumption [HJ03], to programs with *univariate polynomial* resource consumption [HH10b, HH10a], to programs with *multivariate polynomial* resource consumption [HAH11].

The extension of automatic amortized resource analysis from linear to (univariate and multivariate) polynomial bounds is the main contribution of my dissertation. Section 2.3 summarizes the main novel ideas and concepts that I contribute.

2.1 Manual Amortized Analysis

For a given data structure we are often interested in the cost of a sequence of operations whose costs vary depending on the state of the data structure. To analyze such a sequence of operations, Sleator and Tarjan [Tar85] proposed *amortized analysis with the potential method*.

The concept of potential is inspired by the notion of potential energy in physics. The idea is to define a potential function $\Phi(D)$ that maps data structures D to non-negative numbers. Operations that change the data structure can then cause a gain or loss of potential. The amortized cost $A(op(D))$ of an operation $op(D)$ is defined as the sum of its actual cost $K(op(D))$ and the (possibly negative) difference of the potentials before

and after its evaluation:

$$A(op(D)) = K(op(D)) + \Phi(op(D)) - \Phi(D)$$

The sum of the amortized costs taken over a sequence of operations plus the potential of the initial data structure then furnishes an upper bound on the actual cost of that sequence.

$$\begin{aligned}\Phi(D_0) + \sum_{1\le i\le n} A(op(D_i)) &= \Phi(D_0) + \sum_{1\le i\le n} K(op(D_i)) + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \Phi(D_n) + \sum_{1\le i\le n} K(op(D_i)) \;\ge\; \sum_{1\le i\le n} K(op(D_i))\end{aligned}$$

Tarjan [Tar85] describes the advantages of amortized analysis as follows.

> "A worst-case analysis, in which we sum the worst-case times of the individual operations, may be unduly pessimistic, because it ignores correlated effects of the operations on the data structure. On the other hand, an average-case analysis may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. In such a situation, an amortized analysis, in which we average the running time per operation over a (worst-case) sequence of operations, can yield an answer that is both realistic and robust."

A standard example [Oka98] that demonstrates the benefits of the amortized method is the analysis of a functional queue. A *queue* is a first-in-first-out data structure with the operations *enqueue* and *dequeue*. The operation *enqueue*(a) adds a new element a to the queue. The operation *dequeue*() removes the oldest element from the queue. A queue is often implemented with two lists L_{in} and L_{out} that act as stacks. To enqueue a new element in the queue, you simply attach it to the beginning of L_{in}. To dequeue an element from the queue, you detach the first element from L_{out}. If L_{out} is empty then you transfer the elements from L_{in} to L_{out}, thereby reversing the order of the elements.

The problem is now to determine the number of list (or stack) operations (attach and detach) that are needed to perform a sequence of *enqueue* and *dequeue* operations. The difficulty is that the cost of *dequeue* is not constant but depends on the state of the data structure.

To ease the analysis, we introduce a potential $\Phi(L_{in}, L_{out}) = 2 \cdot |L_{in}|$ that is defined as twice the length of the list L_{in}. The amortized cost of *enqueue* is then *A(enqueue)* = 3—one to pay for the attachment to L_{in} and two to pay the increase of potential. The amortized cost of *dequeue* is *A(dequeue)* = 1. To see why, we consider two cases. If L_{out} is not empty then we just detach the first element of L_{out} and the potential is unchanged. So the amortized cost is simply the actual cost 1 in this case. If L_{out} is empty then we have to move the elements in L_{in} to L_{out}. The actual cost is then $2 \cdot |L_{in}|$. Because L_{in} is empty thereafter, $2 \cdot |L_{in}|$ is exactly the decrease of potential that is caused by the move. And since we finally have to detach the first element of L_{out}, the amortized cost is $A(\textit{dequeue}) = (2 \cdot |L_{in}| + 1) + 0 - 2 \cdot |L_{in}| = 1$.

Following the potential method, we now have to sum up the initial potential and the amortized costs of operations. That is why our analysis shows that the number of list operations performed in a sequence of m *enqueue* and n *dequeue* operations is less than $3 \cdot m + n + 2 \cdot k$ if k is the initial length of L_{in}.

2.2 Automatic Amortized Analysis

In the following I apply the potential method of amortized analysis to statically analyze functional programs. In a nutshell, the idea is as follows. Look upon a program as a graph in which the edges are the atomic steps performed by the program and the vertices are the program points between the atomic steps.

We now label each program point with a potential function, a mapping from machine states to numbers. Our goal is to find a labeling that covers the resource costs of all possible evaluations of the program; that is, to find potential functions such that for every possible evaluation, the potential at a program point suffices to cover the cost of the next transition and the potential at the succeeding program point.

In this approach, the amortized costs of the transitions are always less or equal to zero. The initial potential is therefore already an upper bound on the resource consumption of the program.

A programmer should not be bothered with the clutter of the potential functions in her programs. That would reduce her productivity and would make the code harder to read. Instead, the potential functions should be inferred completely automatically by the computer.

To make such an automatic amortized analysis feasible, it is necessary to restrict the choice of potential functions. The more potential functions we allow, the more accurate and wide-ranging is the analysis. However, there is a trade-off between the diversity of potential functions and the efficiency of the analysis.

2.2.1 Linear Potential

The first automatic amortized analysis was introduced by Hofmann and Jost [HJ03] to analyze the heap-space consumption of first-order functional programs. They fixed potential functions to be *linear* in the size of the data in the memory.

The potential at a program point is defined by a static annotation of the reachable data at that point. More precisely, inductive data structures are statically annotated with non-negative rational numbers q to define non-negative potentials $\Phi(n) = q \cdot n$ as a function of the size n of the data. Then a sound, albeit incomplete, type-based analysis of the program text statically verifies that the potential is sufficient to pay for all operations that are performed on this data structure during any possible evaluation of the program.

This idea is best explained by example. Consider the function *attach* that takes an integer and a list of integers and returns a list of pairs of integers such that the

first argument is attached to every element of the list. For instance, the expression *attach(1,[1,2,3,4])* evaluates to *[(1,1),(1,2),(1,3),(1,4)]*. The function can be implemented as follows.

```
attach(x,l) = match l with | nil  →  nil
                           | y::ys →  (x,y)::(attach (x,ys))
```

Suppose that we need three memory cells to create a list cell of the resulting list—two cells for the pair of integers and one cell for the pointer to the next list element. The heap-space usage of an execution of *attach(x,ℓ)* is then $3n$ memory cells if n is the length of ℓ.

To infer an upper bound on the heap-space usage of the function, we annotate the type of *attach* with a priori unknown resource annotations s, s', q and p that range over non-negative rational numbers.

$$attach \quad : \quad (int, L^q(int)) \xrightarrow{s/s'} L^p(int, int)$$

The intuitive meaning of the resulting typing is as follows: to evaluate *attach(x,ℓ)* one needs q memory cells per element in the list ℓ and s additional memory cells. After the evaluation there are s' memory cells and p cells per element of the returned list left. We say that the list ℓ has potential $\Phi(\ell, q) = q \cdot |\ell|$ and that the list $\ell' = attach(x,\ell)$ has potential $\Phi(\ell', p) = p \cdot |\ell'|$.

A static type analysis of the program code then derives linear constraints on the resource annotations. In the case of *attach*, the constraints would essentially state that $q \geq 3 + p$ and $s \geq s'$. Every valid instantiation of the resource annotations must satisfy these constraints. For instance, the following typing of *attach* is valid.

$$attach \quad : \quad (int, L^{(3)}(int)) \xrightarrow{0/0} L^{(0)}(int, int)$$

It states that the heap-space consumption of the function is less than the initial potential $3{\cdot}n$ if n is the length of the input list and thus furnishes a tight upper bound. The function *attach* can also be typed as follows.

$$attach \quad : \quad (int, L^{(5)}(int)) \xrightarrow{6/6} L^{(2)}(int, int)$$

This typing could be used for an inner occurrence of *attach* to type an expression like *f(attach(z,ys))* if the evaluation of $f(\ell)$ would consume $6 + 2 \cdot |\ell|$ heap cells.

The use of linear potential functions relieves one of the burden of having to manipulate symbolic expressions during the analysis by a priori fixing their format. This gives rise to a particularly efficient inference algorithm for the type annotations. It works like a standard type inference in which simple linear constraints are collected as each type rule is applied.

The constraints are solved with a linear-programming solver (LP solver) to obtain the best possible typing for the program. The function type that is needed to minimize the initial potential depends on the context in which the function is applied.

Automatic amortized analysis can be used with generic resource metrics [JLH+09]. As a result it can derive bounds on every quantity whose consumption in an atomic step is bounded by a constant. An important example is *time consumption*. Consider for instance the function *filter*:$(int, L(int)) \to L(int)$ that removes the multiples of a given integer from a list of integers.

```
filter(p,l) = match l with | nil  → nil
                           | x::xs → let xs' = filter(p,xs) in
                                     if x mod p == 0 then xs' else x::xs'
```

Suppose that the evaluation of the expression *filter(p,ℓ)* takes at most $16 \cdot |\ell| + 3$ atomic steps. Then the following typing expresses a tight upper bound.

$$filter \quad : \quad (int, L^{(16)}(int)) \xrightarrow{3/0} L^{(0)}(int)$$

As in the case of heap-space consumption, we can infer these potential annotations by solving the linear constraints that are produced by our type inference algorithm.

Since amortized analysis takes into account the interaction between the steps of a computation, it obtains tighter bounds than a mere addition of the worst case resource bounds of the individual steps. Generally, the constants in the bounds are very precise and often match exactly the worst-case behavior of the functions. Thanks to efficient, off-the-shelf LP solvers, the analysis takes only a few seconds, even on larger programs.

Hofmann and Jost's technique has been successfully applied to object-oriented programs [HJ06, HR09], to generic resource metrics [JLH+09, Cam09], to polymorphic and higher-order programs [JHLH10], and to Java-like bytecode by means of separation logic [Atk10]. The main limitation shared by these analysis systems is their *restriction to linear resource bounds* to enable efficient inference using linear constraint solving.

Chapter 4 formally describes linear automatic amortized analysis for first-order monomorphic functional programs.

2.2.2 Univariate Polynomial Potential

Linear amortized analysis is appealing because it offers a good trade-off between efficiency and range of the analysis. It can analyze many linear functions that appear in programming and the computation of the bounds takes only a view seconds on usual computers.

However, its limitation to linear bounds hampers its applicability in practice. Despite some efforts [SvKvE07], the problem of extending automatic amortized analysis to super-linear bounds remained open for several years.

A challenge in the extension to super-linear potential is to identify a set of functions that is both simple enough to allow for an efficient manipulation and expressive enough to constitute accurate bounds. A key point is the adaption of potential functions if the size of a data structure changes. How can we for instance transform a potential function f to a potential function f' such that $f'(n) = f(n-1)$? Such transformations are constantly needed in pattern matches and data construction. Thus they should be

very easy to compute but should not cause any loss of potential to ensure the precision of the bounds.

Recently, we were able to develop an automatic amortized analysis that efficiently computes (univariate) *polynomial* resource bounds for functional programs at compile time [HH10b, HH10a]. The main innovation is the use of potential functions of the form

$$\sum_{1\le i\le k} q_i \binom{n}{i} \text{ with } q_i \ge 0$$

They are attached to inductive data structures via type annotations of the form $\vec{q} = (q_1,\ldots,q_k)$ with $q_i \in \mathbb{Q}_0^+$. For instance, the typing $\ell{:}L^{(3,2,1)}(int)$, defines the potential $\Phi(\ell,(3,2,1)) = 3|\ell| + 2\binom{|\ell|}{2} + 1\binom{|\ell|}{3}$. One intuition for these numbers is as follows: The annotation $\vec{q}$ assigns the potential q_1 to every element of the list, the potential q_2 to every element of every proper suffix of the list, q_3 to the elements of the suffixes of the suffixes, etc.

To achieve a highly efficient computation of valid polynomial potential annotations we designed a type system that emits *linear constraints* only. In this way, we build on the tried-and-tested technique of the linear analysis system and can use fast LP solvers to compute the bounds.

In a nutshell, our approach is as follows. We start from an as yet unknown potential-function of the form $\sum p_j(n_j)$ with polynomials p_j of a given maximal degree k and n_j referring to the sizes of the parameters. We then derive linear constraints on the coefficients of the p_j by type-checking the program. Recall that the polynomials $p(n)$ of degree k are represented as sums $\sum_{0\le i\le k} q_i\binom{n}{i}$ with $q_i \ge 0$. Compared with the traditional representation $\sum q_i \cdot n^i$, $q_i \ge 0$, the use of binomial coefficients has the following advantages.

1. Some naturally arising resource bounds such as $\sum_{1\le i\le n} i$ cannot be expressed as a polynomial with non-negative coefficients in the traditional representation. On the other hand it is true that $\binom{n}{2} = \sum_{1\le i\le n} i$.

2. It is the largest class $\mathscr{C}$ of non-negative, monotone polynomials such that $p \in \mathscr{C}$ implies $f(n) = p(n+1) - p(n) \in \mathscr{C}$ (see Chapter 5). All three properties are clearly desirable. The latter one, in particular, expresses that the "spill" arising upon shortening a list by one falls itself into $\mathscr{C}$.

3. The identity $\sum_{1\le i\le k} q_i\binom{n+1}{i} = q_1 + \sum_{1\le i\le k-1} q_{i+1}\binom{n}{i} + \sum_{1\le i\le k} q_i\binom{n}{i}$ gives rise to a local typing rule for pattern matches which naturally allows the typing of both recursive calls and other calls to subordinate functions.

4. The linear constraints arising from the type inference have a very simple form due to the above equation. In particular, each constraint involves at most three variables without any multiplicative factors and is thus of the form $x_1 + x_2 - x_3 \ge q$.

A key notion in the polynomial system is the additive shift $\lhd$ of a type annotation which is defined through $\lhd(q_1,\ldots,q_k) = (q_1+q_2,\ldots,q_{k-1}+q_k,q_k)$ to reflect the identity from item 3. It is for instance present in the typing $tail : L^{\vec{q}}(int) \xrightarrow{0/q_1} L^{\lhd(\vec{q})}(int)$ of the function *tail* that removes the first element from a list.

The idea behind the additive shift is that the potential resulting from the contraction $xs{:}L^{\lhd(\vec{q})}(int)$ of a list $(x{::}xs){:}L^{\vec{q}}(int)$ (usually in a pattern match) is used for three purposes: i) to pay the constant costs after and before the recursive calls (using q_1), ii) to fund calls to auxiliary functions (using $(q_2,\ldots,q_n)$), and iii) to pay for the recursive calls (using $(q_1,\ldots,q_n)$).

To see how the polynomial potential annotations are used to compute polynomial resource bounds, consider the function *pairs* that computes the two-element subsets of a given set (representing sets as tuples or lists).

```
pairs l = match l with | nil  → nil
                       | x::xs → append(attach(x,xs),pairs xs)

append(l1,l2) = match l1 with | nil  → l2
                              | x::xs → x::append(xs,l2)
```

The expression *pairs([1,2,3])* evaluates for example to *[(1,2),(1,3),(2,3)]*. The function *append* consumes 3 memory cells for every element in the first argument. Similar to *attach* we can compute a tight resource bound for *append* by inferring the type

$$append \quad : \quad (L^{(3)}(int,int), L^{(0)}(int,int)) \xrightarrow{0/0} L^{(0)}(int,int)\,.$$

The evaluation of the expression *pairs*(ℓ) consumes six memory cells per element of every suffix of ℓ. The type that our system infers for *pairs* is

$$pairs \quad : \quad L^{(0,6)}(int) \xrightarrow{0/0} L^{(0)}(int,int)\,.$$

It states that a list ℓ in an expression *pairs*(ℓ) has the potential $\Phi(\ell,(0,6)) = 0\cdot|\ell| + 6\cdot\binom{|\ell|}{2}$ and thus furnishes a tight upper bound on the heap-space usage.

To type the function's body, the additive shift assigns the type $xs{:}L^{(0+6,6)}(int)$ to the variable *xs* in the pattern match. The potential is shared between the two occurrences of *xs* in the following expression by using $xs{:}L^{(6,0)}(int)$ to pay for *append* and *attach* (ii) and using $xs{:}L^{(0,6)}(int)$ to pay for the recursive call to *pairs* (iii); the constant costs (i) are zero in this example.

To compute the bound, we start with an annotation of the list types with resource variables as before.

```
pairs l = match l^(q1,q2) with | nil  → nil
                               | x::(xs^(p1,p2)) → append(attach(x,xs^(r1,r2)),pairs xs^(s1,s2))
```

The constraints that our type system computes include $q_2 \geq p_2$ and $q_1+q_2 \geq p_1$ (additive shift); $p_1 = r_1+s_1$ and $p_2 = r_2+s_2$ (sharing between two variables); $r_1 \geq 6$ (pay for non-recursive function calls); $q_1 = s_1$, $q_2 = s_2$ (pay for the recursive call). This system is solvable by $q_2 = s_2 = p_1 = p_2 = r_1 = 6$ and $q_1 = s_1 = r_2 = 0$.

For an example of a polynomial evaluation-step bound, consider the function *eratos*:$L(int)\to L(int)$ that implements the sieve of Eratosthenes. It successively calls the function *filter* to delete multiples of the first element from the input list. If *eratos* is called with a list of the form $[2,3,\ldots,n]$ then it computes the list of primes p with $2 \le p \le n$.

```
eratos l = match l with | nil  → nil
                        | x::xs → x::eratos(filter(x,xs))
```

Recall the worst-case number of atomic steps that *filter(x)* needs is $16\cdot|x|+3$. This exact bound is reflected in the typing *filter:* $(int, L^{(16)}(int)) \xrightarrow{3/0} L^{(0)}(int)$.

In an evaluation of *eratos*(ℓ), the function *filter* is called once for every sublist of the input list ℓ in the worst case. The calls of *filter* thus need $16\binom{n}{2}+3n$ atomic steps in the worst-case. This is for example the case if ℓ is a list of pairwise distinct primes. Additionally to the cost caused by *filter*, *eratos* needs 3 steps if the list is empty and 9 steps for each element in the input list. Thus, the total worst-case number of atomic steps the function needs, is $16\binom{n}{2}+12n+3$ if n is the size of the input list.

To bound on the number of atomic steps needed by *eratos*, our analysis system automatically computes the following type.

$$eratos \quad : \quad L^{(12,16)}(int) \xrightarrow{3/0} L^{(0)}(int)$$

Since the typing assigns the initial potential $16\binom{n}{2}+12n+3$ to a function argument of size n, the analysis computes a tight evaluation-step bound for *eratos*.

Univariate polynomial amortize analysis is presented in Chapter 5 in detail.

2.2.3 Multivariate Potential

The univariate polynomial analysis [HH10b, HH10a] works for many functions that admit a worst-case resource consumption that can be expressed by sums of univariate polynomials like n^2+m^2. However, many functions with multiple arguments that appear in practice have *multivariate* cost characteristics like $m\cdot n$. Moreover, if data from different sources are interlinked in a program then multivariate bounds like $(m+n)^2$ arise even if all functions have a univariate resource behavior. In these cases, the analysis fails, or the bounds are hugely over-approximated by $3m^2+3n^2$. The reason is that the potential is attached to a single data structure and does not take into account relations between different data structures.

To overcome these drawbacks, we developed an *automatic type-based amortized analysis for multivariate polynomial resource bounds* [HAH11]. We faced two main challenges in the development of the analysis.

1. The identification of multivariate polynomials that accurately describe the resource cost of typical examples. It is necessary that they are closed under natural operations to be suitable for local typing rules. Moreover, they must handle an unbounded number of arguments to accurately cope with nested data structures.

2. The smooth integration of the inference of size relations and resource bounds to deal with the interactions of different functions while keeping the analysis technically feasible in practice.

To address challenge one, we defined *multivariate resource polynomials* that are a generalization of the resource polynomials that are used in the univariate system (see Chapter 6). These polynomials are used as *global polynomial potential functions* which depend on the sizes of several parts of the input. Consequently, types are annotated with one global resource annotation in contrast to the local list annotations of the linear and univariate systems.

To address challenge two, we introduced local type rules that emit only simple linear constraints and are remarkably modest considering the variety of relations between different parts of the data that are taken into account.

The shape of the global potential annotations depends on the type of the respective data structures. The annotations take into account a wide range of connections between different parts of the data and are syntactically given by an inductively-defined index system. To give a flavor of the basic ideas, I informally introduce this global potential in this section for pairs of integer lists.

The initial potential of a function with arguments that are single integer lists can be expressed as a vector $(q_0, q_1, \ldots, q_k)$ that defines a potential-function of the form $\sum_{0\le i\le k} q_i \binom{n}{i}$. Note that the constant potential q_0 is already included in these global potential annotations. With this notation the types of the functions *pairs* and *eratos* from the previous subsection can be written as follows.

$$\begin{aligned} \mathit{pairs} &: (L(\mathit{int}),(0,0,6)) \to (L(\mathit{int}),(0,0,0)) \\ \mathit{eratos} &: (L(\mathit{int}),(3,12,16)) \to (L(\mathit{int}),(0,0,0)) \end{aligned}$$

To represent mixed terms of degree $\le k$ for a pair of integer lists we use a triangular matrix $Q = (q_{(i,j)})_{0\le i+j\le k}$ with $q_{(i,j)} \ge 0$. Then Q defines a potential-function of the form

$$\sum_{0\le i+j\le k} q_{(i,j)} \binom{n}{i}\binom{m}{j}$$

where m and n are the lengths of the two lists.

This definition has the same advantages as the univariate version of the system. Particularly, we can still use the additive shift to assign potential to sublists. To generalize the additive shift of the univariate system, we use the following identity.

$$\sum_{0\le i+j\le k} q_{(i,j)} \binom{n+1}{i}\binom{m}{j} = \sum_{0\le i+j\le k-1} q_{(i+1,j)} \binom{n}{i}\binom{m}{j} + \sum_{0\le i+j\le k} q_{(i,j)} \binom{n}{i}\binom{m}{j}$$

It is reflected by two additive shifts $\lhd_1(Q) = (q_{(i,j)} + q_{(i+1,j)})_{0\le i+j\le k}$ and $\lhd_2(Q) = (q_{(i,j)} + q_{(i,j+1)})_{0\le i+j\le k}$ where $q_{(i,j)} := 0$ if $i + j > k$. The shift operations can be used like in

the univariate case. For example, we derive the typing *tail1:* $((L(int),L(int)),Q) \to ((L(int),L(int)),\lhd_1(Q))$ for the function *tail1(xs,ys)=(tail xs,ys)* and every annotation Q.

To see how the mixed potential is used, consider the function *dyad* that computes the dyadic product of two lists.

```
mult(x,l) = match l with | nil  → nil
                         | y::ys → x*y::mult(x,ys)

dyad(l,ys) = match l with | nil  → nil
                          | x::xs → (mult(x,ys))::dyad(xs,ys)
```

Similar to previous examples, *mult* consumes $2n$ heap cells if n is the length of input. This exact bound is represented by the typing

$$mult \quad : \quad ((int,L(int)),(0,2,0)) \to (L(int),(0,0,0))$$

that states that the potential is $0+2n+0\binom{n}{2}$ before and 0 after the evaluation of *mult(x,ℓ)* if ℓ is a list of length n.

The function *dyad* consumes $2n+2nm$ heap cells if n is the length of first argument and m is the length of the second argument. This is why the following typing represents a tight heap-space bound for the function.

$$dyad \quad : \quad ((L(int),L(int)),\begin{pmatrix}0&0&0\\2&2&\\0&&\end{pmatrix}) \to (L(L(int)),0)$$

To verify this typing of *dyad*, the additive shift $\lhd_1$ is used in the pattern matching. This results in the potential

$$(xs,ys) \quad : \quad ((L(int),L(int)),\begin{pmatrix}2&2&0\\2&2&\\0&&\end{pmatrix})$$

that is used as in the function *eratos*: the constant potential 2 is used to pay for the *cons* operation (i), the linear potential $ys{:}(L(int),(0,2,0))$ is used to pay the cost of the evaluation of *mult(x,ys)* (ii), the rest of the potential is used to pay for the recursive call *dyad(xs,ys)* (iii).

Multivariate potential is also needed to assign a super-linear potential to the result of a function like *append*. This is, for example, needed in order to type an expression such as *pairs(append(ℓ_1,ℓ_2))*. If we consider heap-space consumption, *append* can have the following type.

$$append \quad : \quad ((L(int),L(int)),\begin{pmatrix}0&0&6\\2&6&\\6&&\end{pmatrix}) \to (L(int),(0,0,6))\,.$$

The correctness of the bound follows from the convolution formula $\binom{n+m}{2}=\binom{n}{2}+\binom{m}{2}+nm$ and from the fact that *append* consumes $2n$ heap cells if n is the length of the first

argument. The respective initial potential $2n + 6(\binom{n}{2} + \binom{m}{2} + mn)$ furnishes a tight bound on the worst-case heap-space consumption of the evaluation of *pairs(append(ℓ_1,ℓ_2))*, where $|\ell_1| = n$ and $|\ell_2| = m$.

I formally describe the multivariate analysis system in Chapter 6.

2.3 Overview of Contributions

The contributions of my dissertation were presented at the 19th European Symposium on Programming (ESOP'10) [HH10b], the eighth Asian Symposium on Programming Languages and Systems (APLAS'10) [HH10a], and the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11) [HAH11].

The main developments of the work with my collaborators are the following.

1. We addressed the longstanding problem of extending amortized analysis to non-linear resource bounds by presenting an automatic amortized analysis that computes *univariate polynomial resource bounds* [HH10b]. *(Chapter 5)*

2. We identified non-negative linear combinations of binomial coefficients as an ideal set of polynomial potential functions. They allow for an easy manipulation in local type rules despite being fine-grained enough to represent accurate bounds [HH10b]. *(Chapter 5)*

3. The main challenge for the inference of polynomial bounds is the need to deal with *resource-polymorphic recursion* (see Chapter 5), which is required to type most of the example programs we tested. It seems to be a hard problem to infer general resource polymorphic recursion, even for the linear system.

   We presented [HH10a] a pragmatic approach to resource-polymorphic recursion that works well and efficiently in practice. Despite being not complete with respect to the type rules, it infers types for most functions that admit a type-derivation. *(Chapters 5 and 6)*

4. Classically, the soundness theorems for automatic amortized analyses show that the derived resource bounds are sound with respect to a big-step operational semantics. A dissatisfying feature of classical big-step semantics is that it does not provide evaluation judgments for non-terminating evaluations. As a result, the soundness theorems for amortized resource analyses have in the past been formulated for terminating evaluations only [HJ03, JLH+09, JHLH10].

   We introduced [HH10a] a novel big-step operational semantics for partial evaluations that agrees with the usual big-step semantics on terminating computations. In this way, we retain the advantages of big-step semantics (shorter, less syntactic proofs; better agreement with actual behaviour of computers) while capturing the resource behaviour of non-terminating programs. This enables the proof of a strong soundness result: if the type analysis has established a resource bound

then the resource consumption of the (possibly non-terminating) evaluation does not exceed the bound. It follows that run-time bounds also prove termination. *(Chapter 3)*

5. We defined multivariate resource polynomials that generalize univariate resource polynomials and developed type annotations that correspond to *global polynomial potential functions* for amortized analysis which depend on the sizes of several data structures [HAH11]. *(Chapter 6)*

6. We developed a *multivariate polynomial amortized analysis* [HAH11]. It uses *local type rules* that modify type annotations for global potential functions. The type rules emit only simple linear constraints and are remarkably modest considering the variety of relations between different parts of the data that are taken into account. *(Chapter 6)*

7. We verified the practicability of our approach with a publicly available implementation and a reproducible *experimental evaluation*[1] [HH10b, HH10a, HAH11].

   Our experiments with the prototype implementation show that our system automatically infers tight univariate and multivariate bounds for complex programs that involve nested data structures such as trees of lists. Additionally, it can deal with the same wide range of linear programs as the previous systems.

   For instance, the prototype automatically infers evaluation-step bounds for the sorting algorithms quick sort and insertion sort that exactly match the measured worst-case behavior of the functions [HH10a].

   Other representative examples are the successful and precise analyses of the dynamic programming algorithm for the length of the longest common subsequence of two lists and of an implementation of matrix multiplication where matrices are lists of lists of integers. *(Chapter 7)*

[1] See `http://raml.tcs.ifi.lmu.de` for a web interface, example programs, and the source code.

Retrofitting a type system onto a language not designed with typechecking in mind can be tricky; ideally, language design should go hand-in-hand with type system design.

BENJAMIN C. PIERCE
Types and Programming Languages (2002)

3 Resource Aware ML

This chapter introduces the functional programming language *Resource Aware ML (RAML)*, a first-order, monomorphic fragment of ML that features lists, binary trees, and recursion.

In Section 3.1, I define the syntax of RAML. Section 3.2 contains a standard type system for RAML, as well as the definitions of well-typed expressions and well-typed programs. To prove the correctness of the resource analyses, I introduce a cost-aware big-step operational semantics for RAML in Section 3.3. It formalizes the call-by-value evaluation of RAML programs and monitors the resource consumption during evaluation. The semantics is parametric in the monitored resource and can track every quantity whose consumption during an atomic step is bounded by a constant.

3.1 Syntax

RAML is a first-order functional language with ML-like syntax. It features booleans, integers, pairs, lists, binary trees, recursion and pattern match. I decided to use an ML-like syntax because most people who know functional programming are familiar with ML. Consequently, it should be easy for them to read and write RAML programs.

I tried to keep the language as small as possible to enable definitions and proofs that are short enough to be checked by the reader in reasonable time. On the other hand, I wanted to include just enough features to demonstrate the main capabilities of the analysis techniques.

There are two main differences between RAML and ML. Firstly, RAML only allows for first-order and monomorphic functions. This greatly simplifies the type system and the semantics. To analyze higher-order and polymorphic programs, it is possible to transform them to equivalent first-order, monomorphic programs prior the analysis by defunctionalization [Rey72]. Moreover, there exists a *linear* amortized analysis system

that directly analyzes higher-order and polymorphic programs [JHLH10]. I think that the techniques described there also apply to the polynomial analysis systems I develop in this theses.

The second difference to ML is that RAML only contains binary trees and lists rather than user-definable inductive data types. This simplifies the type systems and the semantics while providing the main ideas of how to deal with inductive data structures in the analysis systems.

Below is the EBNF grammar for the *expressions of RAML*. I skip the standard definitions of integer constants $n \in \mathbb{Z}$ and variable identifiers $x, f \in \mathrm{VID}$.

$$
\begin{aligned}
e ::= {} & () \mid \mathit{True} \mid \mathit{False} \mid n \mid x \\
& \mid x_1 \ \mathit{binop}\ x_2 \mid f(x) \\
& \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \\
& \mid (x_1, x_2) \mid \mathit{nil} \mid \mathit{cons}(x_h, x_t) \mid \mathit{leaf} \mid \mathit{node}(x_0, x_1, x_2) \\
& \mid \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e \\
& \mid \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \\
& \mid \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2
\end{aligned}
$$

$$
\mathit{binop} ::= + \mid - \mid * \mid \mathit{mod} \mid \mathit{div} \mid \mathit{and} \mid \mathit{or}
$$

The expressions of RAML are in *let normal form*. This means that term formers are applied to variables only, whenever possible. This simplifies typing rules and semantics considerably without hampering expressivity in any way.

In the implementation we transform unrestricted expressions into a let normal form with explicit sharing before the type analysis. This is straightforward and reduces the complexity of the implementation of the analysis. *Explicit sharing* means that multiple occurrences of variables are introduced explicitly. Details on the code transformations that are preformed in the implementation before the analysis are described in Section 7.1.

In the examples in this theses, I use the same unrestricted RAML expressions as in the implementation to make them more readable. I also write *(x::y)* instead of *cons(x,y)*.

For the resource analysis it is unimportant which ground operations are used in the definition of *binop*. In fact, you can use here every function that has a constant worst-case resource consumption. I assume that integers have a fixed length, say 32 bits, to ensure this property of the integer operations. In the implementation we have some more operators such as ==, <, and >.

I also included a destructive pattern match in the implementation to enable manual deallocation. The treatment of destructive pattern matches in the analysis systems is very similar to the treatment of usual pattern matches. Since it does not convey any additional features of the analysis systems, I exclude it from this dissertation. You can find details on destructive pattern matching in the literature [HJ03].

3.2 Simple Types

In this section, I define the well-typed expressions of RAML by assigning a simple type—a usual ML type without resource annotations—to well-typed expressions. I then define well-typed (first-order) RAML programs.

Simple types are data types A and first-order types F as given by the following grammars.

$$A ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid L(A) \mid T(A) \mid (A, A)$$
$$F ::= A \to A$$

Let $\mathscr{A}$ be the set of simple data types and let $\mathscr{F}$ be the set of simple first-order types as defined by the preceding grammars.

To each data type $A \in \mathscr{A}$ we assign a set of *semantic values* $[\![A]\!]$ in the obvious way. For example $[\![T(\mathit{int}, \mathit{int})]\!]$ is the set of finite binary trees whose nodes are labeled with pairs of integers.

If $t \in T(A)$ is a binary tree then I write $\mathrm{elems}(t) = [a_1, \ldots, a_n]$ for the list of nodes $a_1, \ldots, a_n$ of t in pre-order. It is convenient to identify tuples like (A_1, A_2, A_3, A_4) with the pair type $(A_1, (A_2, (A_3, A_4)))$.

A *typing context* $\Gamma : \mathrm{VID} \to \mathscr{A}$ is a partial, finite mapping from variable identifiers to data types. As usual Γ_1, Γ_2 denotes the union of the contexts Γ_1 and Γ_2 provided that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. We thus have the implicit side condition $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$ whenever Γ_1, Γ_2 occurs in a typing rule. Especially, writing $\Gamma = x_1{:}A_1, \ldots, x_k{:}A_k$ means that the variables x_i are pairwise distinct.

Let FID be a set of function identifiers. A *signature* $\Sigma : \mathrm{FID} \to \mathscr{F}$ is a finite, partial mapping of function identifiers to first-order types.

The typing judgment $\Sigma; \Gamma \vdash e : A$ states that the expression e has type A under the signature Σ in the context Γ. It is defined by the *simple typing rules* in Figure 3.1. If $\Sigma; \Gamma \vdash e : A$ for some expression e then I say that e is well-typed in Γ under Σ.

The simple typing rules in Figure 3.1 are a subset of the resource-annotated typing rules from the following chapters if the resource annotations are omitted. As a result, they form an affine linear type system with a sharing rule S:SHARE that explicitly tracks multiple occurrences of variables. The type system thus imposes no linearity restrictions but gives finer information on occurrences of variables than a simple type system does. For simple types this does not result in any advantages compared to usual type rules. I only present the simple type rules with an explicit sharing rule to resemble the annotated type rules in the later chapters. There, this approach greatly simplifies the rules. For now, just note that the set of well-typed expressions is as expected and that the rules in Figure 3.1 are equivalent to the usual rules with this regard.

The expression $e[z/x, z/y]$ is the expression e in which all free occurrences of the variables x and y are replaced by the variable z.

$$\frac{}{\Sigma; x{:}B \vdash x : B}\ (\text{S:VAR}) \qquad \frac{}{\Sigma; \emptyset \vdash () : \mathit{unit}}\ (\text{S:CONSTU}) \qquad \frac{n \in \mathbb{Z}}{\Sigma; \emptyset \vdash n : \mathit{int}}\ (\text{S:CONSTI})$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\}}{\Sigma; \emptyset \vdash b : \mathit{bool}}\ (\text{S:CONSTB}) \qquad \frac{\mathit{op} \in \{+, -, *, \mathit{mod}, \mathit{div}\}}{\Sigma; x_1{:}\mathit{int}, x_2{:}\mathit{int} \vdash x_1\ \mathit{op}\ x_2 : \mathit{int}}\ (\text{S:OPINT})$$

$$\frac{\Sigma(f) = A \to B}{\Sigma; x{:}A \vdash f(x) : B}\ (\text{S:APP}) \qquad \frac{\mathit{op} \in \{\mathit{or}, \mathit{and}\}}{\Sigma; x_1{:}\mathit{bool}, x_2{:}\mathit{bool} \vdash x_1\ \mathit{op}\ x_2 : \mathit{bool}}\ (\text{S:OPBOOL})$$

$$\frac{\Sigma; \Gamma \vdash e_t : B \qquad \Sigma; \Gamma \vdash e_f : B}{\Sigma; \Gamma, x{:}\mathit{bool} \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f : B}\ (\text{S:COND})$$

$$\frac{\Sigma; \Gamma_1 \vdash e_1 : A \qquad \Sigma; \Gamma_2, x{:}A \vdash e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : B}\ (\text{S:LET}) \qquad \frac{B = (B_1, B_2)}{\Sigma; x_1{:}B_1, x_2{:}B_2 \vdash (x_1, x_2) : B}\ (\text{S:PAIR})$$

$$\frac{}{\Sigma; \emptyset \vdash \mathit{nil} : L(A)}\ (\text{S:NIL}) \qquad \frac{A = (A_1, A_2) \qquad \Sigma; \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash e : B}{\Sigma; \Gamma, x{:}A \vdash \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e : B}\ (\text{S:MATP})$$

$$\frac{}{\Sigma; x_h{:}A, x_t{:}L(A) \vdash \mathit{cons}(x_h, x_t) : L(A)}\ (\text{S:CONS}) \qquad \frac{}{\Sigma; \emptyset \vdash \mathit{leaf} : T(A)}\ (\text{S:LEAF})$$

$$\frac{}{\Sigma; x_0{:}A, x_1{:}T(A), x_2{:}T(A) \vdash \mathit{node}(x_0, x_1, x_2) : T(A)}\ (\text{S:NODE})$$

$$\frac{\Sigma; \Gamma \vdash e_1 : B \qquad \Sigma; \Gamma, x_h{:}A, x_t{:}L(A) \vdash e_2 : B}{\Sigma; \Gamma, x{:}L(A) \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : B}\ (\text{S:MATL})$$

$$\frac{\Sigma; \Gamma \vdash e_1 : B \qquad \Sigma; \Gamma, x_0{:}A, x_1{:}T(A), x_2{:}T(A) \vdash e_2 : B}{\Sigma; \Gamma, x{:}T(A) \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 : B}\ (\text{S:MATT})$$

$$\frac{\Sigma; \Gamma \vdash e : B}{\Sigma; \Gamma, x{:}A \vdash e : B}\ (\text{S:AUGMENT}) \qquad \frac{\Sigma; \Gamma, x{:}A, y{:}A \vdash e : B}{\Sigma; \Gamma, z{:}A \vdash e[z/x, z/y] : B}\ (\text{S:SHARE})$$

Figure 3.1: Type rules for simple types.

RAML Programs

A *(well-typed) RAML program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash e_f{:}B$ if $\Sigma(f) = A \to B$.

I write $f(y_1, \ldots, y_k) = e'_f$ to indicate that $\Sigma(f) = (A_1, (A_2, (\ldots, A_k) \cdots) \to B$ and that $\Sigma; y_1{:}A_1, \ldots, y_k{:}A_k \vdash e'_f : B$. In this case, f is defined by $e_f = \text{match } y_f \text{ with } (y_1, y'_f) \to \text{match } y'_f \text{ with } (y_2, y''_f) \ldots e'_f$. Such function definitions are of course also included in the extended syntax of RAML that we use in the prototype implementation (see Chapter 7).

3.3 Resource-Aware Semantics

In this section, I formalize the call-by-value evaluation of RAML programs by defining an operational big-step semantics. I use big-step rather than (term-rewriting) small-step semantics because I think that it is more natural and better agrees with actual behaviour of computers. Moreover, it is preferable to work with big-step semantics in the context of program analysis since it allows for shorter, less syntactic proofs.

In Section 3.3.1, I define a classic (inductive) operational big-step semantics for RAML which is annotated with a counter to monitor the resource usage during the evaluation. In Section 3.3.2, I define the notion of a well-formed environment that is used in some theorems.

A dissatisfying feature of classical big-step semantics is that it does not provide evaluation judgments for non-terminating evaluations. As a result, the soundness theorems for amortized resource analyses have in the past been formulated for terminating evaluations only [HJ03, JLH+09, JHLH10].

To address that issue, Section 3.3.3 contains a novel big-step operational semantics for partial evaluations which agrees with the usual big-step semantics on terminating computations. In this way, we retain the advantages of big-step semantics while capturing the resource behaviour of non-terminating programs. This enables the proof of an improved soundness result (see, i.e., Chapter 4): if the type analysis has established a resource bound for an expression then the resource consumption of its (possibly non-terminating) evaluation does not exceed the bound. It follows that run-time bounds also ensure termination.

3.3.1 Big-Step Operational Semantics

In the following, I define a big-step operational semantics that measures the quantitative resource consumption of programs. It is parametric in the resource of interest and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. The actual constants for a step on a specific system architecture can be derived by analyzing the translation of the step in the compiler implementation for that architecture [JLH+09].

The semantics is formulated with respect to a stack and a heap as usual: Let *Loc* be an infinite set of *locations* modeling memory addresses on a heap. The set of RAML *values Val* is given by

$$v ::= \ell \mid b \mid n \mid \text{NULL} \mid (v_1, v_2)$$

A value $v \in Val$ is either a location $\ell \in Loc$, a boolean constant b, an integer n, a null value NULL or a pair of values (v_1, v_2). I identify the tuple $(v_1, \ldots, v_n)$ with the pair $(v_1, (v_2, \cdots) \cdots)$.

A *heap* is a finite partial mapping $H : Loc \to Val$ that maps locations to values. A *stack* is a finite partial mapping $V : \text{VID} \to Val$ from variable identifiers to values.

Since we also consider resources like memory that can become available during an evaluation, we have to track the *watermark* of the resource usage, that is, the maximal number of resource units that are simultaneously used during an evaluation. To derive a watermark of a sequence of evaluations from the watermarks of the sub evaluations, you have also to take into account the number of resource units that are available after each sub evaluation.

The operational evaluation rules in Figures 3.2 and 3.3 thus define an evaluation judgment of the form

$$V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$$

expressing the following. If the stack V and the initial heap H are given then the expression e evaluates to the value v and the new heap H'. In order to evaluate e one needs at least $q \in \mathbb{Q}^+$ resource units and after the evaluation there are at least $q' \in \mathbb{Q}^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available during the execution of e.

In contrast to similar versions in earlier works there is at most one pair (q, q') such that $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ for a given expression e, a heap H and a stack V. The non-negative number q is the watermark of resources that are used simultaneously during the evaluation.

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid[1] $\mathcal{Q} = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, \cdot)$. The neutral element is $(0, 0)$ which means that resources are neither used nor restituted. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q',\ p') & \text{if } q' \le p \\ (q,\ p' + q' - p) & \text{if } q' > p \end{cases}$$

The intuition is that you need q resource units to perform the first evaluation and after the evaluation q' restituted units remain. Now you have to pay for the second operation which needs p units. If $q' \le p$ then you additionally need $p - q'$ resources to pay for both evaluations and have p' resources left in the end. If $q' > p$ then q units suffice

[1] In fact, it is possible to define the evaluation more abstractly with respect to an arbitrary monoid M.

$$\frac{x \in \mathrm{dom}(V)}{V, H \vdash x \rightsquigarrow V(x), H \mid K^{\mathrm{var}}}\ (\text{E:VAR}) \qquad \frac{}{V, H \vdash () \rightsquigarrow \text{NULL}, H \mid K^{\mathrm{unit}}}\ (\text{E:CONSTU})$$

$$\frac{n \in \mathbb{Z}}{V, H \vdash n \rightsquigarrow n, H \mid K^{\mathrm{int}}}\ (\text{E:CONSTI}) \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{V, H \vdash b \rightsquigarrow b, H \mid K^{\mathrm{bool}}}\ (\text{E:CONSTB})$$

$$\frac{V(x) = v' \qquad [y_f \mapsto v'], H \vdash e_f \rightsquigarrow v, H' \mid (q, q')}{V, H \vdash f(x) \rightsquigarrow v, H' \mid K_1^{\mathrm{app}} \cdot (q, q') \cdot K_2^{\mathrm{app}}}\ (\text{E:APP})$$

$$\frac{x_1, x_2 \in \mathrm{dom}(V) \qquad v = op(V(x_1), V(x_2))}{V, H \vdash x_1\ op\ x_2 \rightsquigarrow v, H \mid K^{op}}\ (\text{E:BINOP})$$

$$\frac{V(x) = \mathit{True} \qquad V, H \vdash e_t \rightsquigarrow v, H' \mid (q, q')}{V, H \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \rightsquigarrow v, H' \mid K_1^{\mathrm{conT}} \cdot (q, q') \cdot K_2^{\mathrm{conT}}}\ (\text{E:CONDT})$$

$$\frac{V(x) = \mathit{False} \qquad V, H \vdash e_f \rightsquigarrow v, H' \mid (q, q')}{V, H \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \rightsquigarrow v, H' \mid K_1^{\mathrm{conF}} \cdot (q, q') \cdot K_2^{\mathrm{conF}}}\ (\text{E:CONDF})$$

$$\frac{V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (q, q') \qquad V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (p, p')}{V, H \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \rightsquigarrow v_2, H_2 \mid K_1^{\mathrm{let}} \cdot (q, q') \cdot K_2^{\mathrm{let}} \cdot (p, p') \cdot K_3^{\mathrm{let}}}\ (\text{E:LET})$$

$$\frac{x_1, x_2 \in \mathrm{dom}(V) \qquad v = (V(x_1), V(x_2))}{V, H \vdash (x_1, x_2) \rightsquigarrow v, H \mid K^{\mathrm{pair}}}\ (\text{E:PAIR})$$

$$\frac{V(x) = (v_1, v_2) \qquad V[x_1 \mapsto v_1, x_2 \mapsto v_2], H \vdash e \rightsquigarrow v, H' \mid (q, q')}{V, H \vdash \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e \rightsquigarrow v, H' \mid K_1^{\mathrm{matP}} \cdot (q, q') \cdot K_2^{\mathrm{matP}}}\ (\text{E:MATP})$$

Figure 3.2: Rules of the big-step operational semantics (1 of 2).

$$\frac{}{V, H \vdash \mathit{nil} \rightsquigarrow \textsc{Null}, H \mid K^{\text{nil}}} \text{ (E:Nil)}$$

$$\frac{x_h, x_t \in \text{dom}(V) \qquad v = (V(x_h), V(x_t)) \qquad \ell \notin \text{dom}(H)}{V, H \vdash \mathit{cons}(x_h, x_t) \rightsquigarrow \ell, H[\ell \mapsto v] \mid K^{\text{cons}}} \text{ (E:Cons)}$$

$$\frac{V(x) = \textsc{Null} \qquad V, H \vdash e_1 \rightsquigarrow v, H' \mid (q, q')}{\begin{array}{c} V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \\ \rightsquigarrow v, H' \mid K_1^{\text{matN}} \cdot (q, q') \cdot K_2^{\text{matN}} \end{array}} \text{ (E:MatNil)}$$

$$\frac{V(x)=\ell \qquad H(\ell)=(v_h, v_t) \qquad V[x_h \mapsto v_h, x_t \mapsto v_t], H \vdash e_2 \rightsquigarrow v, H' \mid (q, q')}{\begin{array}{c} V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \\ \rightsquigarrow v, H' \mid K_1^{\text{matC}} \cdot (q, q') \cdot K_2^{\text{matC}} \end{array}} \text{ (E:MatCons)}$$

$$\frac{}{V, H \vdash \mathit{leaf} \rightsquigarrow \textsc{Null}, H \mid K^{\text{leaf}}} \text{ (E:Leaf)}$$

$$\frac{x_0, x_1, x_2 \in \text{dom}(V) \qquad v = (V(x_0), V(x_1), V(x_2)) \qquad \ell \notin \text{dom}(H)}{V, H \vdash \mathit{node}(x_0, x_1, x_2) \rightsquigarrow \ell, H[\ell \mapsto v] \mid K^{\text{node}}} \text{ (E:Node)}$$

$$\frac{V(x) = \textsc{Null} \qquad V, H \vdash e_1 \rightsquigarrow v, H' \mid (q, q')}{\begin{array}{c} V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 \\ \rightsquigarrow v, H' \mid K_1^{\text{matTL}} \cdot (q, q') \cdot K_2^{\text{matTL}} \end{array}} \text{ (E:MatLeaf)}$$

$$\frac{\begin{array}{c} V(x) = \ell \\ H(\ell) = (v_0, v_1, v_2) \qquad V[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2], H \vdash e_2 \rightsquigarrow v, H' \mid (q, q') \end{array}}{\begin{array}{c} V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 \\ \rightsquigarrow v, H' \mid K_1^{\text{matTN}} \cdot (q, q') \cdot K_2^{\text{matTN}} \end{array}} \text{ (E:MatNode)}$$

Figure 3.3: Rules of the big-step operational semantics (2 of 2).

to perform both evaluations. Additionally, the $q' - p$ units that are not needed for the second evaluation are added to the resources becoming finally available.

The following facts are often used in proofs.

Proposition 3.3.1 Let $(q, q') = (r, r') \cdot (s, s')$.

1. $q \geq r$ and $q - q' = r - r' + s - s'$
2. If $(p, p') = (\bar{r}, r') \cdot (s, s')$ and $\bar{r} \geq r$ then $p \geq q$ and $p' = q'$
3. If $(p, p') = (r, r') \cdot (\bar{s}, s')$ and $\bar{s} \geq s$ then $p \geq q$ and $p' \leq q'$
4. $(r, r') \cdot ((s, s') \cdot (t, t')) = ((r, r') \cdot (s, s')) \cdot (t, t')$

If resources are never restituted (as with time) then we can restrict ourselves to elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$.

I identify (positive and negative) rational numbers with elements of $\mathcal{Q}$ as follows: $q \geq 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$. This notation avoids case distinctions in the evaluation rules since the constants K that appear in the rules can be negative. Then resources are restituted during an evaluation step. This is the case for stack space and also for heap space in a destructive pattern match which is omitted here for simplicity.

The evaluation rules are standard apart from the resource information that measure the resource consumption. These resource annotations are very similar in each rule and I explain them for the rules E:VAR and E:CONDT.

Assume that the resource cost for looking up the value of a variable on the stack and copying it to some register is $K^{\text{var}} \geq 0$. The rule E:VAR then states that the resource consumption of the evaluation of a variable is $(K^{\text{var}}, 0)$. So the watermark of the resource consumption is K^{var} and there are no resources left after the evaluation. If $K^{\text{var}} < 0$ then E:VAR states that the resource consumption of the evaluation of a variable is $(0, -K^{\text{var}})$. So the watermark is zero and after the evaluation there are K^{var} resources available.

Now consider the rule E:CONDT. Assume that the resource cost of looking up the value of the variable x and jumping to the source code of e_t is $K_1^{\text{conT}} \geq 0$. Assume furthermore that the jump back to the code after the conditional costs $K_2^{\text{conT}} \geq 0$ resources. Then the rule E:CONDT states that the cost for the evaluation of are $(K_1^{\text{conT}}, 0) \cdot (q, q') \cdot (K_2^{\text{conT}}, 0)$ if the watermark for the evaluation of e_t is q and if there are q' resources left after the evaluation. There are two cases. If $q' \geq K_2^{\text{conT}}$ then the overall watermark of the evaluation is $q + K_1^{\text{conT}}$ and there are $q' - K_2^{\text{conT}}$ resources available after the evaluation. If $q' < K_2^{\text{conT}}$ then the overall watermark of the evaluation is $q + K_1^{\text{conT}} + K_2^{\text{conT}} - q'$ and there are zero resources available after the evaluation. The statement is similar for negative constants K_i^{conT}.

The values of the constants $K_i^x \in \mathbb{Q}$ in the rules depend on the resource, the implementation and the system architecture. In fact, the value of a constant can also be a function of the type of a subexpression. For instance, the size of a cons cell depends on the size of the value that is stored in the cell in our implementation. Since the types of all subexpressions are available at compile time, this is a straightforward extension.

$$\frac{v \in \{True, False\}}{H \vDash v \mapsto v : bool}\ (\text{V:BOOL}) \qquad \frac{v \in \mathbb{N}}{H \vDash v \mapsto v : int}\ (\text{V:INT}) \qquad \frac{v = \text{NULL}}{H \vDash v \mapsto () : unit}\ (\text{V:UNIT})$$

$$\frac{v = (v_1, v_2) \qquad H \vDash v_1 \mapsto a_1 : A_1 \qquad H \vDash v_2 \mapsto a_2 : A_2}{H \vDash v \mapsto (a_1, a_2) : (A_1, A_2)}\ (\text{V:PAIR})$$

$$\frac{v = \text{NULL} \qquad A \in \mathscr{A}}{H \vDash v \mapsto [] : L(A)}\ (\text{V:NIL}) \qquad \frac{v = \text{NULL} \qquad A \in \mathscr{A}}{H \vDash v \mapsto leaf : T(A)}\ (\text{V:LEAF})$$

$$\frac{\begin{array}{c} v \in Loc \qquad H(v) = (v_1, v_2) \\ H' = H \setminus v \qquad H' \vDash v_1 \mapsto a_1 : A \qquad H' \vDash v_2 \mapsto [a_2, \ldots, a_n] : L(A) \end{array}}{H \vDash v \mapsto [a_1, \ldots, a_n] : L(A)}\ (\text{V:CONS})$$

$$\frac{\begin{array}{c} v \in Loc \qquad H(v) = (v_0, v_1, v_2) \\ H' = H \setminus v \qquad H' \vDash v_0 \mapsto a : A \qquad H' \vDash v_1 \mapsto t_1 : T(A) \qquad H' \vDash v_2 \mapsto t_2 : T(A) \end{array}}{H \vDash v \mapsto tree(a, t_1, t_2) : T(A)}\ (\text{V:NODE})$$

Figure 3.4: Relating heap cells to semantic values.

Actual constants for stack-space, heap-space and clock-cycle consumption were determined for the abstract machine of the language Hume [HM03] for the Renesas M32C/85U architecture. A list can be found in the literature [JLH+09].

The following proposition states that heap cells are never changed during an evaluation after they have been allocated. This is a convenient property to simplify some of the later proofs but it is not necessarily needed. We would not have this property if we would include an destructive pattern matching in RAML. How to formally deal with it is described in the literature [JLH+09].

Proposition 3.3.2 Let e be an expression, V be a stack, and H be a heap. If $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ then $H'(\ell) = H(\ell)$ for all $\ell \in \text{dom}(H)$.

PROOF The only rules that allocate new heap cells are E:CONS and E:NODE. And in these rules we have the side condition $\ell \notin H$ that prevents an old location from being changed by assigning a value to ℓ. ∎

3.3.2 Well-Formed Environments

The notion of a well-formed environment is used in many of the following theorems. Intuitively, a heap and stack are well-formed with respect to some typing context if for each variable, the type assigned by the typing context agrees with the actual value assigned to the variable by the stack and the heap.

If H is a heap, v is a value, A is a type, and $a \in [\![A]\!]$ then I write $H \vDash v \rightarrowtail a : A$ to mean that v defines the semantic value $a \in [\![A]\!]$ when pointers are followed in H in the obvious way. The judgment is formally defined in Figure 3.4.

I write [] for the empty list. For a non-empty list $[a_1, \ldots, a_n]$ I write $[a_1, \ldots, a_n] = a_1 :: [a_2, \ldots, a_n]$. The tree with root a, left subtree t_1 and right subtree t_2 is denoted by $\mathit{tree}(a, t_1, t_2)$. The empty tree is denoted by leaf. For a heap H, I write $H' = H \backslash \ell$ for the heap in which the location ℓ is removed. That is, $\mathrm{dom}(H') = \mathrm{dom}(H) \backslash \{\ell\}$ and $H'(\ell') = H(\ell')$ for all $\ell' \in \mathrm{dom}(H')$.

Note that there exist three semantic values a such that $H \vDash \textsc{Null} \rightarrowtail a : A$ for every heap H; namely $a = ()$, $a = []$, and $a = \mathit{leaf}$. However, if we fix a data type A then the semantic value a is unique.

Proposition 3.3.3 Let H be a heap, v be a value, and let A be a data type. If $H \vDash v \rightarrowtail a : A$ and $H \vDash v \rightarrowtail a' : A$ then $a = a'$.

Proof We prove the claim by induction on the derivation of $H \vDash v \rightarrowtail a : A$.

Assume first that $H \vDash v \rightarrowtail a : A$ has been derived by the application of a single rule. Then the judgment has been derived by one of the rules V:Bool, V:Int, V:Unit, V:Nil, or V:Leaf. An inspection of the rules shows that for given A and v only one of rules is applicable. Thus it follows that $a = a'$.

Assume now that the derivation of $H \vDash v \rightarrowtail a : A$ ends with an application of the rule V:Cons. Then $A = L(B)$, $a = [a_1, \ldots, a_n]$, $v \in \mathit{Loc}$, and $H(v) = (v_1, v_2)$. It follows that the derivation of $H \vDash v \rightarrowtail a' : A$ also ends with an application of V:Cons. Thus we have $a' = [b_1, \ldots, b_m]$. From the premises of V:Cons it follows that

$$\begin{array}{lcl} H' \vDash v_1 & \rightarrowtail & a_1 : A \\ H' \vDash v_2 & \rightarrowtail & [a_2, \ldots, a_n] : L(A) \\ H' \vDash v_1 & \rightarrowtail & b_1 : A \\ H' \vDash v_2 & \rightarrowtail & [b_2, \ldots, b_m] : L(A) \end{array}$$

where $H' = H \backslash v$. It follows by induction that $n = m$ and $b_i = a_i$ for all $1 \leq i \leq n$.

The cases in which the derivation ends with the V:Node or V:Pair are similar. ∎

Note that if $H \vDash v \rightarrowtail a : A$ then v may well point to a data structure with some aliasing, but no circularity is allowed since this would require infinite values a. I do not include them because in our functional language there is no way of generating such values.

I write $H \vDash v : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $H \vDash v \rightarrowtail a : A$. A stack V and a heap H are *well-formed* with respect to a context Γ if $H \vDash V(x) : \Gamma(x)$ holds for every $x \in \mathrm{dom}(\Gamma)$. I then write $H \vDash V : \Gamma$.

Theorem 3.3.4 shows that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment.

Theorem 3.3.4 If $\Sigma; \Gamma \vdash e : B$, $H \vDash V : \Gamma$ and $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ then $H' \vDash V : \Gamma$ and $H' \vDash v : B$.

PROOF From Proposition 3.3.2 it follows $H'(\ell) = H(\ell)$ for all $\ell \in \mathrm{dom}(H)$ and thus $H' \vDash V{:}\Gamma$.

The second part, $H' \vDash v : B$, is proved by induction on the derivations of $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ and $\Sigma; \Gamma \vdash e : B$ where the induction on the evaluation judgment takes priority.

Note that a single induction on the derivation of the evaluation judgment fails because of the structural type rules S:SHARE and S:AUGMENT. If the type derivation ends with one of these rules then you do not obtain type judgments that correspond to the premises of the last evaluation rule. As a result, you can not apply the induction hypothesis.

A single induction on the derivation of the type judgment $\Sigma; \Gamma \vdash e : B$ fails because of the type rule S:APP and the corresponding evaluation rule E:APP. On the one hand, the evaluation of a function application proceeds with the evaluation of the body of the function. On the other hand, a type derivation that ends with S:APP consists of one step only. To apply the induction hypothesis the evaluation of e_f, you need to use a type derivation of e_f which is longer then zero steps. Thus the induction hypothesis can not be applied.

(S:SHARE) Suppose that the derivation of $\Sigma; \Gamma \vdash e : B$ ends with an application of the rule S:SHARE. Then $\Gamma = \Gamma, z{:}A$ and it follows from the premise that

$$\Sigma; \Gamma', x{:}A, y{:}A \vdash e' : B \tag{3.1}$$

for some data type A, a context Γ' and an expression e' with $e'[z/x, z/y] = e$. Since $H \vDash V : \Gamma', z{:}A$ and

$$V, H \vdash e \rightsquigarrow v, H' \mid (q, q') \tag{3.2}$$

it follows that $H \vDash V_{xy} : \Gamma', x{:}A, y{:}A$ and

$$V_{xy}, H \vdash e' \rightsquigarrow v, H' \mid (q, q') \tag{3.3}$$

for $V_{xy} = V \setminus z \cup \{x \mapsto V(z), y \mapsto V(z)\}$. Furthermore, the derivation tree of (3.3) has the same shape as the derivation tree of (3.2). Thus we can apply the induction hypothesis to (3.1) and (3.2), and derive $H' \vDash v : B$.

(S:AUGMENT) If the derivation of $\Sigma; \Gamma \vdash e : B$ ends with an application of the rule S:AUGMENT then we have

$$\Sigma; \Gamma' \vdash e : B \tag{3.4}$$

for a context Γ' with $\Gamma', x{:}A = \Gamma$. But it follows by definition that $H \vDash V : \Gamma'$. Thus we can apply the induction hypothesis to (3.4) and the evaluation judgment, and derive $H' \vDash v : B$.

(S:VAR) If the type derivation ends with the application of the rule S:VAR then the derivation of the evaluation judgment ends with and application of E:VAR. The claim $H' \vDash V(x) : \Gamma(x)$ follows from $H \vDash V : \Gamma'$,and $H' = H$.

(S:CONST*) Assume that the type derivation ends with one of rules (S:CONST*) for constants. Then the derivation of the evaluation judgment ends with an application of the corresponding rule E:CONST*. The claim follows directly from the definition.
(S:OPINT) The evaluation ends with an application of the rule E:BINOP. Since we have $\Sigma; x_1{:}int, x_2{:}int \vdash x_1\ op\ x_2 : int$ and $H \vDash V : x_1{:}int, x_2{:}int$ it follows that $V, H \vdash e \rightsquigarrow n, H' \mid (q, q')$ for an integer n; thus $H' \vDash n : int$.
(S:OPBOOL) Similar to the case (S:OPINT).
(S:APP) Assume the type derivation ends with the derivation of $\Sigma; x{:}A \vdash f(x) : B$, using the rule S:APP. Then the derivation of the evaluation judgment ends with an application of the rule E:APP. From the premise $\Sigma(f) = A \to B$ of S:APP it follows that $\Sigma; y_f{:}A \vdash e_f{:}B$. Since $H \vDash V(x) : A$ we have $H \vDash [y_f \mapsto H(x)] : (y_f{:}A)$. Thus we can apply the induction hypothesis to the premise $[y_f \mapsto H(x)], H \vdash e_f \rightsquigarrow v, H' \mid (q, q')$ of the rule E:APP. It follows that $H' \vDash v : B$.
(S:COND) Then the evaluation ends with an application of the rules E:CONDT or E:CONDF. Assume it ends with E:CONDT; the case E:CONF is similar. We use the premise $\Sigma; \Gamma \vdash e_t : B$ of S:COND and the fact $H \vDash V : \Gamma$ to apply the induction hypothesis to the premise $V, H \vdash e_t \rightsquigarrow v, H' \mid (q, q')$ of E:CONDT. It follows that $H' \vDash v : B$.
(S:LET) Then the derivation of the evaluation judgment ends with an application of the rule E:LET. We have $\Sigma; \Gamma_1 \vdash e_1 : A$ from the premises of S:LET and also $H \vDash V : \Gamma_1$ from $H \vDash V : \Gamma$. So we can apply the induction hypothesis to $V, H_1 \vdash e_1 \rightsquigarrow v_1, H_1 \mid (q, q')$ and derive $H_1 \vDash v_1 : A$. From Proposition 3.3.2 it follows that $H_1(l) = H(l)$ for all $l \in \mathrm{dom}(H)$. Since $H_1 \vDash v_1 : A$ and $\mathrm{img}(V) \subseteq \mathrm{dom}(H)$ we conclude $H_1 \vDash V[x \mapsto v_1] : \Gamma, x{:}A$. Furthermore, $\Sigma; \Gamma_2, x{:}A \vdash e_2 : B$ is a premise of S:LET. Thus we can apply the induction hypothesis a second time to $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (p, p')$ and derive $H_2 \vDash v_2 : B$.
(S:PAIR) Then the evaluation ends with an application of the rule E:PAIR. We conclude from $H \vDash V : (x_1{:}B_1, x_2{:}B_2)$ that $H \vDash (V(x_1), V(x_2)) : (B_1, B_2)$ (using V:PAIR).
(S:MATP) Then the evaluation ends with an application of the rule E:MATP. Since $H \vDash V : \Gamma, x{:}(A_1, A_2)$ it follows that $H \vDash (v_1, v_2){:}(A_1, A_2)$ and thus $H \vDash v_1{:}A_1$ and $H \vDash v_2{:}A_2$ where $V(x) = (v_1, v_2)$. We conclude that $H \vDash V[x_1 \mapsto v_1, x_2 \mapsto v_2] : \Gamma, x_1{:}A_1, x_2{:}A_2$. Furthermore we have the premise $\Sigma; \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash e : B$ in the rule S:MATP. Hence we can apply the induction hypothesis the premise $V[x_1 \mapsto v_1, x_2 \mapsto v_2], H \vdash e \rightsquigarrow v, H' \mid (q, q')$ of E:MATP. It follows that $H' \vDash v : B$.
(S:NIL) and (S:LEAF) Then the corresponding evaluation rules E:NIL or E:LEAF have been applied to derive the evaluation judgment. The claim follows directly from the definition.
(S:CONS) and (S:NODE) Similar to the case (S:PAIR).
(S:MATL) and (S:MATT) Similar to the case (S:MATP). ∎

3.3.3 Partial Big-Step Operational Semantics

A general shortcoming of classic big-step operational semantics is that it does not provide judgments for evaluations that diverge. This is problematic if one intends to prove statements for all computations (divergent and convergent) that do not go wrong.

A straightforward remedy is to use a small-step semantics to describe computations. But in the context of resource analysis, the use of big-step rules seems to be more favorable. Firstly, big-step rules can more directly axiomatize the resource behavior of compiled code on specific machines. Secondly, it allows for shorter and less syntactic proofs.

Another classic approach [CC92, Ler06] is to add divergence rules to the operational semantics that are interpreted coinductively. But then one loses the ability to prove statements by induction on the evaluation which is crucial for the proof of the soundness theorems of the analysis systems (see Chapters 4, 5, and 6). It should also be possible to work with a coinductive definition in the style of Cousot or Leroy [CC92, Ler06]. However, coinductive semantics leans itself less well to formulating and proving semantic soundness theorems of the form "if the program is well-typed and the operational semantics says X then Y holds". For example, in Leroy's Lemmas 17-22 [Ler06] the coinductive definition appears in the conclusion rather than as a premise.

That is why I use a novel approach to the problem here by defining a *big-step semantics for partial evaluations* that directly corresponds to the rules of the big-step semantics in Figures 3.2 and 3.3. The rules in Figures 3.5 and 3.6 define a judgment of the form

$$V, H \vdash e \rightsquigarrow \mid q$$

where V is a stack, H is a heap, $q \in \mathbb{Q}_0^+$, and e is an expression. The meaning is that there is a partial evaluation of e with the initial stack V and the initial heap H that consumes q resources. Here, q is the watermark of the resource usage. We do not have to keep track of the restituted resources since partial evaluations are composed of complete evaluations only.

Since there might be negative constants K, the partial evaluation rules have conclusions of the form $V, H \vdash e \rightsquigarrow \mid \max(q, 0)$ to ensure non-negative values. For simplicity, I just write $V, H \vdash e \rightsquigarrow \mid q$ instead of $V, H \vdash e \rightsquigarrow \mid \max(q, 0)$ in each conclusion of the rules in Figures 3.5 and 3.6.

Note that the rule P:ZERO is essential for the partiality of the semantics. It can be applied at any point to stop the evaluation and thus yields to a non-deterministic evaluation judgment. I explain the other rules with three representative examples.

The rule P:VAR can be understood as follows. To partially evaluate a variable, you can only do one evaluation step, namely evaluating the variable thereby producing the cost K^{var} if $K^{\text{var}} > 0$ and zero cost otherwise.

The rule P:LET1 can be read as follows. If there is a partial evaluation of e_1 that needs q resources then you can partially evaluate *let* $x = e_1$ *in* e_2 by starting the evaluation of the let expression which costs $K_1^{\text{let}} \geq 0$ or reimburses $K_1^{\text{let}} < 0$ resources. Then you can partially evaluate e_1, deriving a partial evaluation of the let expression that produces the watermark $K_1^{\text{let}} + q$.

Another way to partially evaluate the let expression *let* $x = e_1$ *in* e_2 is to use the rule P:LET2. There we completely evaluate e_1 measuring the resource consumption (q, q'). Then we partially evaluate e_2 using p resources. Then we compose the two

$$\frac{}{V, H \vdash e \rightsquigarrow \mid 0} \text{ (P:ZERO)} \qquad \frac{}{V, H \vdash () \rightsquigarrow \mid K^{\text{unit}}} \text{ (P:CONSTU)}$$

$$\frac{b \in \{True, False\}}{V, H \vdash b \rightsquigarrow \mid K^{\text{bool}}} \text{ (P:CONSTB)} \qquad \frac{n \in \mathbb{Z}}{V, H \vdash n \rightsquigarrow \mid K^{\text{int}}} \text{ (P:CONSTI)}$$

$$\frac{x \in \text{dom}(V)}{V, H \vdash x \rightsquigarrow \mid K^{\text{var}}} \text{ (P:VAR)} \qquad \frac{V(x) = v \qquad [y_f \mapsto v], H \vdash e_f \rightsquigarrow \mid q}{V, H \vdash f(x) \rightsquigarrow \mid K_1^{\text{app}} + q} \text{ (P:APP)}$$

$$\frac{x_1, x_2 \in \text{dom}(V)}{V, H \vdash x_1 \; op \; x_2 \rightsquigarrow \mid K^{op}} \text{ (P:BINOP)} \qquad \frac{V, H \vdash e_1 \rightsquigarrow \mid q}{V, H \vdash let \; x = e_1 \; in \; e_2 \rightsquigarrow \mid K_1^{\text{let}} + q} \text{ (P:LET1)}$$

$$\frac{\begin{array}{c} V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (q, q') \\ V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow \mid p \qquad K_1^{\text{let}} \cdot (q, q') \cdot K_2^{\text{let}} \cdot (p, 0) = (r, r') \end{array}}{V, H \vdash let \; x = e_1 \; in \; e_2 \rightsquigarrow \mid r} \text{ (P:LET2)}$$

$$\frac{V(x) = True \qquad V, H \vdash e_t \rightsquigarrow \mid q}{V, H \vdash if \; x \; then \; e_t \; else \; e_f \rightsquigarrow \mid K_1^{\text{conT}} + q} \text{ (P:CONDT)}$$

$$\frac{V(x) = False \qquad V, H \vdash e_f \rightsquigarrow \mid q}{V, H \vdash if \; x \; then \; e_t \; else \; e_f \rightsquigarrow \mid K_1^{\text{conF}} + q} \text{ (P:CONDF)} \qquad \frac{}{V, H \vdash nil \rightsquigarrow \mid K^{\text{nil}}} \text{ (P:NIL)}$$

$$\frac{x_1, x_2 \in \text{dom}(V)}{V, H \vdash (x_1, x_2) \rightsquigarrow \mid K^{\text{pair}}} \text{ (P:PAIR)} \qquad \frac{x_h, x_t \in \text{dom}(V)}{V, H \vdash cons(x_h, x_t) \rightsquigarrow \mid K^{\text{cons}}} \text{ (P:CONS)}$$

$$\frac{V(x) = (v_1, v_2) \qquad V[x_1 \mapsto v_1, x_2 \mapsto v_2], H \vdash e \rightsquigarrow \mid q}{V, H \vdash match \; x \; with \; (x_1, x_2) \rightarrow e \rightsquigarrow \mid K_1^{\text{matP}} + q} \text{ (P:MATP)}$$

$$\frac{V(x) = \text{NULL} \qquad V, H \vdash e_1 \rightsquigarrow \mid q}{V, H \vdash match \; x \; with \mid nil \rightarrow e_1 \mid cons(x_h, x_t) \rightarrow e_2 \rightsquigarrow \mid K_1^{\text{matN}} + q} \text{ (P:MATNIL)}$$

$$\frac{V(x) = \ell \qquad H(\ell) = (v_h, v_t) \qquad V[x_h \mapsto v_h, x_t \mapsto v_t], H \vdash e_2 \rightsquigarrow \mid q}{V, H \vdash match \; x \; with \mid nil \rightarrow e_1 \mid cons(x_h, x_t) \rightarrow e_2 \rightsquigarrow \mid K_1^{\text{matC}} + q} \text{ (P:MATCONS)}$$

Figure 3.5: Partial big-step operational semantics (1 of 2).

$$\frac{}{V, H \vdash \mathit{leaf} \rightsquigarrow \mid K^{\text{leaf}}} \text{(P:LEAF)} \qquad \frac{x_0, x_1, x_2 \in \text{dom}(V)}{V, H \vdash \mathit{node}(x_0, x_1, x_2) \rightsquigarrow \mid K^{\text{node}}} \text{(P:NODE)}$$

$$\frac{V(x) = \text{NULL} \qquad V, H \vdash e_1 \rightsquigarrow \mid q}{V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \rightarrow e_1 \mid \mathit{node}(x_0, x_1, x_2) \rightarrow e_2 \rightsquigarrow \mid K_1^{\text{matTL}} + q} \text{(P:MATLEAF)}$$

$$\frac{\begin{array}{c} V(x) = \ell \\ H(\ell) = (v_0, v_1, v_2) \qquad V[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2], H \vdash e_2 \rightsquigarrow \mid q \end{array}}{V, H \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \rightarrow e_1 \mid \mathit{node}(x_0, x_1, x_2) \rightarrow e_2 \rightsquigarrow \mid K_1^{\text{matTN}} + q} \text{(P:MATNODE)}$$

Figure 3.6: Partial big-step operational semantics (2 of 2).

evaluations and obtain a partial evaluation for the let expression that uses r resources where $(r, r') = K_1^{\text{let}} \cdot (q, q') \cdot K_2^{\text{let}} \cdot (p, 0)$.

Theorem 3.3.5 proves that if an expression converges in a given environment then the resource-usage watermark of the evaluation is an upper bound for the resource usage of every partial evaluation of the expression in that environment.

Theorem 3.3.5 If $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ and $V, H \vdash e \rightsquigarrow \mid p$ then $p \leq q$.

PROOF By induction on the derivation D of the judgment $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$. To prove the *induction basis* let D consist of one step. Then e is a constant c, a variable x, a binary operation $x_1\ op\ x_2$, a pair (x_1, x_2), the constant *nil*, *leaf*, $\mathit{cons}(x_1, x_2)$, or $\mathit{node}(x_1, x_2, x_3)$. Let e be for instance a variable x. Then by definition of E:VAR it follows that $V, H \vdash e \rightsquigarrow v, H' \mid (K^{\text{var}}, 0)$ or $V, H \vdash e \rightsquigarrow v, H' \mid (0, -K^{\text{var}})$. Thus $q = \max(0, K^{\text{var}})$. The only P-rules that apply to x are P:VAR and P:ZERO. Thus it follows that if $V, H \vdash e \rightsquigarrow \mid p$ then then $p = \max(0, K^{\text{var}})$. The other cases are similar.

For the induction step assume that $|D| > 1$. Then e is a pattern match, a function application, a conditional, or a let expression. For instance, let e be the expression $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$. Then it follows from rule E:LET that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (q_1, q_1')$, $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (q_2, q_2')$ and

$$(q, q') = K_1^{\text{let}} \cdot (q_1, q_1') \cdot K_2^{\text{let}} \cdot (q_2, q_2') \cdot K_3^{\text{let}} \tag{3.5}$$

By induction we conclude

$$\text{if } V, H \vdash e_1 \rightsquigarrow \mid p_1 \text{ then } p_1 \leq q_1 \tag{3.6}$$

$$\text{if } V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow \mid p_2 \text{ then } p_2 \leq q_2 \tag{3.7}$$

Now let $V, H \vdash e \rightsquigarrow \mid p$. Then this judgment was derived via the rules P:LET1 or P:LET2. In the first case it follows by definition that $p = \max(p_1 + K_1^{\text{let}}, 0)$ for some p_1 and $p_1 \leq q_1$ by (3.6) and (3.5) that $p \leq q$.

If $V, H \vdash e \rightsquigarrow \mid p$ was derived by P:LET2 then it follows that $(p, p') = K_1^{\text{let}} \cdot (q_1, q_1') \cdot K_2^{\text{let}} \cdot (p_2, 0)$ for some p', p_2. We conclude from (3.7) that $p_2 \leq q_2$ and hence from Proposition 3.3.1 and (3.5) $p \leq q$. The other cases are similar to the case P:LET1. ∎

Theorem 3.3.9 states that, in a well-formed environment, every well-typed expression either diverges or evaluates to a value of the stated type. To this end we instantiate the resource constants in the rules to count the number of evaluation steps.

Proposition 3.3.6 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. Let $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ and let the derivation of the judgment have n steps. Then $q = n$ and $q' = 0$.

PROOF By induction on the derivation D of $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$.

If D consists of only one step ($|D| = 1$) then e is a constant c, a variable x, a binary operation x_1 *op* x_2, a pair (x_1, x_2), the constant *nil*, *leaf*, *cons*(x_1, x_2), or *node*(x_1, x_2, x_3). In each case, $q = 1$ and $q' = 0$ follows immediately from the respective evaluation rule.

Now let $|D| > 1$. Then e is a pattern match, a function application, a conditional, or a let expression. For instance, let e be the expression *let* $x = e_1$ *in* e_2. Then it follows from rule E:LET that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (q_1, q_1')$, $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (q_2, q_2')$ and

$$(q, q') = 1 \cdot (q_1, q_1') \cdot 0 \cdot (q_2, q_2') \cdot 0 = (1 + q_1, q_1') \cdot (q_2, q_2')$$

Let n_1 be the evaluation steps needed by e_1 and let n_2 be the number of evaluation steps needed by e_2. By induction it follows that $q_1 = n_1$, $q_2 = n_2$ and $q_1' = q_2' = 0$. Thus $q = n_1 + n_2 + 1 = n$.

The other cases are similar. ∎

The following lemma shows that if there is a complete evaluation that uses n steps then there are partial evaluations that use i steps for $0 \leq i \leq n$. It is used in the proof of Theorem 3.3.9 with $i = n$.

Lemma 3.3.7 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$ then $V, H \vdash e \rightsquigarrow \mid i$ for every $0 \leq i \leq n$.

PROOF By induction on the derivation D of $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$. The proof is very similar to the proof of Theorem 3.3.5. ∎

Lemma 3.3.8 proves that you can always make one partial evaluation step for a well-typed expression in a well-formed environment. It is used in the induction basis of the proof of Theorem 3.3.9.

Lemma 3.3.8 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $\Sigma; \Gamma \vdash e : A$, $H \vDash V : \Gamma$ then $V, H \vdash e \rightsquigarrow \mid 1$.

PROOF By case distinction on e. The proof is straightforward so I only demonstrate two characteristic cases.

Let e for instance be a variable x. Then it follows from $\Sigma;\Gamma \vdash x : A$ and $H \vDash V : \Gamma$ that $x \in V$. Thus $V, H \vdash x \rightsquigarrow \mid 1$ by (P:VAR).

Let e now be a conditional *if x then* e_t *else* e_f. Then it follows from $\Sigma;\Gamma \vdash e : A$ and $H \vDash V : \Gamma$ that $V(x) \in \{True, False\}$. Furthermore, we derive $V, H \vdash e_t \rightsquigarrow \mid 0$ and $V, H \vdash e_f \rightsquigarrow \mid 0$ with the rule P:ZERO. Thus we can use either P:CONDT or P:CONDF to derive $V, H \vdash e \rightsquigarrow \mid 1$. ∎

Theorem 3.3.9 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_i^x = 0$ for all x and all $i > 1$. If $\Sigma;\Gamma \vdash e : A$ and $H \vDash V : \Gamma$ then $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$ for an $n \in \mathbb{N}$ or $V, H \vdash e \rightsquigarrow \mid m$ for every $m \in \mathbb{N}$.

PROOF We show by induction on n that if

$$\Sigma;\Gamma \vdash e : A,\ V, H \vdash e \rightsquigarrow \mid n \text{ and } H \vDash V : \Gamma \tag{3.8}$$

then $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$ or $V, H \vdash e \rightsquigarrow \mid n+1$. Then Theorem 3.3.9 follows since $V, H \vdash e \rightsquigarrow \mid 0$ for every V, H and e.

Induction basis $n = 0$: We use Lemma 3.3.8 to conclude from the well-formedness of the environment (3.8) that $V, H \vdash e \rightsquigarrow \mid 1$.

Induction step $n > 0$: Assume (3.8). If e is a constant c, a variable x, a binary operation x_1 *op* x_2, a pair (x_1, x_2), the constant *nil*, or $cons(x_1, x_2)$. Then $n = 1$ and we derive $V, H \vdash e \rightsquigarrow v, H' \mid (1, 0)$ immediately from the corresponding evaluation rule.

If e is a pattern match, a function application, a conditional, or a let expression then we use the induction hypothesis. Since the other cases are similar, we provide the argument only for the case where e is a let expression *let* $x = e_1$ *in* e_2. Then $V, H \vdash e \rightsquigarrow \mid n$ was derived via P:LET1 or P:LET2. In the case of P:LET1 it follows that $V, H \vdash e_1 \rightsquigarrow \mid n-1$. By the induction hypothesis we conclude that either $V, H \vdash e_1 \rightsquigarrow \mid n$ or $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (n-1, 0)$. In the first case we can use P:LET1 to derive $V, H \vdash e \rightsquigarrow \mid n+1$. In the second case it follows from Theorem 3.3.4 that $H_1 \vDash V : \Gamma$ and $H_1 \vDash v_1 : A$ and thus $H_1 \vDash V[x \mapsto v_1] : \Gamma, x : A$. We then apply Lemma 3.3.8 to obtain $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow \mid 1$. Therefore we can apply P:LET2 to derive $V, H \vdash e \rightsquigarrow \mid n+1$.

Assume now that e was derived by the use of P:LET2. Then it is true that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (n_1, 0)$ and $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow \mid n_2$ for some n_1, n_2 with $n_1 + n_2 + 1 = n$. From Theorem 3.3.4 it follows that $H_1 \vDash V[x \mapsto v_1] : \Gamma, x : A$. Therefore we can apply the induction hypothesis to infer that $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (n_2, 0)$ or $V[x \mapsto v_1], H_1 \vdash e_2 \rightsquigarrow \mid n_2 + 1$. In the first case we apply E:LET and derive $V, H \vdash e \rightsquigarrow v_2, H_2 \mid (n, 0)$. In the second case we apply P:LET2 and derive $V, H \vdash e \rightsquigarrow \mid n+1$. ∎

Cost-Free Metric

The type inference algorithm makes use of the *cost-free* resource metric. This is the metric in which all constants K that appear in the rules are instantiated to zero. I use it in

Chapters 5 and 6 to define a resource-polymorphic recursion that uses cost-free function types to pass potential from the argument to the result. The following proposition can be proved analogous to Proposition 3.3.6.

Proposition 3.3.10 Let all resource constants K be instantiated by $K = 0$. If $V, H \vdash e \rightsquigarrow v, H' \mid (q, q')$ then $q = q' = 0$. If $V, H \vdash e \rightsquigarrow \mid q$ then $q = 0$.

Elegance is not a dispensable luxury but a quality that decides between success and failure.

EDSGER W. DIJKSTRA
Keynote address at the ACM Symposium on Applied Computing (1999)

4 Linear Potential

Hofmann and Jost introduced linear automated amortized analysis in 2003 to analyze the heap-space consumption of first-order functional programs. As I am writing this thesis, their work [HJ03] has been cited more then 200 times[1] and has been developed further in several directions. Linear amortized analysis has been applied to analyze object-oriented programs [HJ06, HR09], to compute bounds for generic resources [JLH+09, Cam09], to analyze polymorphic and higher-order programs [JHLH10], and to analyze Java-like bytecode by means of separation logic [Atk10].

In this chapter I present a linear amortized analysis system for generic resources, following [JLH+09]. It is the basis of the polynomial analysis systems that I develop in the following two chapters and introduces many concepts that are used there. An informal introduction to linear amortized analysis can be found in Section 2.2.1.

The chapter is organized as follows. In Section 4.1, I define linear resource-annotated data types and the potential functions that the annotations represent. I then, in Section 4.2, introduce type judgments that constitute resource bounds together with type rules to derive the judgments for RAML programs. In Section 4.3, I prove the soundness of the type system. It states that derived type judgments constitute correct bounds. Section 4.4 explains how the type analysis can be automated through an inference of the type derivations. Finally, Section 4.5 demonstrates the analysis on several example programs.

4.1 Resource Annotations

The first step in the design of an automatic amortized analysis is to choose a set of potential functions. In this chapter, I use potential functions that are linear in the size of the data in the memory.

[1] according to Google Scholar

To represent the linear potential functions in the type system, types of inductive data structures are annotated with non-negative rational numbers[2] $q \in \mathbb{Q}_0^+$. The following EBNF grammar defines the *(linear) resource-annotated data types* of RAML.

$$A ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid L^q(A) \mid T^q(A) \mid (A, A)$$

Let $\mathscr{A}_{\text{lin}}$ be the set of linear resource-annotated data types. Let $A \in \mathscr{A}_{\text{lin}}$ be an annotated data type. As in Section 3.2, I write $[\![A]\!]$ for the set of semantic values of type A. For instance, $[\![L^q(\mathit{int})]\!]$ is the set of (finite) lists of integers. Similarly, we extend all other definitions—such as $H \vDash v \mapsto a : A$ and $H \vDash v : A$—for simple data types to resource-annotated data types by ignoring the resource annotations.

Let $A \in \mathscr{A}_{\text{lin}}$ by a resource-annotated data type and let $a \in [\![A]\!]$. The *potential* $\Phi(a{:}A)$ of a under type A is defined as follows. Recall from Section 3.2 that $\text{elems}(t)$ are the elements of the tree $t \in [\![T(A)]\!]$ in pre-order.

$$\begin{aligned}
\Phi(a{:}A) &= 0 && \text{if } A \in \{\mathit{unit}, \mathit{int}, \mathit{bool}\}\\
\Phi(a{:}(A_1, A_2)) &= \Phi(a_1{:}A_1) + \Phi(a_2{:}A_2) && \text{if } a = (a_1, a_2)\\
\Phi(\ell{:}L^q(B)) &= q \cdot n + \sum_{i=1,\ldots,n} \Phi(a_i{:}B) && \text{if } \ell = [a_1, \ldots, a_n]\\
\Phi(t{:}T^q(B)) &= q \cdot n + \sum_{i=1,\ldots,n} \Phi(a_i{:}B) && \text{if } \text{elems}(t) = [a_1, \ldots, a_n]
\end{aligned}$$

Let $A \in \mathscr{A}_{\text{lin}}$, let H be a heap, and let $v \in \mathit{Val}$ be a value such that $H \vDash v \mapsto a : A$. The potential $\Phi_H(v{:}A)$ of v under type A in H is then defined as $\Phi_H(v{:}A) = \Phi(a{:}A)$.

In the following I will sometimes explain an idea by talking about the potential $\Phi(x{:}A)$ of a variable x with respect to an annotated type A. In such a case I mean in fact the potential $\Phi_H(V(x){:}A)$ with respect to a stack V and a heap H that I do not want to describe precisely.

Lemma 4.1.1 states some facts about the potential of a value without referring to the corresponding semantic value. These facts can also be used to define the potential function Φ.

Lemma 4.1.1 Let $A \in \mathscr{A}_{\text{lin}}$, let H be a heap and let $v \in \mathit{Val}$ be a value with $H \vDash v : A$. Then the following is true.

1. $\Phi_H(v{:}A) = 0$ if $v = \text{NULL}$ or if $A \in \{\mathit{int}, \mathit{unit}, \mathit{bool}\}$
2. $\Phi_H((v_1, v_2){:}(A_1, A_2)) = \Phi_H(v_1{:}A_1) + \Phi_H(v_2{:}A_2)$
3. $\Phi_H(\ell{:}L^q(B)) = q + \Phi_H(v_1{:}B) + \Phi_H(\ell'{:}L^q(B))$ if $H(\ell){=}(v_1, \ell')$.
4. $\Phi_H(\ell{:}T^q(B)) = q + \Phi_H(v_1{:}B) + \Phi_H(\ell_1{:}T^q(B)) + \Phi_H(\ell_2{:}T^q(B))$ if $H(\ell){=}(v_1, \ell_1, \ell_2)$

[2] The use of rational rather than natural numbers in the potential annotations leads to more precise bounds. An example is given in Section 4.5.

PROOF 1. Since $H \vDash v : A$, we have $H \vDash v \mapsto [] : L(A')$, $H \vDash v \mapsto \mathit{leaf} : T(A')$, $H \vDash v \mapsto n : \mathit{int}$, $H \vDash v \mapsto () : \mathit{unit}$, or $H \vDash v \mapsto a : \mathit{bool}$ for $a \in \{\mathit{True}, \mathit{False}\}$. Then the claim follows from the definition of Φ.

2. It follows from definition that $H \vDash v \mapsto (a_1, a_2) : (A_1, A_2)$, $H \vDash v_1 \mapsto a_1 : A_1$, and $H \vDash v_2 \mapsto a_2 : A_2$. The claim is thus a direct consequence of the definition of Φ.

3. From rule V:CONS we conclude that $H \vDash v_1 \mapsto a_1 : B$, $H \vDash \ell' \mapsto [a_2, \ldots, a_n] : L(B)$ and $H \vDash \ell \mapsto [a_1, \ldots, a_n] : L(B)$. Then $\Phi_H(\ell{:}L^q(B)) = qn + \sum_{1 \le i \le n} \Phi(a_i{:}B) = (q + \Phi(a_1{:}B)) + (q(n-1)\sum_{2 \le i \le n} \Phi(a_i{:}B)) = q + \Phi_H(v_1{:}B) + \Phi_H(\ell'{:}L^q(B))$.

4. The proof is similar to the list case. In addition, one has to use the fact that $\mathrm{elems}(\mathit{tree}(a, t_1, t_2)) = [a, a_1, \ldots, a_m, b_1, \ldots, b_m]$ where $\mathrm{elems}(t_1) = [a_1, \ldots, a_m]$ and $\mathrm{elems}(t_f) = [b_1, \ldots, b_m]$. ■

For instance, we have $\Phi([b_1, \ldots, b_n] : L^q(\mathit{bool}) = q \cdot n$ for a list $[b_1, \ldots, b_n]$ of Booleans. Similarly, we have for a list of lists of Booleans that $\Phi([[b_{11}, \ldots, b_{1,m_1}], \ldots, [b_{n1}, \ldots, b_{nm_n}]] : L^q(L^p(\mathit{bool})) = q \cdot n + p \cdot (m_1 + \cdots + m_n)$. Note that potential functions incorporate the length of each individual inner data structure. This is an important property that enables the precise analysis of nested data structures.

The Subtyping Relation

Intuitively, it is true that a resource-annotated data type A is a subtype of a resource-annotated data type B if and only if A and B have the same set $[\![A]\!]$ of semantic values, and for every value $a \in [\![A]\!]$ the potential $\Phi(a{:}A)$ is greater or equal than the potential of $\phi(a{:}B)$. More formal, we define $<:$ to be the smallest relation such that the following is true.

$$
\begin{array}{rl}
C <: C & \text{if } C \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\} \\
(A_1, A_2) <: (B_1, B_2) & \text{if } A_1 <: B_1 \text{ and } A_2 <: B_2 \\
L^p(A) <: L^q(B) & \text{if } A <: B \text{ and } p \ge q \\
T^p(A) <: T^q(B) & \text{if } A <: B \text{ and } p \ge q
\end{array}
$$

Lemma 4.1.2 Let A, B be two resource-annotated data types with $A <: B$. Then $[\![A]\!] = [\![B]\!]$ and $\Phi(a{:}A) \ge \Phi(a{:}B)$ for all $a \in [\![A]\!]$.

PROOF By induction on the definition of subtyping relation. If $A = B \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\}$ then $[\![A]\!] = [\![B]\!]$ and $\Phi(a{:}A) = 0 = \Phi(a{:}B)$.

If $A = (A_1, A_2)$ then $B = (B_1, B_2)$, $A_1 <: B_1$ and $A_2 <: B_2$. By induction it follows that $[\![A_i]\!] = [\![B_i]\!]$ and $\Phi(a_i{:}A_i) \ge \Phi(a_i{:}B_i)$ for all $(a_1, a_2) \in (A_1, A_2)$. But then $[\![A]\!] = [\![B]\!]$ and $\Phi((a_1, a_2){:}A) = \Phi(a_1{:}A_1) + \Phi(a_2{:}A_2) \ge \Phi(a_1{:}B_1) + \Phi(a_2{:}B_2) = \Phi(a{:}B)$.

If $A = L^p(A')$ then $B = L^q(B')$ for a $q \in \mathbb{Q}_0^+$, $A <: B$, and $p \geq q$. By induction we have $[\![A']\!] = [\![B']\!]$ and thus $[\![A]\!] = [\![B]\!]$. Let $[a_1, \ldots, a_n] \in [\![L^p(A')]\!]$. Then

$$\begin{aligned}
\Phi([a_1, \ldots, a_n] : L^p(A')) &= pn + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}A') && \text{(Def.)} \\
&\geq qn + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}A') && (p \geq q) \\
&\geq qn + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}B') && \text{(Ind.)} \\
&= \Phi([a_1, \ldots, a_n] : L^q(B')) && \text{(Def.)}
\end{aligned}$$

The case $A = T^p(A')$ is very similar to the case $A = L^q(A')$. ∎

The Sharing Relation

The sharing relation $\curlyvee$ defines how the potential of a (zero-order) variable can be shared by multiple occurrences of that variable. We have $A \curlyvee (A_1, A_2)$ if and only if A, A_1 and A_2 are structural identical, that is, have the same set $[\![A]\!]$ of semantic values, and for every value $a \in [\![A]\!]$ the potential $\Phi(a{:}A)$ is identical to the sum $\Phi(a{:}A_1) + \Phi(a{:}A_2)$. The sharing relation $\curlyvee$ is the smallest relation such that following holds.

$$\begin{aligned}
C &\curlyvee (C, C) && \text{if } C \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\} \\
(A, B) &\curlyvee ((A_1, B_1), (A_2, B_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } B \curlyvee (B_1, B_2) \\
L^p(A) &\curlyvee (L^q(A_1), L^r(A_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } p = q + r \\
T^p(A) &\curlyvee (T^q(A_1), T^r(A_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } p = q + r
\end{aligned}$$

Lemma 4.1.3 Let A, A_1, and A_2 be resource-annotated data types with $A \curlyvee (A_1, A_2)$. Then $[\![A]\!] = [\![A_1]\!] = [\![A_2]\!]$ and $\Phi(a{:}A) = \Phi(a{:}A_1) + \Phi(a{:}A_2)$ for all $a \in [\![A]\!]$.

PROOF The proof is similar to the proof of Lemma 4.1.2 ∎

4.2 Type Rules

This section presents typing rules that assign resource-annotated data types to RAML expressions.

Like in the case of simple types, a *typing context* is a partial finite mapping $\Gamma : \mathrm{VID} \to \mathscr{A}_{\mathrm{lin}}$ from variable identifiers to resource-annotated data types. The potential of a typing context Γ with respect to a heap H and a stack V is

$$\Phi_{V,H}(\Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi_H(V(x){:}\Gamma(x)) \,.$$

Sometimes I just write $\Phi(\Gamma)$ in informal discussions leaving stack and heap implicit.

The *(linear) resource-annotated first-order types* are defined by the following grammar.

$$F ::= A \xrightarrow{q/q'} A$$

Here, q, q' are rational numbers and A ranges over the resource-annotated data types. The intended meaning is that q is the constant potential before a call to the function and q' is the constant potential after the call to the function. Let $\mathscr{F}_{\text{lin}}$ denote the set of resource-annotated first-order types.

A *resource-annotated signature* $\Sigma : \text{FID} \to (\mathscr{P}(\mathscr{F}_{\text{lin}})\setminus\emptyset)$ is a finite, partial mapping of function identifiers to *non-empty sets of* resource-annotated first-order types. As a result, every function can have different resource annotations depending on the context.

A *resource-annotated typing judgment* has the form

$$\Sigma;\Gamma \vdash^{q}_{q'} e{:}A$$

where e is a RAML expression, $q, q' \in \mathbb{Q}_0^+$ are non-negative rational numbers, Σ is a resource-annotated signature, Γ is a resource-annotated context and A is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $q + \Phi(\Gamma)$ resource units available then this is sufficient to evaluate e and there are more than $q' + \Phi(v{:}A)$ resource units left if e evaluates to a value v.

Similarly as for simple types, a RAML program with resource-annotated types consists of a resource-annotated signature Σ and a family $(e_f, y_f)_{f\in\text{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash^{q}_{q'} e_f{:}B$ for each $A \xrightarrow{q/q'} B \in \Sigma(f)$.

Figures 4.1 and 4.2 contain the type rules to derive resource-annotated type judgments for RAML expressions. All rationals that appear in the rules are non-negative. If an arithmetic expression like $p - q$ occurs in a rule then we have the implicit side condition that $p - q \geq 0$. Also recall, that I write $e[z/x]$ to denote the expression e with all free occurrences of the variable x replaced with the variable z.

Figure 4.1 contains only syntax-directed rules. This means that there is exactly one rule for every syntactic expression. Figure 4.2 contains one syntax-directed rule (namely L:MATT) and structural rules that can be applied to every syntactic form. In the type inference, the structural rules have to be incorporated into the syntax-directed rules. Details are given in Section 4.4.

The most interesting syntax-directed rules are the ones for lists and trees. Before I explain them, I describe the rules L:VAR and L:APP that are more suitable to explain the general idea.

(L:VAR) According to the operational semantics of RAML, the evaluation of a variable costs K^{var} resources. The rule (L:VAR) reflects this fact by requiring the constant potential before the evaluation of a variable to be $q + K^{\text{var}}$. The potential K^{var} is used up after the evaluation and there is the constant potential q left. If $K^{\text{var}} < 0$ then the resulting potential is greater then the initial potential. In this case, we have the implicit side condition $q + K^{\text{var}} \geq 0$ since all potential annotations must be non-negative.

(L:APP) The evaluation of a function application costs K_1^{app} resources before the evaluation of the body of the function, and K_2^{app} resources after the valuation of the body. Since $A \xrightarrow{q/q'} B \in \Sigma(f)$, we have $\Sigma; y_f{:}A \vdash^{q}_{q'} e_f{:}B$. So we need $q + \Phi(x{:}A)$ resources to evaluate the body e_f of the function. Thus we require the initial potential

$$\frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{unit}}}{q}} () : \mathit{unit}}\ (\text{L:ConstU}) \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{bool}}}{q}} b : \mathit{bool}}\ (\text{L:ConstB})$$

$$\frac{n \in \mathbb{Z}}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{int}}}{q}} n : \mathit{int}}\ (\text{L:ConstI}) \qquad \frac{\mathit{op} \in \{+, -, *, \mathit{mod}, \mathit{div}\}}{\Sigma; x_1{:}\mathit{int}, x_2{:}\mathit{int} \vdash^{\frac{q+K^{op}}{q}} x_1\ \mathit{op}\ x_2 : \mathit{int}}\ (\text{L:OpInt})$$

$$\frac{}{\Sigma; x{:}B \vdash^{\frac{q+K^{\text{var}}}{q}} x : B}\ (\text{L:Var}) \qquad \frac{\mathit{op} \in \{\mathit{or}, \mathit{and}\}}{\Sigma; x_1{:}\mathit{bool}, x_2{:}\mathit{bool} \vdash^{\frac{q+K^{op}}{q}} x_1\ \mathit{op}\ x_2 : \mathit{bool}}\ (\text{L:OpBool})$$

$$\frac{\Sigma;\Gamma \vdash^{\frac{q-K_1^{\text{conT}}}{q'+K_2^{\text{conT}}}} e_t : B \qquad \Sigma;\Gamma \vdash^{\frac{q-K_1^{\text{conF}}}{q'+K_2^{\text{conF}}}} e_f : B}{\Sigma;\Gamma, x{:}\mathit{bool} \vdash^{\frac{q}{q'}} \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f : B}\ (\text{L:Cond})$$

$$\frac{\Sigma;\Gamma_1 \vdash^{\frac{q-K_1^{\text{let}}}{p}} e_1 : A \qquad \Sigma;\Gamma_2, x{:}A \vdash^{\frac{p-K_2^{\text{let}}}{q'+K_3^{\text{let}}}} e_2 : B}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{\frac{q}{q'}} \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : B}\ (\text{L:Let})$$

$$\frac{A \xrightarrow{q/q'} B \in \Sigma(f)}{\Sigma; x{:}A \vdash^{\frac{q+K_1^{\text{app}}}{q'-K_2^{\text{app}}}} f(x) : B}\ (\text{L:App}) \qquad \frac{}{\Sigma; x_1{:}A_1, x_2{:}A_2 \vdash^{\frac{q+K^{\text{pair}}}{q}} (x_1, x_2) : (A_1, A_2)}\ (\text{L:Pair})$$

$$\frac{A = (A_1, A_2) \qquad \Sigma;\Gamma, x_1{:}A_1, x_2{:}A_2 \vdash^{\frac{q-K_1^{\text{matP}}}{q'+K_2^{\text{matP}}}} e : B}{\Sigma;\Gamma, x{:}A \vdash^{\frac{q}{q'}} \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e : B}\ (\text{L:MatP})$$

$$\frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{nil}}}{q}} \mathit{nil} : L^p(A)}\ (\text{L:Nil}) \qquad \frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{leaf}}}{q}} \mathit{leaf} : T^p(A)}\ (\text{L:Leaf})$$

$$\frac{}{\Sigma; x_h{:}A, x_t{:}L^p(A) \vdash^{\frac{q+p+K^{\text{cons}}}{q}} \mathit{cons}(x_h, x_t) : L^p(A)}\ (\text{L:Cons})$$

$$\frac{}{\Sigma; x_0{:}A, x_1{:}T^p(A), x_2{:}T^p(A) \vdash^{\frac{q+p+K^{\text{node}}}{q}} \mathit{node}(x_0, x_1, x_2) : T^p(A)}\ (\text{L:Node})$$

$$\frac{\Sigma;\Gamma \vdash^{\frac{q-K_1^{\text{matN}}}{q'+K_2^{\text{matN}}}} e_1 : B \qquad \Sigma;\Gamma, x_h{:}A, x_t{:}L^p(A) \vdash^{\frac{q+p-K_1^{\text{matC}}}{q'+K_2^{\text{matC}}}} e_2 : B}{\Sigma;\Gamma, x{:}L^p(A) \vdash^{\frac{q}{q'}} \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : B}\ (\text{L:MatL})$$

Figure 4.1: Linear resource-annotated type rules (1 of 2).

$$\frac{\Sigma;\Gamma \vdash^{q-K_1^{\text{matTL}}}_{q'+K_2^{\text{matTL}}} e_1 : B \qquad \Sigma;\Gamma, x_0{:}A, x_1{:}T^p(A), x_2{:}T^p(A) \vdash^{q+p-K_1^{\text{matTN}}}_{q'+K_2^{\text{matTN}}} e_2 : B}{\Sigma;\Gamma, x{:}T^p(A) \vdash^q_{q'} \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 : B}\ (\text{L:MATT})$$

$$\frac{\Sigma;\Gamma, x{:}A_1, y{:}A_2 \vdash^q_{q'} e : B \qquad A \curlyvee (A_1, A_2)}{\Sigma;\Gamma, z{:}A \vdash^q_{q'} e[z/x, z/y] : B}\ (\text{L:SHARE})$$

$$\frac{\Sigma;\Gamma, x{:}A \vdash^q_{q'} e : B \qquad A' <: A}{\Sigma;\Gamma, x{:}A' \vdash^q_{q'} e : B}\ (\text{L:SUPERTYPE}) \qquad \frac{\Sigma;\Gamma \vdash^q_{q'} e : B \qquad B <: B'}{\Sigma;\Gamma \vdash^q_{q'} e : B'}\ (\text{L:SUBTYPE})$$

$$\frac{\Sigma;\Gamma \vdash^p_{p'} e : B \qquad q \geq p \qquad q-p \geq q'-p'}{\Sigma;\Gamma \vdash^q_{q'} e : B}\ (\text{L:RELAX}) \qquad \frac{\Sigma;\Gamma \vdash^q_{q'} e : B}{\Sigma;\Gamma, x{:}A \vdash^q_{q'} e : B}\ (\text{L:AUGMENT})$$

Figure 4.2: Linear resource-annotated type rules (2 of 2).

$q + K_1^{\text{app}} + \Phi(x{:}A)$ in the rule L:APP. After the evaluation of the body of the function there are $q' + \Phi(f(x){:}B)$ resources left. Hence there are $q' - K_2^{\text{app}} + \Phi(f(x){:}B)$ resources left after the function application. Remember that we have the implicit side condition $q' - K_2^{\text{app}} \geq 0$.

(L:CONS) The construction of a new list element costs K^{cons} resource units[3]. Additionally, we have to pay for potential $\Phi(\mathit{cons}(x_h, x_t){:}L^p(A))$ of the resulting list. The potential $\Phi(x_t{:}L^p(A))$ of the tail and the potential $\Phi(x_h{:}A)$ is paid by the potential of the context. The missing potential p of the new list element, the resource cost K^{cons}, and the resulting constant potential q, are paid by the constant initial potential $q + p + K^{\text{cons}}$.

(L:MATL) The rule L:MATL defines how to use the potential of a list to pay for resource consumptions. First, it matches the corresponding rules E:MATN and E:MATC from the operational semantics in terms of constant resource cost (like L:APP). But it also incorporates the fact that either e_1 or e_2 is evaluated. The cons case is inverse to the rule L:CONS and allows one to use the potential associated with a list. For one thing, p resource units become available directly to pay for the evaluation of e_2. For another thing, the tail of the list is annotated with potential p.

The rules L:NIL and L:LEAF are similar to the rule L:VAR. It is safe to attach any potential annotation p to empty data structures since the resulting potential is always zero. The rules L:NODE and L:MATT are similar to L:CONS and L:MATL, respectively.

The structural type rules have three purposes. (1) Multiple occurrences of variables

[3] In fact, the resource cost of the construction of a list element often depend on the type A of the list elements. Since A is known at compile time this can be easily implemented in the type system. Just replace K^{cons} with $K^{\text{cons}}(A)$.

in expressions have to be introduced by the sharing rule L:SHARE. The sharing relation $\curlyvee$ ensures that the potential associated with the variable z, which occurs twice, is split between the variables x and y such that potential is neither gained nor lost.

(2) The syntax-directed rules are formulated with contexts that are minimal in the sense that they only mention variables that are needed in the rule. For instance, the rule L:VAR uses the context $x{:}B$ instead of $\Gamma, x{:}B$ for every Γ. If a variable occurs in a larger expression, then the rule L:AUGMENT can be used to delete variables from the context. If the deleted variable points to a list or a tree then its deletion can cause a loss of potential.

(3) There are many cases in which the syntax-directed rules implicitly assume that two resource annotations are equal or differ by a fixed constant. For instance, the rule L:CONS requires a context of the form $x_h{:}A_1, x_t{:}L^p(A_2)$ such that $A_1 = A_2$. Another example is the rule L:COND. It has the two premises $\Sigma;\Gamma \vdash^{q_1}_{q_1'} e_t : B$ and $\Sigma;\Gamma \vdash^{q_2}_{q_2'} e_f : B$ where $q_1 = q_2 + K_1^{\text{conF}} - K_1^{\text{conT}}$ and $q_1' = q_2' - K_2^{\text{conF}} + K_2^{\text{conT}}$. In practice, these requirements are often too rigid. That is why the rules L:RELAX, L:SUBTYPE, and L:SUPERTYPE can be used to equal two potential annotations in order to apply the syntax-directed rules. Their application can cause a loss of potential.

4.3 Soundness

In this section, I prove that type derivations establish correct bounds. An annotated type judgment for an expression e shows that if e evaluates to a value v in a well-formed environment then the initial potential of the context is an upper bound on the watermark of the resource usage. Moreover, the difference between the initial and the final potential is an upper bound on the consumed resources.

The introduction of the partial evaluation rules enables the formulation of a stronger soundness theorem than in earlier works on amortized analysis, as for instance, in [HH10b] or [JLH+09]. It states that the bounds derived from annotated type judgments also hold for non-terminating evaluations. Additionally, the new accounting of resource usage in the operational semantics allows for a more concise statement.

Theorem 4.3.1 (Soundness) Let $H \vDash V{:}\Gamma$ and let $\Sigma;\Gamma \vdash^{q}_{q'} e{:}B$.

1. If $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ then $p \leq \Phi_{V,H}(\Gamma) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q')$.

2. If $V, H \vdash e \rightsquigarrow \mid p$ then $p \leq \Phi_{V,H}(\Gamma) + q$.

It follows from Theorem 4.3.1 and Theorem 3.3.9 that run-time bounds also prove the termination of programs. Corollary 4.3.2 states this fact formally.

Corollary 4.3.2 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $H \vDash V{:}\Gamma$ and $\Sigma;\Gamma \vdash^{q}_{q'} e{:}A$ then there is an $n \in \mathbb{N}, n \leq \Phi_{V,H}(\Gamma) + q$ such that $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$.

Theorem 4.3.1 is proved by a nested induction on the derivation of the evaluation judgment—$V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ or $V, H \vdash e \rightsquigarrow \mid p$, respectively—and the type judgment $\Sigma; \Gamma \vdash^{q}_{q'} e{:}B$. The inner induction on the type judgment is needed because of the structural rules (compare the discussion in the proof of Theorem 3.3.4). There is one proof for all possible instantiations of the resource constants. It is technically involved but conceptually unsurprising.

The proof uses Lemma 4.3.3 to show the soundness of the rule L:LET. It states that the potential of a context is invariant during the evaluation. This is a consequence of allocated heap-cells being immutable with the language features that I describe in this thesis. Note, however, that it suffices to use the weaker statement $\Phi_{V,H}(\Gamma) \geq \Phi_{V,H'}(\Gamma)$ (rather than $\Phi_{V,H}(\Gamma) = \Phi_{V,H'}(\Gamma)$) in the soundness proof. It remains true in the presence of a destructive pattern match. The intuition is that the deallocation of heap cells can lead to a reduction of potential.

Lemma 4.3.3 Let $H \vDash V : \Gamma$, $\Sigma; \Gamma \vdash^{q}_{q'} e{:}A$, and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$. Then it is true that $\Phi_{V,H}(\Gamma) = \Phi_{V,H'}(\Gamma)$.

PROOF The lemma is a direct consequence of the definition of the potential Φ and the fact that $H'(\ell) = H(\ell)$ for all $\ell \in \mathrm{dom}(H)$ which is proved in Proposition 3.3.2. ∎

Proof of the Soundness Theorem

In the remainder of this section I prove Theorem 4.3.1.

PROOF (PART 1) I prove $p \leq \Phi_{V,H}(\Gamma) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q')$ by induction on the derivations of $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ and $\Sigma; \Gamma \vdash^{q}_{q'} e : B$, where the induction on the evaluation judgment takes priority.

(L:SHARE) Suppose that the derivation of $\Sigma; \Gamma \vdash^{q}_{q'} e{:}B$ ends with an application of the rule L:SHARE. Then $\Gamma = \Gamma', z{:}A$. It follows from the premise that

$$\Sigma; \Gamma', x{:}A_1, y{:}A_2 \vdash^{q}_{q'} e' : B \tag{4.1}$$

for data types A_i with $A \curlyvee (A_1, A_2)$ and an expression e' with $e'[z/x, z/y] = e$. Since $H \vDash V : \Gamma', z{:}A$ and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ it follows that $H \vDash V_{xy} : \Gamma', x{:}A, y{:}A$ and

$$V_{xy}, H \vdash e' \rightsquigarrow v, H' \mid (p, p') \tag{4.2}$$

where $V_{xy} = V \setminus z \cup \{x \mapsto V(x), y \mapsto V(z)\}$. Thus we can apply the induction hypothesis to (4.1) and (4.2) and derive

$$p \leq \Phi_{V_{xy},H}(\Gamma', x{:}A_1, y{:}A_2) + q \tag{4.3}$$

and

$$p - p' \leq \Phi_{V_{xy},H}(\Gamma', x{:}A_1, y{:}A_2) + q - (\Phi_{H'}(v{:}B) + q')\,. \tag{4.4}$$

By Lemma 4.1.3 we have that $\Phi_{V_{xy},H}(x{:}A_1) + \Phi_{V_{xy},H}(y{:}A_2) = \Phi_{V,H}(z{:}A)$ and hence

$$\Phi_{V_{xy},H}(\Gamma', x{:}A_1, y{:}A_2) = \Phi_{V,H}(\Gamma', z{:}A) \tag{4.5}$$

by Lemma 4.1.1. The claim follows from (4.3), (4.4), and (4.5).

(L:AUGMENT) If the derivation of $\Sigma;\Gamma \vdash^{q}_{q'} e : B$ ends with an application of the rule L:AUGMENT then we have $\Sigma;\Gamma' \vdash^{q}_{q'} e : B$ for a context Γ' with $\Gamma', x{:}A = \Gamma$. From the premise $H \vDash V : \Gamma', x{:}A$ it follows that $H \vDash V : \Gamma'$. Thus we can apply the induction hypothesis and derive $p \leq \Phi_{V,H}(\Gamma') + q \leq \Phi_{V,H}(\Gamma) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma') + q - (\Phi_{H'}(v{:}A) + q') \leq \Phi_{V,H}(\Gamma' + q - (\Phi_{H'}(v{:}A) + q')$. The respective second inequalities follow from $\Phi_{V,H}(\Gamma') \leq \Phi_{V,H}(\Gamma)$, which is a direct consequence of Lemma 4.1.1.

(L:SUPERTYPE) Assume the derivation of the typing judgment ends with an application of the type rule L:SUPERTYPE. Then we have $\Gamma = \Gamma', x{:}A$. Furthermore we have the premise

$$\Sigma;\Gamma', x{:}A \vdash^{q}_{q'} e : B \tag{4.6}$$

and $A' <: A$. Since A' and A have the same set of inhabitants (Lemma 4.1.2) it is true that $H \vDash V{:}\Gamma', x{:}A$. So we can apply the induction hypothesis to (4.6) and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ and derive $p \leq \Phi_{V,H}(\Gamma', x{:}A) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma', x{:}A) + q - (\Phi_{H'}(v{:}B) + q')$. From Lemma 4.1.2 it follows that $\Phi(a{:}A') \geq \Phi(a{:}A)$ for all $a \in [\![A]\!]$. Thus $p \leq \Phi_{V,H}(\Gamma', x{:}A') + q$ and $p - p' \leq \Phi_{V,H}(\Gamma', x{:}A') + q - (\Phi_{H'}(v{:}B) + q')$ follows directly from Lemma 4.1.1.

(L:SUBTYPE) Similar to the case (L:SUPERTYPE).

(L:RELAX) We apply the induction hypothesis to $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ and to the premise $\Sigma;\Gamma \vdash^{r}_{r'} e : B$ of L:RELAX. Then we have $p \leq \Phi_{V,H}(\Gamma) + r$ and $p - p' \leq \Phi_{V,H}(\Gamma) + r - (\Phi_{H'}(v{:}B) + r')$. From the premise of L:RELAX we have $q \geq r$ and $q - r \geq q' - r'$ and thus $q - q' \geq r - r'$. Therefrom the claim follows.

(L:VAR) Assume that e is a variable x that has been evaluated with the rule E:VAR. Assume first that $K^{\text{var}} \geq 0$. Then it follows by definition that $p = K^{\text{var}}$ and $p' = 0$. The type judgment $\Sigma;\Gamma \vdash^{q}_{q'} x{:}B$ has been derived by a single application of the rule L:VAR. Thus we have $0 \leq q' = q - K^{\text{var}}$ and therefore $p = K^{\text{var}} \leq q \leq \Phi_{V,H}(x{:}B) + q$. Furthermore it follows from the evaluation rule E:VAR that $v = V(x)$ and thus $p - p' = K^{\text{var}} = q - q' = \Phi_{V,H}(x{:}B) + q - (\Phi_{H'}(v{:}B) + q')$

Assume now that $K^{\text{var}} < 0$. Then it follows by definition that $p = 0$ and $p' = -K^{\text{var}}$. Thus $p = 0 \leq \Phi_{V,H}(x{:}B) + q$. We have again that $q - q' = K^{\text{var}} = p - p'$. Therefore the second part of the statement follows like in the case where $K^{\text{var}} \geq 0$.

(L:CONST*) Similar to the case (L:VAR).

(L:OPINT) Assume that the type derivation ends with an application of the rule L:OPINT. Then e has the form $x_1\ op\ x_2$ and the evaluation consists of an application of the rule E:BINOP. From the rule L:OPINT it follows that $0 \leq q' = q - K^{op}$. If $K^{op} \geq 0$

then $p = K^{op}$ and $p' = 0$. Thus $p = K^{op} \leq q = \Phi_{V,H}(x_1{:}int, x_2{:}int) + q$ and $p - p' = K^{op} = \Phi_{V,H}(x_1{:}int, x_2{:}int) + q - (\Phi_{H'}(v{:}int) + q')$.

If $K^{op} < 0$ then $p = 0$ and $p' = -K^{op}$. Thus $p \leq q = \Phi_{V,H}(x_1{:}int, x_2{:}int) + q$ and $p - p' = K^{op} = \Phi_{V,H}(x_1{:}int, x_2{:}int) + q - (\Phi_{H'}(v{:}int) + q')$.

(L:OpBool) The case in which the type derivation ends with an application of L:OpBool is similar to the case (L:OpInt).

(L:Nil) If the type derivation ends with an application of L:Nil then we have $e = nil$, $B = L^r(A)$ for some A, and $0 \leq q' = q - K^{\text{nil}}$. The corresponding evaluation rule E:Nil has been applied to derive the evaluation judgment and hence $v = \text{Null}$. If $K^{\text{nil}} \geq 0$ then $p = K^{\text{nil}}$ and $p' = 0$. Thus $p = K^{\text{nil}} \leq q = \Phi_{V,H}(\emptyset) + q$. Furthermore it follows from Lemma 4.1.1 that $\Phi_{H'}(\text{Null}{:}L^r(A)) = 0$. Thus $p - p' = K^{\text{nil}} = \Phi_{V,H}(\emptyset) + q - (\Phi_{H'}(\text{Null}{:}L^r(A)) + q')$. If $K^{\text{nil}} < 0$ then $p = 0$ and $p' = -K^{\text{nil}}$. Then $p \leq q$ and again $p - p' = K^{\text{nil}}$.

(L:Cons) If the type derivation ends with an application of the rule L:Cons then e has the form $cons(x_h, x_t)$ and it has been evaluated with the rule E:Cons. It follows by definition that $V, H \vdash cons(x_h, x_t) \rightsquigarrow \ell, H[\ell \mapsto v'] \mid K^{\text{cons}}$, $x_h, x_t \in \text{dom}(V)$, $v = (V(x_h), V(x_t))$, and $\ell \notin \text{dom}(H)$. Thus

$$p = K^{\text{cons}} \text{ and } p' = 0 \tag{4.7}$$

or (if $K^{\text{cons}} < 0$)

$$p = 0 \text{ and } p' = -K^{\text{cons}} \tag{4.8}$$

We have $B = L^s(A)$ and the type judgment $\Sigma; x_h{:}A, x_t{:}L^s(A) \vdash^{q}_{q'} cons(x_h, x_t) : L^s(A)$ has been derived by a single application of the rule L:Cons; thus

$$0 \leq q' = q - s - K^{\text{cons}}\,. \tag{4.9}$$

If $p = 0$ then $p \leq \Phi_{V,H}(\Gamma) + q$ holds because of our implicit side condition $q \geq 0$. Otherwise we have $p = K^{\text{cons}} \leq q \leq \Phi_{V,H}(\Gamma) + q$.

From Lemma 4.1.1 it follows that

$$s + \Phi_{V,H}(x_h{:}A, x_t{:}L^s(A)) = \Phi_{H[\ell \mapsto v']}(\ell : L^s(A)) \tag{4.10}$$

Therefore

$$\begin{aligned} \Phi_{V,H}(\Gamma) + q \quad &= \quad \Phi_{V,H}(x_h{:}A, x_t{:}L^s(A)) + q \\ &\overset{(4.9)}{=} \quad \Phi_{V,H}(x_h{:}A, x_t{:}L^s(A)) + q' + s + K^{\text{cons}} \\ &\overset{(4.10)}{=} \quad q' + K^{\text{cons}} + \Phi_{H[\ell \mapsto v']}(\ell : L^s(A)) \end{aligned}$$

and thus $\Phi_{V,H}(\Gamma) + q - (\Phi_{H[\ell \mapsto v']}(\ell{:}L^s(A)) + q') = K^{\text{cons}} = p - p'$.

(L:Leaf) This case is proved like the case (L:Nil).

(L:Pair and L:Node) Similar to the case (L:Cons).

(L:MATP) Assume that e is a pattern match *match x with* $(x_1, x_2) \to e'$ for a pair. Then the rule E:MATP has been used at the root of the derivation of the evaluation judgment. Therefore we have $V(x) = (v_1, v_2)$ and $V', H \vdash e' \rightsquigarrow v, H' \mid (r, r')$ for $V[x_1 \mapsto v_1, x_2 \mapsto v_2]$ and some r, r' with

$$(p, p') = K_1^{\text{matP}} \cdot (r, r') \cdot K_2^{\text{matP}} \tag{4.11}$$

Similarly, the type judgment for e has been derived by an application of the rule L:MATP and thus $\Gamma = \Gamma', x{:}A,\ \ A = (A_1, A_2),\ \ \Sigma; \Gamma', x_1{:}A_1, x_2{:}A_2 \vdash^{\frac{s}{s'}} e : B$, and

$$q = s + K_1^{\text{matP}} \text{ and } s' - K_2^{\text{matP}} = q' \geq 0 \tag{4.12}$$

for some A_1, A_2, s, s'. Since $H \vDash V' : \Gamma', x_1{:}A_1, x_2{:}A_2$ we can apply the induction hypothesis and with $\Phi_{V',H}(\Gamma', x_1{:}A_1, x_2{:}A_2) = \Phi_{V,H}(\Gamma)$ we derive

$$r \leq \Phi_{V,H}(\Gamma) + s \tag{4.13}$$

$$r - r' \leq \Phi_{V,H}(\Gamma) + s - (\Phi_{H'}(v{:}B) + s') \tag{4.14}$$

Let

$$(u, u') = K_1^{\text{matP}} \cdot (\Phi_{V,H}(\Gamma) + s, \Phi_{H'}(v{:}B) + s') \cdot K_2^{\text{matP}} \tag{4.15}$$

Per definition and since $s' \geq K_2^{\text{matP}}$, it follows that $u = \max(0, s + K_1^{\text{matP}} + \Phi_{V,H}(\Gamma))$ (recall that K_1^{matP} might be negative). From Proposition 3.3.1 applied to (4.13), (4.15) and (4.11) we derive $u \geq p$. If $s + K_1^{\text{matP}} + \Phi_{V,H}(\Gamma) \leq 0$ then $u = p = 0$ and $q + \Phi_{V,H}(\Gamma) \geq p$ trivially holds. If $s + K_1^{\text{matP}} + \Phi_{V,H}(\Gamma) > 0$ then it follows from (4.12) that

$$q + \Phi_{V,H}(\Gamma) = s + K_1^{\text{matP}} + \Phi_{V,H}(\Gamma) = u \geq p$$

Similarly, we apply Proposition 3.3.1 to (4.11) and use (4.14) and (4.12) to see that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{matP}} + K_2^{\text{matP}} \\ &\leq \Phi_{V,H}(\Gamma) + s - (\Phi_{H'}(v{:}A) + s') + K_1^{\text{matP}} + K_2^{\text{matP}} \\ &\leq \Phi_{V,H}(\Gamma) + (s + K_1^{\text{matP}}) - (\Phi_{H'}(v{:}A) + (s' - K_2^{\text{matP}}) \\ &= \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}A) + q') \end{aligned}$$

(L:APP) Assume that e is a function application of the form $f(x)$. The evaluation of e then ends with an application of the rule E:APP. Thus we have $V(x) = v'$ and $[y_f \mapsto v'], H \vdash e_f \rightsquigarrow v, H' \mid (r, r')$ for some r, r' with

$$(p, p') = K_1^{\text{app}} \cdot (r, r') \cdot K_2^{\text{app}} \tag{4.16}$$

The derivation of the type judgment for e ends with an application of L:FUN. Therefore it is true that $\Gamma = x{:}A,\ A \xrightarrow{s/s'} B \in \Sigma(f)$, and

$$q = s + K_1^{\text{app}} \text{ and } q' = s' - K_2^{\text{app}}\,. \tag{4.17}$$

In order to apply the induction hypothesis to the evaluation of the function body e_f we recall from the definition of a well-formed program that $A \xrightarrow{s/s'} B \in \Sigma(f)$ implies that $\Sigma; y_f{:}A \vdash^{s}_{s'} e_f{:}B$. Since $H \vDash V : x{:}A$ and $V(x) = v'$ it follows that $H \vDash [y_f \mapsto v'] : y_f{:}A$. We obtain by induction that

$$\begin{aligned} r &\leq \Phi_{[y_f \mapsto v'],H}(y_f{:}A) + s && (4.18)\\ r - r' &\leq \Phi_{[y_f \mapsto v'],H}(y_f{:}A) + s - (\Phi_{H'}(v{:}B) + s') && (4.19) \end{aligned}$$

Now everything is in place to proceed as in the case of E:MATP. Let

$$(u, u') = K_1^{\text{app}} \cdot (\Phi_{[y_f \mapsto v'],H}(y_f{:}A) + s, \Phi_{H'}(v{:}B) + s') \cdot K_2^{\text{app}} \,. \tag{4.20}$$

Then it follows that $p \leq u = \max(0, K_1^{\text{app}} + \Phi_{[y_f \mapsto v'],H}(y_f{:}A) + s)$. Furthermore we have $\Phi_{[y_f \mapsto v'],H}(y_f{:}A) = \Phi_{V,H}(x{:}A)$ and with (4.17) it follows that $p \leq q + \Phi_{V,H}(x{:}A)$.

For the second part for the statement observe that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{app}} + K_2^{\text{app}} \\ &\overset{(4.19)}{\leq} \Phi_{[y_f \mapsto v'],H}(y_f{:}A) + s - (\Phi_{H'}(v{:}B) + s') + K_1^{\text{app}} + K_2^{\text{app}} \\ &\leq \Phi_{V,H}(x{:}A) + s - (\Phi_{H'}(v{:}B) + s') + K_1^{\text{app}} + K_2^{\text{app}} \\ &= \Phi_{V,H}(x{:}A) + s + K_1^{\text{app}} - (\Phi_{H'}(v{:}B) + s' - K_2^{\text{app}}) \\ &\overset{(4.17)}{=} \Phi_{V,H}(x{:}A) + q - (\Phi_{H'}(v{:}B) + q') \end{aligned}$$

(L:COND) Similar to the case (L:MATP).

(L:MATL) Assume that the type derivation of e ends with an application of the rule L:MATL. Then e is a pattern match of the form $\mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2$ whose evaluation ends with an application of the rule E:MATCONS or E:MATNIL. The latter case is similar to the case (L:MATP). So assume the derivation of the evaluation judgment ends with an application of E:MATCONS.

Then $V(x) = \ell$, $H(\ell) = (v_h, v_t)$, and $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ for $V' = V[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r, r' with

$$(p, p') = K_1^{\text{matC}} \cdot (r, r') \cdot K_2^{\text{matC}} \tag{4.21}$$

Since the derivation of $\Sigma; \Gamma \vdash^{q}_{q'} e{:}A$ ends with an application of L:MATL, we have $\Gamma = \Gamma', x{:}L^t(A)$, $\Sigma; \Gamma', x_h{:}A, x_t{:}L^t(A) \vdash^{s}_{s'} e_2 : B$, and

$$q = s + K_1^{\text{matC}} - t \ \text{ and } \ q' = s' - K_2^{\text{matC}} \,. \tag{4.22}$$

It is true (by Lemma 4.1.1) that $\Phi_H(v{:}L^t(A)) = t + \Phi_H(v_h{:}A) + \Phi_H(v_t{:}L^t(A))$ and therefore

$$\Phi_{H,V}(\Gamma) = t + \Phi_{H,V'}(\Gamma', x_h{:}A, x_t{:}L^t(A)) \,. \tag{4.23}$$

Since $H \vDash V' : \Gamma', x_h{:}A, x_t{:}L^t(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ and obtain (with (4.23))

$$r \leq \Phi_{V,H}(\Gamma) - t + s \tag{4.24}$$
$$r - r' \leq \Phi_{V,H}(\Gamma) - t + s - (\Phi_{H'}(v{:}B) + s') \tag{4.25}$$

Note that $\Phi_{V,H}(\Gamma) - t \geq 0$ and let

$$(u, u') = K_1^{\text{matC}} \cdot (\Phi_{V,H}(\Gamma) - t + s, \Phi_{H'}(v{:}B) + s') \cdot K_2^{\text{matC}} \tag{4.26}$$

Per definition and from (4.22) it follows that $u = \max(0, \Phi_{V,H}(\Gamma) - t + s + K_1^{\text{matC}})$. From Proposition 3.3.1 applied to (4.24), (4.26) and (4.21) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma) - t + s + K_1^{\text{matC}} \leq 0$ then $u = p = 0$ and $q + \Phi_{V,H}(\Gamma) \geq p$ trivially holds. If $\Phi_{V,H}(\Gamma) - t + s + K_1^{\text{matC}} > 0$ then it follows from (4.22) that

$$q + \Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma) - t + s + K_1^{\text{matC}} = u \geq p\,.$$

Finally, we apply Proposition 3.3.1 to (4.21) to see that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{matC}} + K_2^{\text{matC}} \\ &\overset{(4.25)}{\leq} \Phi_{V,H}(\Gamma) - t + s - (\Phi_{H'}(v{:}B) + s') + K_1^{\text{matC}} + K_2^{\text{matC}} \\ &= \Phi_{V,H}(\Gamma) + (s + K_1^{\text{matC}} - t) - (\Phi_{H'}(v{:}B) + (s' - K_2^{\text{matC}})) \\ &\overset{(4.22)}{\leq} \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q') \end{aligned}$$

(L:MatT) Similar to the case (L:MatL).

(L:Let) If the type derivation ends with an application of L:Let then e is a let expression of the from *let* $x = e_1$ *in* e_2 that has eventually been evaluated with the rule E:Let. Then it follows that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (r, r')$ and $V', H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (t, t')$ for $V' = V[x \mapsto v_1]$ and r, r', t, t' with

$$(p, p') = K_1^{\text{let}} \cdot (r, r') \cdot K_2^{\text{let}} \cdot (t, t') \cdot K_3^{\text{let}} \tag{4.27}$$

The derivation of the type judgment for e ends with an application of L:Let. Hence $\Gamma = \Gamma_1, \Gamma_2$, $\Sigma; \Gamma_1 \vdash^{\frac{s_1}{s_1'}} e_1 : A$, $\Sigma; \Gamma_2, x{:}A \vdash^{\frac{s_2}{s_2'}} e_2 : B$, and

$$q = s_1 + K_1^{\text{let}} \tag{4.28}$$
$$s_1' = s_2 + K_2^{\text{let}} \tag{4.29}$$
$$q' = s_2' - K_3^{\text{let}} \tag{4.30}$$

It follows from the definition of Φ that

$$\Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) \tag{4.31}$$

Since $H \vDash V : \Gamma$ we have also $H \vDash V : \Gamma_1$ and can thus apply the induction hypothesis for the evaluation judgment for e_1 to derive

$$\begin{aligned} r &\leq \Phi_{V,H}(\Gamma_1) + s_1 && (4.32)\\ r - r' &\leq \Phi_{V,H}(\Gamma_1) + s_1 - (\Phi_{H_1}(v_1{:}A) + s_1') && (4.33)\end{aligned}$$

Form Theorem 3.3.4 it follows that $H_2 \vDash V' : \Gamma_2, x{:}A$ and thus again by induction

$$\begin{aligned} t &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2 && (4.34)\\ t - t' &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2 - (\Phi_{H_2}(v_2{:}B) + s_2') && (4.35)\end{aligned}$$

Now let

$$\begin{aligned}(u, u') = {} & K_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + s_1, \Phi_{H_1}(v_1{:}A) + s_1') \cdot K_2^{\text{let}} \cdot \\ & (\Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2, \Phi_{H_2}(v_2{:}B) + s_2') \cdot K_3^{\text{let}}\end{aligned}$$

Then it follows that

$$\begin{aligned}(u, u') &\overset{(4.29,4.30)}{=} K_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + s_1, \Phi_{H_1}(v_1{:}A) + s_1' - K_2^{\text{let}}) \cdot \\ &\qquad (\Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2, \Phi_{H_2}(v_2{:}B) + s_2' - K_3^{\text{let}}) \\ &= K_1^{\text{let}} \cdot (v + \Phi_{V,H}(\Gamma_1) + s_1, v')\end{aligned}$$

for $v, v' \in \mathbb{Q}_0^+$ with

$$\begin{aligned} v &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2 - (\Phi_{H_1}(v_1{:}A) + s_1' - K_2^{\text{let}}) \\ &= \Phi_{V',H_1}(\Gamma_2) + s_2 - (s_1' - K_2^{\text{let}}) \\ &\overset{(4.29)}{=} \Phi_{V',H_1}(\Gamma_2)\end{aligned}$$

and thus

$$\begin{aligned} u &\leq \max(0, \Phi_{V,H}(\Gamma_1) + \Phi_{V',H_1}(\Gamma_2) + s_1 + K_1^{\text{let}}) \\ &\overset{(\textit{Lem. } 4.3.3)}{\leq} \max(0, \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) + s_1 + K_1^{\text{let}}) \\ &\overset{(4.28)}{\leq} \Phi_{V,H}(\Gamma) + q\end{aligned}$$

Finally, it follows with Proposition 3.3.1 applied to (4.32), (4.34), and (4.27) that $u \geq p$.

For the second part of the statement we apply Proposition 3.3.1 to (4.27) and derive

the following.

$$\begin{array}{rcl}
p - p' & = & r - r' + t - t' + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
& \overset{(4.35,4.33)}{\leq} & \Phi_{V,H}(\Gamma_1) + s_1 - (\Phi_{H_1}(v_1{:}A) + s_1') + \Phi_{V',H_1}(\Gamma_2, x{:}A) \\
& & + s_2 - (\Phi_{H_2}(v_2{:}B) + s_2') + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
& = & (\Phi_{V,H}(\Gamma_1) + \Phi_{V',H_1}(\Gamma_2) + s_1) \\
& & + (s_2 + K_2^{\text{let}} - s_1') - (\Phi_{H_2}(v_2{:}B) + s_2') + K_1^{\text{let}} + K_3^{\text{let}} \\
& \overset{(4.29)}{=} & \Phi_{V,H}(\Gamma_1,) + \Phi_{V',H_1}(\Gamma_2) + s_1 - (\Phi_{H_2}(v_2{:}B) + s_2') + K_1^{\text{let}} + K_3^{\text{let}} \\
& \overset{(L.\ 4.3.3)}{\leq} & \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) + s_1 - (\Phi_{H_2}(v_2{:}B) + s_2') + K_1^{\text{let}} + K_3^{\text{let}} \\
& = & \Phi_{V,H}(\Gamma) + s_1 + K_1^{\text{let}} - (\Phi_{H_2}(v_2{:}B) + s_2' - K_3^{\text{let}}) \\
& \overset{(4.28,4.30)}{\leq} & \Phi_{V,H}(\Gamma) + q - (\Phi_{H_2}(v_2{:}B) + q')
\end{array}$$

■

PROOF (PART 2) The proof of part 2 is similar but simpler than the proof of part 1. However, it uses part 1 in the case of the rule P:LET2. Like in the proof of part 1, I prove $p \leq \Phi_{V,H}(\Gamma) + q$ by induction on the derivations of $V, H \vdash e \rightsquigarrow \mid p$ and $\Sigma; \Gamma \vdash^{q}_{q'} e : B$, where the induction on the partial evaluation judgment takes priority.

I only present some cases to convince you that the proof is similar to the poof of part 1.

(L:VAR) Assume that e is a variable x and that the type judgment $\Sigma; \Gamma \vdash^{q}_{q'} x{:}B$ has been derived by a single application of the rule L:VAR. Thus we have $0 \leq q' = q - K^{\text{var}}$.

Then e has been evaluated with a single application of the rule P:VAR and it follows by definition that $p = \max(K^{\text{var}}, 0)$. (Remember that $V, H \vdash x \rightsquigarrow \mid K^{\text{var}}$ is an abbreviation for $V, H \vdash x \rightsquigarrow \mid \max(K^{\text{var}}, 0)$ in P:VAR.)

Assume first that $K^{\text{var}} \geq 0$. Then we have $0 \leq q' = q - K^{\text{var}}$ and therefore $p = K^{\text{var}} \leq q \leq \Phi_{V,H}(x{:}B) + q$. Assume now that $K^{\text{var}} < 0$. Then it follows by definition that $p = 0$. Thus $p = 0 \leq \Phi_{V,H}(x{:}B) + q$.

(L:CONS) If the type derivation ends with an application of the rule L:CONS then e has the form $cons(x_h, x_t)$ and it has been evaluated with the rule P:CONS. It follows by definition that $V, H \vdash cons(x_h, x_t) \rightsquigarrow \mid \max(K^{\text{cons}}, 0)$. If $K^{\text{cons}} \leq 0$ and $p = 0$ then the claim follows immediately from the fact that the potential is non-negative. If $K^{\text{cons}} > 0$ and $p = K^{\text{cons}}$ then it follows with the rule L:CONS that $0 \leq q' = q - K^{\text{cons}}$ and thus $p = K^{\text{cons}} \leq q \leq \Phi_{V,H}(x{:}A) + q$.

(L:MATL) Assume that the type derivation of e ends with an application of the rule L:MATL. Then e is a pattern match of the form $match\ x\ with \mid nil \rightarrow e_1 \mid cons(x_h, x_t) \rightarrow e_2$ whose evaluation ends with an application of the rule P:MATCONS or P:MATNIL. Assume first that the derivation of the evaluation judgment ends with an application of P:MATCONS.

Then $V(x) = \ell$, $H(\ell) = (v_h, v_t)$, and $V', H \vdash e_2 \rightsquigarrow \mid r$ for $V' = V[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r with

$$p = \max(K_1^{\text{matC}} + r, 0) \tag{4.36}$$

Since, the derivation of $\Sigma;\Gamma \vdash^{q}_{q'} e{:}A$ ends with an application of L:MatL, we have $\Gamma = \Gamma', x{:}L^t(A)$, $\Sigma;\Gamma', x_h{:}A, x_t{:}L^t(A) \vdash^{s}_{s'} e_2 : B$, and

$$q = s + K_1^{\text{matC}} - t \tag{4.37}$$

It is true (by Lemma 4.1.1) that $\Phi_H(v{:}L^t(A)) = t + \Phi_H(v_h{:}A) + \Phi_H(v_t{:}L^t(A))$ and therefore

$$\Phi_{H,V}(\Gamma) = t + \Phi_{H,V'}(\Gamma', x_h{:}A, x_t{:}L^t(A)) \tag{4.38}$$

Since $H \vDash V' : \Gamma', x_h{:}A, x_t{:}L^t(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow \mid r$ and obtain (with (4.38))

$$r \quad \leq \quad s + \Phi_{V,H}(\Gamma) - t \tag{4.39}$$

If $p = 0$ then the claim follows immediately. Thus assume that $p = K_1^{\text{matC}} + r$. Then it follows from (4.39) and (4.37) that

$$p = K_1^{\text{matC}} + r \leq K_1^{\text{matC}} + s + \Phi_{V,H}(\Gamma) - t = q + \Phi_{V,H}(\Gamma)\,.$$

Assume now that the derivation of the evaluation judgment ends with an application of P:MatNil. Then $V, H \vdash e_1 \rightsquigarrow \mid r$ for a r with

$$p = \max(K_1^{\text{matN}} + r, 0)$$

Since, the derivation of $\Sigma;\Gamma \vdash^{q}_{q'} e{:}A$ ends with an application of L:MatL, we have $\Gamma = \Gamma', x{:}L^t(A)$, $\Sigma;\Gamma' \vdash^{s}_{s'} e_1 : B$, and

$$q = s + K_1^{\text{matN}} \tag{4.40}$$

Since $H \vDash V : \Gamma'$, we can apply the induction hypothesis to $V, H \vdash e_1 \rightsquigarrow \mid r$ and obtain (with (4.38))

$$r \quad \leq \quad s + \Phi_{V,H}(\Gamma) - t\,. \tag{4.41}$$

If $p = 0$ then the claim follows immediately. Assume that $p = K_1^{\text{matN}} + r$. Then it follows from (4.41) and (4.40) that

$$p = K_1^{\text{matN}} + r \leq K_1^{\text{matN}} + s + \Phi_{V,H}(\Gamma) - t \leq q + \Phi_{V,H}(\Gamma)\,.$$

(L:Let) If the type derivation ends with an application of L:Let then e is a let expression of the from *let* $x = e_1$ *in* e_2 that has eventually been evaluated with the rule P:Let1 or with the rule P:Let2.

Assume first that the evaluation judgment ends with an application of the rule P:LET2. Then it follows that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (r, r')$ and $V', H_1 \vdash e_2 \rightsquigarrow \mid t$ for $V' = V[x \mapsto v_1]$ and r, r', t with

$$(p, p') = K_1^{\text{let}} \cdot (r, r') \cdot K_2^{\text{let}} \cdot (t, 0) \tag{4.42}$$

The derivation of the type judgment for e ends with an application of L:LET. Hence $\Gamma = \Gamma_1, \Gamma_2, \quad \Sigma; \Gamma_1 \vdash^{\frac{s_1}{s_1'}} e_1 : A$, $\Sigma; \Gamma_2, x{:}A \vdash^{\frac{s_2}{s_2'}} e_2 : B$ and

$$q = s_1 + K_1^{\text{let}} \tag{4.43}$$
$$s_1' = s_2 + K_2^{\text{let}} \tag{4.44}$$

It follows from the definition of Φ that

$$\Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) \tag{4.45}$$

Since $H \vDash V : \Gamma$ we have also $H \vDash V : \Gamma_1$ and can thus apply part 1 of the soundness theorem to the evaluation judgment for e_1 to derive

$$r \leq \Phi_{V,H}(\Gamma_1) + s_1 \tag{4.46}$$
$$r - r' \leq \Phi_{V,H}(\Gamma_1) + s_1 - (\Phi_{H_1}(v_1{:}A) + s_1') \tag{4.47}$$

Form Theorem 3.3.4 it follows that $H_2 \vDash V' : \Gamma_2, x{:}A$ and we can apply the induction hypothesis for the partial evaluation judgment for e_2 to obtain

$$t \leq \Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2\,. \tag{4.48}$$

Now let

$$\begin{aligned}(u, u') = {} & K_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + s_1, \Phi_{H_1}(v_1{:}A) + s_1') \cdot K_2^{\text{let}} \cdot \\ & (\Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2, \Phi_{H_2}(v_2{:}B) + s_2')\end{aligned}$$

Then it follows that

$$\begin{aligned}(u, u') &\overset{(4.44)}{=} K_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + s_1, \Phi_{H_1}(v_1{:}A) + s_1' - K_2^{\text{let}}) \cdot \\ &\qquad (\Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2, \Phi_{H_2}(v_2{:}B) + s_2') \\ &= K_1^{\text{let}} \cdot (v + \Phi_{V,H}(\Gamma_1) + s_1, v')\end{aligned}$$

for $v, v' \in \mathbb{Q}_0^+$ with

$$\begin{aligned}v &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A) + s_2 - (\Phi_{H_1}(v_1{:}A) + s_1' - K_2^{\text{let}}) \\ &= \Phi_{V',H_1}(\Gamma_2) + s_2 - (s_1' - K_2^{\text{let}}) \\ &\overset{(4.44)}{=} \Phi_{V',H_1}(\Gamma_2)\end{aligned}$$

and thus

$$\begin{array}{rcl} u & \leq & \max(0, \Phi_{V,H}(\Gamma_1) + \Phi_{V',H_1}(\Gamma_2) + s_1 + K_1^{\text{let}}) \\ & \overset{(Lem.\ 4.3.3)}{\leq} & \max(0, \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) + s_1 + K_1^{\text{let}}) \\ & \overset{(4.43)}{\leq} & \Phi_{V,H}(\Gamma) + q \end{array}$$

Finally, it follows with Proposition 3.3.1 applied to (4.46), (4.48), and (4.42) that $u \geq p$.

Assume now that the evaluation judgment ends with an application of the rule P:LET1. Then it follows that $V, H \vdash e_1 \rightsquigarrow \mid r$ and

$$p = \max(K_1^{\text{let}} + r, 0)\,.$$

The derivation of the type judgment for e ends with an application of L:LET. Hence $\Gamma = \Gamma_1, \Gamma_2$, $\quad \Sigma; \Gamma_1 \vdash \frac{s}{s'}\ e_1 : A$, and

$$q \quad = \quad s + K_1^{\text{let}}\,. \tag{4.49}$$

It follows from the definition of Φ that $\Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2)$. Since $H \vDash V : \Gamma$ we have also $H \vDash V : \Gamma_1$ and can apply the induction hypothesis to the evaluation judgment for e_1 to derive

$$r \quad \leq \quad \Phi_{V,H}(\Gamma_1) + s\,. \tag{4.50}$$

If $p = 0$ then the claim follows immediately. Otherwise it is true that $p = K_1^{\text{let}} + r$. Then it follows from (4.50) and (4.49) that

$$p = K_1^{\text{let}} + r \leq K_1^{\text{let}} + s + \Phi_{V,H}(\Gamma_1) \leq q + \Phi_{V,H}(\Gamma)\,.$$

The other cases are similar to the case in which the derivation of the evaluation judgment ends with an application of P:LET1. ∎

4.4 Type Inference

In a nutshell, the type inference for linear amortized resource analysis is a usual type inference that collects linear constraints, which are solved by a linear programming solver (LP solver). You can think of the collection of the linear constraints as being performed in three steps.[4]

First, a standard type inference algorithm computes a type derivation of simple types (see Section 3.2) of RAML functions. Descriptions of such algorithms can be found in textbooks such as *Types and Programming Languages* [Pie02]. Since RAML programs are monomorphic, the user has to specify function types.

[4] In practice, we do it in one step only.

$$\frac{q \geq q' + K^{\text{var}}}{\Sigma; \Gamma, x{:}B \vdash^{q}_{q'} x : B} \text{ (A:VAR)}$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_n) \xrightarrow{p/p'} B \qquad q = p + c + K_1^{\text{app}} \qquad q' = p' + c - K_2^{\text{app}}}{\Sigma; \Gamma, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^{q}_{q'} f(x_1, \ldots, x_n) : B} \text{ (A:APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{s_n}_{s_n'} e_1{:}B_1 \qquad \Gamma, x_h{:}A, x_t{:}L^p(A) \vdash^{s_c}_{s_c'} e_2{:}B_2 \qquad B_i <: B \text{ for } i{=}1,2 \\ q{+}p \geq s_c{+}K_1^{\text{matC}} \qquad q \geq s_n{+}K_1^{\text{matN}} \qquad s_c' \geq q'{+}K_2^{\text{matC}} \qquad s_n' \geq q'{+}K_2^{\text{matN}}\end{array}}{\Sigma; \Gamma, x{:}L^p(A) \vdash^{q}_{q'} \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : B} \text{ (A:MATL)}$$

Figure 4.3: Representative resource-annotated algorithmic type rules.

Second, type derivation trees for simple types are converted into type derivation trees for resource-annotated types with yet unknown resource variables. To this end, every type in the derivation is replaced with a corresponding resource-annotated type with fresh resource variables. For instance, $L(\mathit{int})$ is replaced with $L^q(\mathit{int})$ such that the variable q occurs nowhere else in the derivation. Similarly, every occurrence of the symbol $\vdash$ is replaced with $\vdash^{q}_{q'}$ for fresh variables q and q'.

The third step is the collection of constraints on the resource annotations as required by algorithmic versions of the annotated type rules in Figures 4.1 and 4.2.

Algorithmic Type Rules

To obtain algorithmic type rules that can be used to produce the constraints during the type inference, the structural rules in Figure 4.2 have to be integrated in the syntax directed rules.

This integration is outlined in Section 4.2. In short, if the syntax-directed rules implicitly assume that two resource annotations are equal or differ by a fixed constant, an integration of the rules L:RELAX, L:SUBTYPE, or L:SUPERTYPE enable the analysis of a wider range of programs. Figure 4.3 shows algorithmic versions of some representative linear resource-annotated type rules. For convenience, I integrated construction and destruction of tuples into the rule A:APP.

A difference to standard type systems is the sharing rule S:SHARE that has to be applied if the same free variable is used more than once in an expression. The rule is not problematic for the type inference and there are several ways to deal with it in practice. The easiest way is maybe to transform input programs into programs that make sharing explicit before the type inference using a syntactic construct. Such a transformation is straightforward: Each time a free variable x occurs twice in an expression e, we replace the first occurrence of x with x_1 and the second occurrence of x with x_2 obtaining a

new expression e'. We then replace e with $share(x, x_1, x_2)$ *in* e'. In this way, the sharing rule becomes a normal syntax directed rule in the type inference. Another possibility is to integrate sharing directly into the type rule for let expressions as we did in an earlier work [HH10a]. Then you have to ensure a that variable only occurs once in each function or constructor call.

A fine point of the type inference arises from the treatment of function applications. The simplest way to treat them is to assume one fixed resource-annotated function type for each function. Each (possibly recursive) function application then uses this type. However, a context-sensitive analysis of functions extends the accuracy and range of the analysis. The reason is that one sometimes has to analyze function applications context-sensitively with respect to the call stack. Consider for example the expression *f(attach(x,l))* from Chapter 1 where you need to attach a potential to the result of *attach(x,l)* that depends on the resource consumption of the function *f*.

In our implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph.

Recursive function calls are always typed resource-monomorphically, that is to say, with the same type as the caller (see the following example). This approach enables an efficient inference. However, it makes the type inference incomplete with respect to the type rules. I give an example that cannot be typed with resource-monomorphic recursion in Section 4.5.

Example

In the following, I use the rules A:VAR, A:APP, and A:MATL from Figure 4.3 to demonstrate the process of inferring a resource annotated type. As an example I use the function *last:*$(int, L(int)) \to int$ that returns the last element of the input list or the integer input in the first component if the list is empty. It is implemented as follows.

```
last (acc,l) = match l with | nil  → acc
                            | x::xs → last(x,xs)
```

Figure 4.4 shows a classic type derivation that is annotated with resource variables. Below the derivation is the set of linear constraints as defined by the used algorithmic type rules. In the recursive call we require the function application to match its specification.

The constrains in Figure 4.4 can be simplified to $q \geq K_1^{\text{matN}} + K^{\text{var}} + K_2^{\text{matN}} + q'$ and $p \geq K_1^{\text{app}} + K_2^{\text{app}} + K_1^{\text{matC}} + K_2^{\text{matC}}$. Note that it is not always possible to simplify the constraints that arise in the type inference to such a simple form. In general p, q, and q' might appear in multiple constraints.

Objective Function

The final step of the type inference is to solve the linear constraints with an LP solver. The solver minimizes the variables in the constraints with respect to a given objective function. In the example in Figure 4.4, the objective function is $q_\Sigma + 1000p_\Sigma$. The

$$\dfrac{\dfrac{}{acc{:}int \vdash^{\frac{q_v}{q'_v}} acc : int}\,(\text{A:VAR}) \qquad \dfrac{\Sigma(last) = (int, L^{p_\Sigma}(int)) \xrightarrow{q_\Sigma / q'_\Sigma} int}{acc{:}int, x{:}int, xs{:}L^p(int) \vdash^{\frac{q_a}{q'_a}} last(x,xs) : int}\,(\text{A:APP})}{acc{:}int, l{:}L^p(int) \vdash^{\frac{q}{q'}} match\ l\ with \mid nil \to acc \mid cons(x,xs) \to last(x,xs) : int}\,(\text{A:MATL})$$

A:VAR:	$q_v \geq q'_v + K^{\text{var}}$			
A:APP:	$q_a = q_\Sigma + c + K_1^{\text{app}}$	$q'_a = q'_\Sigma + c - K_2^{\text{app}}$		
A:MATL:	$q + p \geq q_a + K_1^{\text{matC}}$	$q \geq q_v + K_1^{\text{matN}}$	$q'_a \geq q' + K_2^{\text{matC}}$	$q'_v \geq q' + K_2^{\text{matN}}$
Recursive:	$p_\Sigma = p$	$q_\Sigma = q$	$q'_\Sigma = q'$	

Minimize: $q_\Sigma + 1000\,p_\Sigma$

Figure 4.4: Inferring a linear resource-annotated type for the *last*: the annotated type derivation, the linear constraints derived from the algorithmic type rules, and the objective function.

multiplicative factors 1 and 1000 reflect that linear potential (p) is more expensive then constant potential (q). In general, we state in objective functions that inner potential, say, in list of list, is more expensive than outer potential.

The choice of the multiplicative factors is a heuristic. You can always construct RAML programs that will admit a linear constraint system in which the objective function is minimized by a solution that assign more potential to linear annotations than necessary. The problem is that classic linear programming does permit objectives that state that the minimization of one constraint is more important than the minimization of another.

In practice, the objective function is however not very important. The results are generally stable when changing the constant factors in the objective function. The reason is that cases where the LP solver has an option to trade linear for constant potential are relatively seldom. The example in Figure 4.4 is representative in this regard.

4.5 Examples

This section exemplifies the analysis with different RAML programs. At first, I demonstrate that the analysis works well on typical linear functions on lists and trees like *map*, *fold*, and *filter* operations, which are naturally implemented by using structural induction. Hereafter, I demonstrate the advantages of amortization by automatically analyzing a breath-fist search on trees that uses a stack. Then I give more theoretically motivated examples that demonstrate the need of *rational* potential and the possibility of analyzing non-terminating functions.

Structural Recursion

Many functions that often appear in functional programming are usually implemented using structural recursion and one recursive function call. Examples of such functions are *map*, *fold*, and *filter* operations on tree-like data structures. Linear amortized resource analysis works reliably and precisely for these functions. In most cases, the computed bounds for these functions exactly match the actual worst-case behavior. This is important for a successful deployment of the analysis in practice.

Consider for instance the function *plus:* $(T(int), int) \to T(int)$ that adds an integer to every node in an integer-labeled binary tree. It can be naturally implemented as follows.

```
plus (t,n) = match t with | leaf  → leaf
                          | node(x,t1,t2) → let t1' = plus(t1,n) in
                                            let t2' = plus(t2,n) in
                                            node (x+n,t1',t2')
```

Our prototype implementation computes the heap-space bound $3n$ and the evaluation-step bound $21n+3$, where n is the number of nodes in the input tree. Both bounds match the exact run-time behavior of the function. The respective types are the following.

$$\begin{aligned} plus &: (T^{3}(int), int) \xrightarrow{0/0} T^{0}(int) \\ plus &: (T^{21}(int), int) \xrightarrow{3/0} T^{0}(int) \end{aligned}$$

The inference first fixes the function type *plus:*$(T^{q}(int), int) \xrightarrow{p/p'} T^{q'}(int)$, where p, p', q and q' are variables that range over non-negative rational numbers. In the heap-space case it computes linear constraints that essentially state that $p \geq p'$ and $q \geq q' + 3$.

Another example is the function *zip:* $(L(int), L(int)) \to L(int)$ that can be implemented as follows.

```
zip (l1,l2) = match l1 with | nil  → nil
                            | x::xs → match l2 with
                                      | nil  → nil
                                      | y::ys → (x,y)::zip(xs,ys)
```

The expression *zip([1,2,3],[4,5,6])* evaluates for instance to *[(1,4),(2,5),(3,6)]*. The prototype implementation computes the heap-space bound $3m$ and the evaluation-step bound $10m + 2n + 3$, where n is the length of the first component and m is the length of the second component of the input. Both bounds are tight if $n = m$. A tight bound for inputs with $m \neq n$ would however be $\min(n, m)$ which cannot be expressed by the analysis system. However, our prototype computes exact bounds for the function in cases where concrete inputs are available. This is possible since both of the following heap-space typings are inferred depending on the context.

$$\begin{aligned} zip &: (L^{3}(int), L^{0}(int)) \xrightarrow{0/0} L^{0}(int) \\ zip &: (L^{0}(int), L^{3}(int)) \xrightarrow{0/0} L^{0}(int) \end{aligned}$$

A mixed typing like *zip:*$(L^1(int), L^2(int)) \xrightarrow{0/0} L^0(int)$ is also correct. The linear program that is inferred from the function definition states essentially that $q_1 + q_2 \geq 3 + q'$ and $p \geq p'$ if the function type is *zip:*$(L^{q_1}(int), L^{q_2}(int)) \xrightarrow{p/p'} L^q(int)$.

Breadth-First Search

The implementation of breadth-first search below is a nice example whose analysis relies heavily on amortization. The function *bfs'* uses a FIFO-queue that is implemented with two lists (in the functions *queue* and *fqueue*).

```
appendrev : (L(T(int)),L(T(int))) → L(T(int))

appendrev (toreverse,sofar) = match toreverse with
    | nil → sofar
    | a::as → appendrev(as,a::sofar);

reverse: L(T(int)) → L(T(int))

reverse xs = appendrev(xs,[]);

bfs : (T(int),int) → T(int)

bfs(t,x) = bfs'([t],[],x);

bfs' : (L(T(int)),L(T(int)),int) → T(int)

bfs'(queue,fqueue,x) = match queue with
  | nil → match fqueue with
            | nil → leaf
            | t::ts → bfs'(reverse(t::ts),[],x)
  | (t::ts) → match t with
              | leaf → bfs'(ts,fqueue,x)
              | node(y,t1,t2) → if x==y then node(y,t1,t2)
                                else bfs'(ts,t2::t1::fqueue,x);
```

For the evaluation-step metric, the prototype computes the following typing for the function *bfs*.

$$bfs \quad : \quad (T^{80}(int), int) \xrightarrow{21/0} T^0(int)$$

It states the fact that an evaluation of *bfs(t,x)* needs less then $80n+21$ evaluation steps if the tree t has n nodes. The previous typing is an instance of the more general typing *bfs:* $(T^q(int), int) \xrightarrow{p/p'} T^{q'}(int)$ if $p + 21 \geq p'$, $q \geq 80$ and $q \geq q'$. It is a particularly nice feature of this typing is that the potential of the subtree returned by *bfs* is not wasted but can be used in the rest of the program. An alternate type of the function is for instance *bfs:* $(T^{80}(int), int) \xrightarrow{51/30} T^{80}(int)$.

Rational Potential

To get a precise bound it is sometimes essential to assign rational potential to a data structure. A simple example is the function *group2:* $L(int) \to L((int, int))$ that is implemented below.

```
group2 l = match l with
           | nil  → nil
           | x::xs  → match xs with
                      | nil  → nil
                      | y::ys  → (x,y)::group2 ys
```

The expression *group2([1,2,3,4,5])* evaluates for example to *[(1,2),(3,4)]*. By inferring the following type, the prototype implementation computes the heap-space bound $1.5n$ for inputs of length n.

$$group2 \quad : \quad L^{1.5}(int) \xrightarrow{0/0} L^{0}((int, int))$$

This bound is the exact heap-space usage if n is even. If n is odd then the heap-space usage is $1.5(n-1)$. The constraints for the generic type *group2:* $L^{q}(int) \xrightarrow{p/p'} L^{q'}((int, int))$ are $p \geq p'$ and $2q \geq q' + 3$.

Non-Termination

Note that there is no syntactic restriction on the functions that can by analyzed by automatic amortized resource analysis. If a function does not consume resources then even non-termination is unproblematic.

Consider the function *omega* defined as follows.

```
omega (x) = omega (x)
```

For the heap-space metric and the generic type *omega:* $L^{q}(int) \xrightarrow{p/p'} L^{q'}(int)$, the constraint system states no restrictions on the values of the resource annotations. Consequently, our prototype infers that no heap-space is used by *omega*.

Since the prototype can infer the typing

$$omega \quad : \quad L^{0}(int) \xrightarrow{0/0'} L^{3}(int)$$

it can also infer that the expression *let l' = omega l in zip(l',l')* needs zero heap cells.

For an example of a non-terminating function that consumes heap-space consider the following function *fibs* that successively stores all Fibonacci numbers on the heap.

```
fibs l = matchD l with
         | nil  → ()
         | n::ls  → matchD ls with
                    | nil  → ()
                    | m::_  → fibs [m,n+m];
main = fibs [0,1]
```

The destructive pattern matching *matchD* deallocates the matched node of the list ℓ and frees 2 memory cells.[5] As a result, the function *fibs* stores the Fibonacci numbers in the heap space that is occupied by the input list ℓ without requiring additional space. The prototype implementation infers the following types for the program.

$$\begin{array}{rcl} \mathit{fibs} & : & L^0(\mathit{int}) \xrightarrow{0/0'} \mathit{unit} \\ \mathit{main} & : & \mathit{unit} \xrightarrow{4/0'} \mathit{unit} \end{array}$$

The type of *fibs* states that the function does not need any heap space and the type of *main* states that the main expressions requires four heap cells. These cells are used to create the initial list *[0,1]*.

[5] See Chapter 7 for details on destructive pattern matches.

The tension between conservativity and expressiveness is a fundamental fact of life in the design of type systems. The desire to allow more programs to be typed—by assigning more accurate types to their parts—is the main force driving research in the field.

BENJAMIN C. PIERCE
Types and Programming Languages (2002)

5 Univariate Polynomial Potential

Linear automatic amortized analysis works well in practice because of three reasons: it is compositional; it computes precise bounds; and the type inference uses linear constraint solving only. The main shortcoming of the analysis is its limitation to linear bounds.

In this chapter, I show how to overcome this shortcoming while preserving the appealing features of the analysis system. I describe an automatic amortized resource analysis that computes *univariate polynomial bounds*. It is based on two works that I presented at the 19th European Symposium on Programming (ESOP'10) [HH10b] and the eighth Asian Symposium on Programming Languages and Systems (APLAS'10) [HH10a]. You find an informal introduction of the main ideas in Section 2.2.2.

The structure of the chapter resembles the structure of Chapter 4. In Section 5.1, I introduce polynomial resource annotations and binomial coefficients as a basis for potential functions. A key notion is the additive shift that relates resource annotations of data structures with different sizes. Section 5.2 defines type judgments for annotated types that establish polynomial resource bounds and type rules that derive such type judgments for RAML programs. In Section 5.3, I prove the soundness of the resource bounds that are derived by resource-annotated type derivations.

Section 5.4 deals with the inference of type derivations. Despite of establishing polynomial bounds, the inference algorithm relies on linear constraint solving. A main challenge in the inference is the treatment of polymorphic recursion. Finally, in Section 5.5 I demonstrate the analysis with illustrative examples.

5.1 Resource Annotations

In this chapter I use potential functions that are non-negative linear combinations of binomial coefficients $\binom{n}{k}$, where k is a natural number and n is some size parameter

derived from the data structure. Notice that n depends sometimes on the height of a tree-like data structure; this is not the case in Chapters 4 and 6.

The following EBNF-grammar defines the *(univariate) resource-annotated data types* of RAML. A *resource annotation* $\vec{q} = (q_1, \ldots, q_k) \in (\mathbb{Q}_0^+)^k$ is a vector of non-negative rational numbers.

$$A ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid L^{\vec{q}}(A) \mid T^{\vec{q}}(A) \mid (A, A)$$

Let $\mathscr{A}_{\text{pol}}$ be the set of univariate resource-annotated data types.

Let $A \in \mathscr{A}_{\text{pol}}$ be an annotated data type. As in Chapter 4, I write $[\![A]\!]$ for the set of semantic values of type A. For instance, $[\![L^{\vec{q}}(\mathit{int})]\!]$ is the set of (finite) lists of integers. Also like in Chapter 4, all other definitions for simple data types from Section 3.2—such as $H \vDash v \mapsto a : A$ and $H \vDash v : A$—are extended to resource-annotated data types by ignoring the resource annotations.

For two resource annotations $\vec{p} = (p_1, \ldots, p_k)$ and $\vec{q} = (q_1, \ldots, q_\ell)$ I write $\vec{p} \le \vec{q}$ if $k \le \ell$ and $p_i \le q_i$ for all $1 \le i \le k$. If $\ell \ge k$ then we define $\vec{p} + \vec{q} = (p_1 + q_1, \ldots, p_k + q_k, q_{k+1}, \ldots, q_\ell)$.

One intuition for the resource annotations is as follows: The annotation $\vec{q}$ assigns the potential q_1 to every element of the data structures, the potential q_2 to every element of every proper suffix (sublist or subtree, respectively) of the data structure, q_3 to the elements of the suffixes of the suffixes, etc.

For linear potential annotations we can simply assign potential to sublists and subtrees by using the same annotations as for the corresponding parental data structures. This would however lead to a substantial loss of potential in the polynomial case. For that reason, I use an additive shift operation to assign potential to sublists and subtrees. It is an important concept of my work and discussed in more detailed in the remainder of this section.

Let $\vec{q} = (q_1, \ldots, q_k)$ be a resource annotation. The *additive shift* of $\vec{p}$ is

$$\lhd(\vec{p}) = (q_1 + q_2, q_2 + q_3, \ldots, q_{k-1} + q_k, q_k) \,.$$

In contrast with the definitions in Chapters 4 and 6, the potential Φ is defined recursively to unify the treatment of lists and trees (compare Lemma 4.1.1). I then develop closed formulas for the potential functions.

Let $A \in \mathscr{A}_{\text{pol}}$ by a resource-annotated data type and let $a \in [\![A]\!]$. The *potential* $\Phi(a{:}A)$ of a under type A is defined as follows.

$$\begin{aligned}
\Phi(a{:}A) &= 0 && \text{if } A \in \{\mathit{unit}, \mathit{int}, \mathit{bool}\} \\
\Phi((a_1, a_2){:}(A_1, A_2)) &= \Phi(a_1{:}A_1) + \Phi(a_2{:}A_2) \\
\Phi([]{:}L^{\vec{q}}(B)) &= 0 \\
\Phi((a :: \ell){:}L^{\vec{q}}(B)) &= q_1 + \Phi(a{:}B) + \Phi(\ell{:}L^{\lhd(\vec{q})}(B)) \\
\Phi(\mathit{leaf}{:}T^{\vec{q}}(B)) &= 0 \\
\Phi(\mathit{tree}(a, t_1, t_2){:}T^{\vec{q}}(B)) &= q_1 + \Phi(a{:}B) + \Phi(t_1{:}T^{\lhd(\vec{q})}(B)) + \Phi(t_2{:}T^{\lhd(\vec{q})}(B))
\end{aligned}$$

As usual, I assume in the definition that $\vec{q} = (q_1, \ldots, q_k)$.

Let $A \in \mathscr{A}_{\text{pol}}$, let H be a heap, and let $v \in \mathit{Val}$ be a value such that $H \vDash v \mapsto a : A$. The potential $\Phi_H(v{:}A)$ of v under type A in H is then defined as $\Phi_H(v{:}A) = \Phi(a{:}A)$.

In the following I will sometimes explain an idea referring to the potential $\Phi(x{:}A)$ of a variable x with respect to an annotated type A without mentioning a stack V and a heap H.

After having all the basic definitions in place, we investigate in the following what the potential Φ and the additive shift $\lhd$ mean for different data structures.

The Potential of Lists

To understand the potential functions for lists, we first consider some simple examples. Let for instance $\ell = [a_1 \ldots, a_n] : L(\mathit{int})$ be list of integers. Then the following is true for all $q_1, q_2, q_3 \in \mathbb{Q}_0^+$.

$$\begin{aligned}
\Phi(\ell{:}L^{(q_1)}(\mathit{int})) &= q_1 \cdot n \\
\Phi(\ell{:}L^{(0,q_2)}(\mathit{int})) &= \sum_{i=1}^{n-1} q_2 \cdot i = q_2 \frac{n \cdot (n-1)}{2} \\
\Phi(\ell{:}L^{(0,0,q_3)}(\mathit{int})) &= \sum_{i=1}^{n-1} q_3 \frac{i \cdot (i-1)}{2} = q_3 \frac{n \cdot (n-1) \cdot (n-2)}{6}
\end{aligned}$$

In fact, the potential of a list can always be written as a non-negative linear combination of binomial coefficients. This is proved by the following lemma. We define

$$\phi(n, \vec{p}) = \sum_{i=1}^{k} \binom{n}{i} p_i \,.$$

Lemma 5.1.1 Let $\ell = [a_1 \ldots, a_n] : L(A)$ be a list of type A and let $\vec{p} = (p_1, \ldots, p_k)$ be a resource annotation. Then

$$\Phi(\ell{:}L^{\vec{p}}(A)) = \phi(n, \vec{p}) + \sum_{i=1}^{n} \Phi(a_i{:}A) \,.$$

PROOF We prove the statement by induction on n. If $n = 0$ then $\ell = []$ and we have $\Phi(\ell{:}L^{\vec{p}}(A)) = 0 = \sum_{i=1}^{0} \Phi(a_i{:}A) + \phi(0, \vec{p})$.

Let $n > 0$. It then follows by induction that

$$\begin{aligned}
\Phi(\ell{:}L^{\vec{p}}(A)) &= p_1 + \Phi(a_1{:}A) + \Phi([a_2, \ldots, a_n]{:}\, L^{\lhd(\vec{p})}(A)) \\
&= p_1 + \sum_{i=1}^{n} \Phi(a_i{:}A) + \phi(n-1, \lhd(\vec{p}))
\end{aligned}$$

But since

$$\binom{n-1}{i} + \binom{n-1}{i+1} = \binom{n}{i+1} \tag{5.1}$$

it is true that

$$\begin{aligned}\phi(n-1,\lhd(\vec{p})) &= \sum_{i=1}^{k}\binom{n-1}{i}p_i + \sum_{i=1}^{k-1}\binom{n-1}{i}p_{i+1}\\ &= (n-1)p_1 + \sum_{i=1}^{k-1}\left(\binom{n-1}{i+1}+\binom{n-1}{i}\right)p_{i+1}\\ &= (n-1)p_1 + \sum_{i=1}^{k-1}\binom{n}{i+1}p_{i+1} \qquad \text{(by (5.1))}\\ &= \sum_{i=1}^{k}\binom{n}{i}p_i - p_1 = \phi(n,\vec{p}) - p_1 \qquad \blacksquare\end{aligned}$$

The use of binomial coefficients rather than powers of variables has many advantages as discussed in Section 2.2.2. In particular, the identity

$$\sum_{i=1,\ldots,k} q_i\binom{n+1}{i} = q_1 + \sum_{i=1,\ldots,k-1} q_{i+1}\binom{n}{i} + \sum_{i=1,\ldots,k} q_i\binom{n}{i}$$

gives rise to a local typing rule for *cons match* which naturally allows the typing of both recursive calls and other calls to subordinate functions in branches of a pattern match.

It is essential for the type system that ϕ is linear in the sense of the following lemma that follows directly from the definition of ϕ.

Lemma 5.1.2 Let $n \in \mathbb{N}$, $\alpha \in \mathbb{Q}$ and let $\vec{p},\vec{q}$ be resource annotations. Then $\phi(n,\vec{p}) + \phi(n,\vec{q}) = \phi(n,\vec{p}+\vec{q})$ and $\alpha\cdot\phi(n,\vec{p}) = \phi(n,\alpha\cdot\vec{p})$.

It is a general pattern in functional programs to compute a task on a list recursively for the tail of the list and to use the result of the recursive call to compute the result of the function. In such a recursive function it is natural to assign a uniform potential to each sublist (depending on its length) that occurs in a recursive call. In other words: one wants to use the potential of the input list to assign a uniform potential to every suffix of the list. With this view, the list potential $\alpha = \phi(n,(p_1,p_2,\cdots,p_k))$ can be read as follows: a recursive function on a list ℓ of length n that has the potential α can use the potential $\phi(i,(p_2,\cdots,p_k)$ for the suffixes of ℓ of length $1 \le i < n$ that occurs in the recursion. This intuition is proved by the following lemma.

Lemma 5.1.3 Let $\vec{p}$=$(p_1,\ldots,p_k)$ be a resource annotation, let $n \in \mathbb{N}$ and define $\phi(n,()) = 0$. Then $\phi(n,(p_1,\ldots,p_k)) = n\cdot p_1 + \sum_{i=1}^{n-1}\phi(i,(p_2,\ldots,p_k))$.

PROOF The proof uses the following well-known equation.

$$\sum_{i=1}^{n-1}\binom{i}{k} = \binom{n}{k+1} \text{ for each } k \in \mathbb{N} \tag{5.2}$$

Let now $k \geq 0$. Then

$$\begin{aligned}
\phi(n,(p_1,\ldots,p_{k+1})) &= \sum_{j=1}^{k+1} \binom{n}{j} p_j \\
&= n \cdot p_1 + \sum_{j=1}^{k} \binom{n}{j+1} p_{j+1} \\
&= n \cdot p_1 + \sum_{j=1}^{k} (\sum_{i=1}^{n-1} \binom{i}{j} p_{j+1})) && \text{(by (5.2))} \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} (\sum_{j=1}^{k} \binom{i}{j} p_{j+1})) \\
&= n \cdot p_1 + \sum_{i=1}^{n-1} \phi(i,(p_2,\ldots,p_{k+1})) && \text{(by definition)}
\end{aligned}$$

■

Note that the binomial coefficients are a basis of the vector space of the polynomials. Here, however, we are only interested in non-negative linear combinations of binomial coefficients. These admit a natural characterization in terms of growth: for $f : \mathbb{N} \to \mathbb{N}$ define $(\Delta f)(n) = f(n+1) - f(n)$. Call f *hereditarily non-negative* if $\Delta^i f \geq 0$ for all $i \geq 0$. One can show that a polynomial p is hereditarily non-negative if and only if it can be written as a non-negative linear combination of binomial coefficients. To wit, the coefficient of $\binom{n}{i}$ in the representation of p is $(\Delta^i p)(0)$. The hereditarily non-negative polynomials are scalar multiples of unary *resource polynomials* [GSS92] and thus are closed under sum, product, and composition. Note that they include all non-negative linear combinations of the polynomials $(x^i)_{i\in\mathbb{N}}$. In Chapter 6, I consider multivariate linear combinations of binomial coefficients and study their properties in more detail.

The Potential of Trees

As in the case of lists, closed forms of the potential functions for trees involve binomial coefficients. In contrast to the potential functions for trees in the linear and the multivariate system, the closed form depends on the shape of the tree.

The advantage of this univariate tree potential is that it allows for more precise bounds. The disadvantage is that it is not possible to transfer super-linear potential from a tree to a list or to a tree of a different shape. That is why I prefer the multivariate version of tree potential that I present in Chapter 6. It is of course possible to combine both forms of potential in a single analysis system.

Lemma 5.1.4 shows that there is a closed formula that exactly describes the potential of a tree. Note that the root of a tree has *height* 1 and that children of a node at height h have height $h+1$.

Lemma 5.1.4 Let $t : T(A)$ be a tree of height h with nodes $a_1,\ldots,a_n$ such that n_i is the number of nodes at level i. let $\vec{p} = (p_1,\ldots,p_k)$ be a resource annotation and define

$p_i = 0$ for $i > k$. Then

$$\Phi(t{:}T^{\vec{p}}(A)) = \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{h} n_i \left(\sum_{j=1}^{i} p_j \binom{i-1}{j-1} \right).$$

PROOF We prove the statement by induction on h. If $h = 0$ then $n = 0$ and the statement follows directly from the definition of Φ.

Let now $h > 0$. Then $t = \mathit{tree}(a_1, t_1, t_2)$ and

$$\begin{aligned}
\Phi(t{:}T^{\vec{p}}(A)) &= p_1 + \Phi(a_1{:}A) + \Phi(t_1{:}\, T^{\lhd(\vec{p})}(A)) + \Phi(t_2{:}\, T^{\lhd(\vec{p})}(A)) \\
&= p_1 + \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{h-1} n_{i+1} \left(\sum_{j=1}^{i} (p_j + p_{j+1}) \binom{i-1}{j-1} \right) \\
&= p_1 + \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=2}^{h} n_i \left(\sum_{j=1}^{i-1} (p_j + p_{j+1}) \binom{i-2}{j-1} \right) \\
&= p_1 + \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=2}^{h} n_i \left(p_1 + p_i + \sum_{j=2}^{i-1} p_j \left(\binom{i-2}{j-2} + \binom{i-2}{j-1} \right) \right) \\
&= \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{h} n_i \left(\sum_{j=1}^{i} p_j \binom{i-1}{j-1} \right)
\end{aligned}$$

■

Lemma 5.1.5 shows two simple bounds for tree potential functions that can be presented to a user after the analysis.

Lemma 5.1.5 Let $t : T(A)$ be a tree of height h with nodes $a_1, \ldots, a_n$ of type A and let $\vec{p} = (p_1, \ldots, p_k)$ be a resource annotation.

1. $\Phi(t{:}T^{\vec{p}}(A)) \le \phi(n, \vec{p}) + \sum_{i=1}^{n} \Phi(a_i{:}A)$
2. $\Phi(t{:}T^{\vec{p}}(A)) \le \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{k} p_i \cdot n \cdot (h-1)^{i-1}$

PROOF Part 1 follows by induction on n and from the fact that $\phi(n_1, \vec{p}) + \phi(n_2, \vec{p}) \le \phi(n_1 + n_2, \vec{p})$.

To prove part 2 let n_i be the number of nodes on level i. It follows from Lemma 5.1.4 that

$$\begin{aligned}
\Phi(t{:}T^{\vec{p}}(A)) &= \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{h} n_i \left(\sum_{j=1}^{i} p_j \binom{i-1}{j-1} \right) \\
&\le \sum_{i=1}^{n} \Phi(a_i{:}A) + \sum_{i=1}^{h} n_i \left(\sum_{j=1}^{h} p_j \binom{h-1}{j-1} \right) \\
&\le \sum_{i=1}^{n} \Phi(a_i{:}A) + n \left(\sum_{j=1}^{h} p_j \binom{h-1}{j-1} \right) \\
&\le \sum_{i=1}^{n} \Phi(a_i{:}A) + n \left(\sum_{j=1}^{k} p_j (h-1)^{j-1} \right)
\end{aligned}$$

■

The Subtyping Relation

Intuitively, a resource-annotated data type A is a subtype of a resource-annotated data type B if A and B have the same set $[\![A]\!]$ of semantic values, and for every value $a \in [\![A]\!]$ the potential $\Phi(a{:}A)$ is greater or equal than the potential of $\phi(a{:}B)$. More formal, we define $<:$ to be the smallest relation such that the following is true.

$$\begin{aligned} C &<: C && \text{if } C \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\} \\ (A_1, A_2) &<: (B_1, B_2) && \text{if } A_1 <: B_1 \text{ and } A_2 <: B_2 \\ L^{\vec{p}}(A) &<: L^{\vec{q}}(B) && \text{if } A <: B \text{ and } \vec{p} \geq \vec{q} \\ T^{\vec{p}}(A) &<: T^{\vec{q}}(B) && \text{if } A <: B \text{ and } \vec{p} \geq \vec{q} \end{aligned}$$

Lemma 5.1.6 Let A, B be two resource-annotated data types with $A <: B$. Then $[\![A]\!] = [\![B]\!]$ and $\Phi(a{:}A) \geq \Phi(a{:}B)$ for all $a \in [\![A]\!]$.

PROOF By induction on the definition of subtyping relation. If $A = B \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\}$ then $[\![A]\!] = [\![B]\!]$ and $\Phi(a{:}A) = 0 = \Phi(a{:}B)$.

If $A = (A_1, A_2)$ then $B = (B_1, B_2)$, $A_1 <: B_1$ and $A_2 <: B_2$. By induction it follows that $[\![A_i]\!] = [\![B_i]\!]$ and $\Phi(a_i{:}A_i) \geq \Phi(a_i{:}B_i)$ for all $(a_1, a_2) \in (A_1, A_2)$. But then $[\![A]\!] = [\![B]\!]$ and $\Phi((a_1, a_2){:}A) = \Phi(a_1{:}A_1) + \Phi(a_2{:}A_2) \geq \Phi(a_1{:}B_1) + \Phi(a_2{:}B_2) = \Phi(a{:}B)$.

If $A = L^{\vec{p}}(A')$ then $B = L^{\vec{q}}(B')$ for a $\vec{q}$, $A <: B$, and $\vec{p} \geq \vec{q}$. By induction we have $[\![A']\!] = [\![B']\!]$ and thus $[\![A]\!] = [\![B]\!]$. Let $[a_1, \ldots, a_n] \in [\![L^{\vec{p}}(A')]\!]$. Then

$$\begin{aligned} \Phi([a_1, \ldots, a_n] : L^{\vec{p}}(A')) &= \phi(n, \vec{p}) + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}A') && \text{(Lemma 5.1.3)} \\ &\geq \phi(n, \vec{q}) + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}A') && (p \geq q) \\ &\geq \phi(n, \vec{q}) + \textstyle\sum_{1 \leq i \leq n} \Phi(a_i{:}B') && \text{(Ind.)} \\ &= \Phi([a_1, \ldots, a_n] : L^{\vec{q}}(B')) && \text{(Lemma 5.1.3)} \end{aligned}$$

The case in which $A = T^{\vec{p}}(A')$ can be proved similarly to the case $A = L^{\vec{q}}(A')$ using Lemma 5.1.4. ∎

The Sharing Relation

The sharing relation $\curlyvee$ defines how the potential of a variable can be shared by multiple occurrences of that variable. We have $A \curlyvee (A_1, A_2)$ if and only if A, A_1 and A_2 are structural identical, that is, have the same set $[\![A]\!]$ of semantic values, and for every value $a \in [\![A]\!]$ the potential $\Phi(a{:}A)$ is identical to the sum $\Phi(a{:}A_1) + \Phi(a{:}A_2)$. The sharing relation $\curlyvee$ is the smallest relation such that following holds.

$$\begin{aligned} C &\curlyvee (C, C) && \text{if } C \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\} \\ (A, B) &\curlyvee ((A_1, B_1), (A_2, B_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } B \curlyvee (B_1, B_2) \\ L^{\vec{p}}(A) &\curlyvee (L^{\vec{q}}(A_1), L^{\vec{r}}(A_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } \vec{p} = \vec{q} + \vec{r} \\ T^{\vec{p}}(A) &\curlyvee (T^{\vec{q}}(A_1), T^{\vec{r}}(A_2)) && \text{if } A \curlyvee (A_1, A_2) \text{ and } \vec{p} = \vec{q} + \vec{r} \end{aligned}$$

Lemma 5.1.7 Let A, A_1, A_2 be three resource-annotated data types with $A \curlyvee (A_1, A_2)$. Then $[\![A]\!] = [\![A_1]\!] = [\![A_2]\!]$ and $\Phi(a{:}A) = \Phi(a{:}A_1) + \Phi(a{:}A_2)$ for all $a \in [\![A]\!]$.

PROOF The proof is similar to the proof of Lemma 5.1.6 ∎

5.2 Type Rules

In this section I define typing rules that assign univariate resource-annotated data types to RAML expressions. Some of the rules are identical to their linear counterparts from Chapter 4. The most import differences are the rules for construction and destruction of data structures.

As in the case of linear types, a *typing context* is a partial finite mapping $\Gamma : \mathrm{VID} \to \mathscr{A}_{\mathrm{pol}}$ from variable identifiers to (univariate) resource-annotated data types. In this chapter, however, potential annotations are vectors of non-negative rational numbers rather than single numbers.

The potential of a typing context Γ with respect to a heap H and a stack V is

$$\Phi_{V,H}(\Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi_H(V(x){:}\Gamma(x)) \,.$$

Sometimes I just write $\Phi(\Gamma)$ in informal discussions leaving stack and heap implicit.

Univariate resource-annotated first-order types are defined by the following grammar.

$$F ::= A \xrightarrow{q\,/\,q'} A$$

Here, q, q' are rational numbers and A ranges over the resource-annotated data types. The intended meaning is that q is the constant potential before a call to the function and q' is the constant potential after the call to the function. Let $\mathscr{F}_{\mathrm{pol}}$ denote the set of resource-annotated first-order types.

A *resource-annotated signature* $\Sigma : \mathrm{FID} \to (\mathscr{P}(\mathscr{F}_{\mathrm{pol}}) \setminus \emptyset)$ is a finite, partial mapping of function identifiers to *non-empty sets of* resource-annotated first-order types. As a result, every function can have different resource annotations depending on the context.

A *resource-annotated typing judgment* has the form

$$\Sigma; \Gamma \vdash^{q}_{q'} e{:}A$$

where e is a RAML expression, $q, q' \in \mathbb{Q}^+_0$ are non-negative rational numbers, Σ is a resource-annotated signature, Γ is a resource-annotated context and A is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $q + \Phi(\Gamma)$ resource units available then this is sufficient to evaluate e and there are more than $q' + \Phi(v{:}A)$ resource units left if e evaluates to a value v.

As for linearly annotated types, a RAML program with resource-annotated types consists of a resource-annotated signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash^{q}_{q'} e_f{:}B$ for each $A \xrightarrow{q\,/\,q'} B \in \Sigma(f)$.

$$\frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{unit}}}{q}} () : \mathit{unit}}\ (\text{U:ConstU}) \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{bool}}}{q}} b : \mathit{bool}}\ (\text{U:ConstB})$$

$$\frac{n \in \mathbb{Z}}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{int}}}{q}} n : \mathit{int}}\ (\text{U:ConstI}) \qquad \frac{\mathit{op} \in \{+, -, *, \mathit{mod}, \mathit{div}\}}{\Sigma; x_1{:}\mathit{int}, x_2{:}\mathit{int} \vdash^{\frac{q+K^{op}}{q}} x_1\ \mathit{op}\ x_2 : \mathit{int}}\ (\text{U:OpInt})$$

$$\frac{}{\Sigma; x{:}B \vdash^{\frac{q+K^{\text{var}}}{q}} x : B}\ (\text{U:Var}) \qquad \frac{\mathit{op} \in \{\mathit{or}, \mathit{and}\}}{\Sigma; x_1{:}\mathit{bool}, x_2{:}\mathit{bool} \vdash^{\frac{q+K^{op}}{q}} x_1\ \mathit{op}\ x_2 : \mathit{bool}}\ (\text{U:OpBool})$$

$$\frac{\Sigma;\Gamma \vdash^{\frac{q-K_1^{\text{conT}}}{q'+K_2^{\text{conT}}}} e_t : B \qquad \Sigma;\Gamma \vdash^{\frac{q-K_1^{\text{conF}}}{q'+K_2^{\text{conF}}}} e_f : B}{\Sigma;\Gamma, x{:}\mathit{bool} \vdash^{\frac{q}{q'}} \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f : B}\ (\text{U:Cond})$$

$$\frac{\Sigma;\Gamma_1 \vdash^{\frac{q-K_1^{\text{let}}}{p}} e_1 : A \qquad \Sigma;\Gamma_2, x{:}A \vdash^{\frac{p-K_2^{\text{let}}}{q'+K_3^{\text{let}}}} e_2 : B}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{\frac{q}{q'}} \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : B}\ (\text{U:Let})$$

$$\frac{A \xrightarrow{q/q'} B \in \Sigma(f)}{\Sigma; x{:}A \vdash^{\frac{q+K_1^{\text{app}}}{q'-K_2^{\text{app}}}} f(x) : B}\ (\text{U:App}) \qquad \frac{}{\Sigma; x_1{:}A_1, x_2{:}A_2 \vdash^{\frac{q+K^{\text{pair}}}{q}} (x_1, x_2) : (A_1, A_2)}\ (\text{U:Pair})$$

$$\frac{A = (A_1, A_2) \qquad \Sigma;\Gamma, x_1{:}A_1, x_2{:}A_2 \vdash^{\frac{q-K_1^{\text{matP}}}{q'+K_2^{\text{matP}}}} e : B}{\Sigma;\Gamma, x{:}A \vdash^{\frac{q}{q'}} \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e : B}\ (\text{U:MatP})$$

$$\frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{nil}}}{q}} \mathit{nil} : L^{\vec{p}}(A)}\ (\text{U:Nil}) \qquad \frac{}{\Sigma;\emptyset \vdash^{\frac{q+K^{\text{leaf}}}{q}} \mathit{leaf} : T^{\vec{p}}(A)}\ (\text{U:Leaf})$$

$$\frac{\vec{p} = (p_1, \ldots, p_k)}{\Sigma; x_h{:}A, x_t{:}L^{\lhd(\vec{p})}(A) \vdash^{\frac{q+p_1+K^{\text{cons}}}{q}} \mathit{cons}(x_h, x_t) : L^{\vec{p}}(A)}\ (\text{U:Cons})$$

$$\frac{\vec{p} = (p_1, \ldots, p_k)}{\Sigma; x_0{:}A, x_1{:}T^{\lhd(\vec{p})}(A), x_2{:}T^{\lhd(\vec{p})}(A) \vdash^{\frac{q+p_1+K^{\text{node}}}{q}} \mathit{node}(x_0, x_1, x_2) : T^{\vec{p}}(A)}\ (\text{U:Node})$$

Figure 5.1: Univariate resource-annotated type rules (part 1 of 2).

$$\frac{\begin{array}{c}\vec{p} = (p_1, \ldots, p_k) \\ \Sigma; \Gamma \vdash^{\frac{q - K_1^{\text{matN}}}{q' + K_2^{\text{matN}}}} e_1 : B \qquad \Sigma; \Gamma, x_h{:}A, x_t{:}L^{\lhd(\vec{p})}(A) \vdash^{\frac{q + p_1 - K_1^{\text{matC}}}{q' + K_2^{\text{matC}}}} e_2 : B\end{array}}{\Sigma; \Gamma, x{:}L^{\vec{p}}(A) \vdash^{\frac{q}{q'}} \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : B} \text{(U:MatL)}$$

$$\frac{\begin{array}{c}\vec{p} = (p_1, \ldots, p_k) \qquad \Sigma; \Gamma \vdash^{\frac{q - K_1^{\text{matTL}}}{q' + K_2^{\text{matTL}}}} e_1 : B \\ \Sigma; \Gamma, x_0{:}A, x_1{:}T^{\lhd(\vec{p})}(A), x_2{:}T^{\lhd(\vec{p})}(A) \vdash^{\frac{q + p_1 - K_1^{\text{matTN}}}{q' + K_2^{\text{matTN}}}} e_2 : B\end{array}}{\Sigma; \Gamma, x{:}T^{\vec{p}}(A) \vdash^{\frac{q}{q'}} \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 : B} \text{(U:MatT)}$$

$$\frac{\Sigma; \Gamma, x{:}A_1, y{:}A_2 \vdash^{\frac{q}{q'}} e : B \qquad A \curlyvee (A_1, A_2)}{\Sigma; \Gamma, z{:}A \vdash^{\frac{q}{q'}} e[z/x, z/y] : B} \text{(U:Share)}$$

$$\frac{\Sigma; \Gamma, x{:}A \vdash^{\frac{q}{q'}} e : B \qquad A' <: A}{\Sigma; \Gamma, x{:}A' \vdash^{\frac{q}{q'}} e : B} \text{(U:SuperType)} \qquad \frac{\Sigma; \Gamma \vdash^{\frac{q}{q'}} e : B \qquad B <: B'}{\Sigma; \Gamma \vdash^{\frac{q}{q'}} e : B'} \text{(U:SubType)}$$

$$\frac{\Sigma; \Gamma \vdash^{\frac{p}{p'}} e : B \qquad q \geq p \qquad q - p \geq q' - p'}{\Sigma; \Gamma \vdash^{\frac{q}{q'}} e : B} \text{(U:Relax)} \qquad \frac{\Sigma; \Gamma \vdash^{\frac{q}{q'}} e : B}{\Sigma; \Gamma, x{:}A \vdash^{\frac{q}{q'}} e : B} \text{(U:Augment)}$$

Figure 5.2: Univariate resource-annotated type rules (part 2 of 2).

Figures 5.1 and 5.2 contain the type rules to derive resource-annotated type judgments for RAML expressions. All rationals that appear in the rules are non-negative. If an arithmetic expression like $p - q$ occurs in a rule then we have the implicit side condition that $p - q \geq 0$. Recall, that I write $e[z/x]$ to denote the expression e with all free occurrences of the variable x replaced with the variable z.

There are syntax-directed and structural type rules. The purpose of the structural rules is described in Section 4.2. In the type inference, the structural rules have to be incorporated into the syntax-directed rules. I discuss this in more detail in Section 5.4.

Most of the rules are identical to the type rules for linear resource-annotated types from Chapter 4. You find explanations in Section 4.2. The rules that differ are U:NIL, U:CONS, U:MATL, U:LEAF, U:NODE, and U:MATT. They can be read as follows.

(U:NIL) According to the operational semantics of RAML, the evaluation of a variable costs K^{nil} resources. The rule U:NIL reflects this fact by requiring the constant potential before the evaluation of a variable to be $q + K^{\text{nil}}$. The potential K^{nil} is used up after the evaluation and there is the constant potential q left. If $K^{\text{nil}} < 0$ then the resulting potential is greater then the initial potential. In this case, we have the implicit side condition $q - K^{\text{nil}} \geq 0$ since all potential annotations must be non-negative. It is sound to attach any potential annotation $\vec{p}$ to empty data structures since the resulting potential is always zero.

(U:CONS) The rule U:CONS formalizes the fact that one has to pay for the resource consumption of the evaluation of $\mathit{cons}(x_h, x_t)$—that is, basically the allocation of a new heap-cell that points to x_h and x_t. This is represented by the constant K^{cons} that depends on the resource that is studied. In addition one has to pay for the potential that is assigned to the new list of type $L^{\vec{p}}(A)$. We do so by requiring x_t to have the type $L^{\lhd(\vec{p})}(A)$ and to have p_1 resource units available. It corresponds exactly to the recursive definition of the potential function Φ and ensures that potential is neither gained nor lost.

(U:MATL) The rule U:MATL defines how to use the potential of a list to pay for resource consumptions. First, it matches the corresponding rules E:MATCONS and E:MATNIL from the operational semantics in terms of resource consumption and incorporates the fact that either e_1 or e_2 is evaluated. More interestingly, the cons case is inverse to the rule U:CONS and allows one to use the potential associated with a list. For one thing, p_1 resource units become available directly, for another the tail of the list is annotated with $\lhd(\vec{p})$ rather than $\vec{p}$, permitting for example a recursive call requiring annotation $\vec{p}$ and an additional use of the tail with annotation $(p_2, \ldots, p_k)$.

The rules U:LEAF. U:NODE and U:MATT are similar to U:NIL, U:CONS and U:MATL, respectively. By way of example, I describe U:MATT in detail.

(U:MATT) The rule U:MATT shows how the potential of a tree is divided to pay for resource consumptions. The initial potential $\Phi(\Gamma) + \Phi(x{:}T^{\vec{p}}(A)) + q$ must be sufficient to pay for the resource consumption of the evaluation of e_1 and the cost K_1^{matTL} of the pattern match in this case. It must also be sufficient to pay for the evaluation of e_2 the cost K_1^{matTN} of the pattern match in that case. The potential after each evaluation must

be sufficient to pay for the potential of the result and for K_2^{matTL} or K_2^{matTN}, respectively. In the case of e_2, we can use the initial potential $\Phi(\Gamma) + \Phi(x_0{:}A) + \Phi(x_1{:}T^{\lhd(\vec{p})}(A)) + \Phi(x_2{:}T^{\lhd(\vec{p})}(A)) + q + p_1 - K_1^{\text{matTN}}$ to pay for the evaluation of e_2. This corresponds again to the recursive definition of the potential function Φ. In this way, potential is neither gained nor lost. The initial potential can be used similar as for lists. For one thing, p_1 resource units become available directly, for another the subtrees of the matched tree are annotated with $\lhd(\vec{p})$, permitting a recursive call (requiring the annotation $\vec{p}$) for every subtree and an additional use of the subtrees with the annotation $(p_2, \ldots, p_k)$.

5.3 Soundness

As for the linear system, I prove that univariate annotated type derivations establish correct bounds.

Assume, that we have derived an annotated type judgment for an expression e by using the rules from Section 5.2 and that e evaluates to a value v in a well-formed environment. Then the initial potential of the context in the type judgment in that environment is an upper bound on the watermark of the resource usage during the evaluation. Furthermore, the difference between the initial and the final potential is an upper bound on the consumed resources.

Using the partial evaluation rules, we can moreover prove that the bounds derived from annotated type judgments also apply to non-terminating evaluations. Additionally, the novel way of cost monitoring in the operational semantics enables a concise statement.

Theorem 5.3.1 (Soundness) Let $H \vDash V : \Gamma$ and let $\Sigma; \Gamma \vdash^{q}_{q'} e{:}B$.

1. If $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ then $p \leq \Phi_{V,H}(\Gamma) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q')$.

2. If $V, H \vdash e \rightsquigarrow \mid p$ then $p \leq \Phi_{V,H}(\Gamma) + q$.

It follows from Theorem 5.3.1 and Theorem 3.3.9 that run-time bounds also prove termination of programs. Corollary 5.3.2 states this fact formally.

Corollary 5.3.2 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $H \vDash V : \Gamma$ and $\Sigma; \Gamma \vdash^{q}_{q'} e{:}A$ then there is an $n \in \mathbb{N}$, $n \leq \Phi_{V,H}(\Gamma) + q$ such that $V, H \vdash e \rightsquigarrow v, H' \mid (n, 0)$.

Note that the formulation of Theorem 5.3.1 is identical to the formulation of Theorem 4.3.1, its linear equivalent. However, it makes a stronger statement since it refers to the univariate polynomial type system of this chapter.

As for linear version, I prove the soundness theorem by a nested induction on the derivation of the evaluation judgment—$V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ or $V, H \vdash e \rightsquigarrow \mid p$, respectively—and the type judgment $\Sigma; \Gamma \vdash^{q}_{q'} e{:}B$. The inner induction on the type

judgment is needed because of the structural rules (compare the discussion in the proof of Theorem 3.3.4).

Formally, I cannot build on Theorem 4.3.1 to prove Theorem 5.3.1. But many of the cases in the proofs are similar. In fact, I could copy the entire proof except of the cases that directly involve the univariate potential annotations; namely U:LEAF, U:NODE, U:MATT, U:NIL, U:CONS, and U:MATL.

The same is true for Lemma 5.3.3, which is the polynomial equivalent to Lemma 4.3.3. It is needed to show the soundness of the rule U:LET and states that the potential of a context is invariant during the evaluation. This is a consequence of allocated heap-cells being immutable with the language features that I describe in this thesis.

Lemma 5.3.3 Let $H \vDash V{:}\Gamma$, $\Sigma;\Gamma \vdash^{q}_{q'} e{:}A$ and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$. Then $\Phi_{V,H}(\Gamma) = \Phi_{V,H'}(\Gamma)$.

PROOF The lemma is a direct consequence of the definition of the potential Φ and the fact that $H'(\ell) = H(\ell)$ for all $\ell \in \mathrm{dom}(H)$, which is proved in Proposition 3.3.2. ∎

Proof of the Soundness Theorem

In the remainder of this section I prove Theorem 5.3.1. Large parts of the proof are identical to the proof of Theorem 4.3.1, the soundness theorem for the linear type system. So I refer to the proof of Theorem 4.3.1 and only provide the parts of the proof that differ from the linear case. This includes all parts that directly involve the polynomial potential annotations. Additionally, I only provide the arguments for the more involved proof of part 1. Again, the proof of part 2 is almost identical the proof of part 2 of Theorem 4.3.1.

PROOF (PART 1) I prove $p \leq \Phi_{V,H}(\Gamma) + q$ and $p - p' \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q')$ by induction on the derivations of $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ and $\Sigma;\Gamma \vdash^{q}_{q'} e : B$, where the induction on the evaluation judgment takes priority.

(U:NIL) If the type derivation ends with an application of U:NIL then we have $e = nil$, $B = L^{\vec{r}}(A)$ for some A, and $0 \leq q' = q - K^{\mathrm{nil}}$. The corresponding evaluation rule E:NIL has been applied to derive the evaluation judgment and hence $v = \mathrm{NULL}$. If $K^{\mathrm{nil}} \geq 0$ then $p = K^{\mathrm{nil}}$ and $p' = 0$. Thus $p = K^{\mathrm{nil}} \leq q = \Phi_{V,H}(\emptyset) + q$. Furthermore, it follows from the definition of Φ that $\Phi_{H'}(\mathrm{NULL}{:}L^{\vec{r}}(A)) = 0$. Thus $p - p' = K^{\mathrm{nil}} = \Phi_{V,H}(\emptyset) + q - (\Phi_{H'}(\mathrm{NULL}{:}L^{\vec{r}}(A)) + q')$. If $K^{\mathrm{nil}} < 0$ then $p = 0$ and $p' = -K^{\mathrm{nil}}$. Then $p \leq q$ and again $p - p' = K^{\mathrm{nil}}$.

(U:CONS) If the type derivation ends with an application of the rule U:CONS then e has the form $cons(x_h, x_t)$ and has been evaluated with the rule E:CONS. It follows by definition that $V, H \vdash cons(x_h, x_t) \rightsquigarrow \ell, H[\ell \mapsto v'] \mid K^{\mathrm{cons}}$, $x_h, x_t \in \mathrm{dom}(V)$, $v' = (V(x_h), V(x_t))$, and $\ell \notin \mathrm{dom}(H)$. Thus

$$p = K^{\mathrm{cons}} \text{ and } p' = 0 \tag{5.3}$$

or (if $K^{\text{cons}} < 0$)

$$p = 0 \text{ and } p' = -K^{\text{cons}} \tag{5.4}$$

Furthermore $B = L^{\vec{s}}(A)$ and the type judgment $\Sigma; x_h{:}A, x_t{:}L^{\lhd(\vec{s})}(A) \vdash^{q}_{q'} \mathit{cons}(x_h, x_t) : L^{\vec{s}}(A)$ has been derived by a single application of the rule U:CONS; thus

$$0 \le q' = q - s_1 - K^{\text{cons}}\,. \tag{5.5}$$

If $p = 0$ then $p \le \Phi_{V,H}(\Gamma) + q$ holds because of the implicit side condition $q \ge 0$. Otherwise we have $p = K^{\text{cons}} \le q \le \Phi_{V,H}(\Gamma) + q$.

From the definition of Φ it follows that

$$s_1 + \Phi_{V,H}(x_h{:}A, x_t{:}L^{\lhd(\vec{s})}(A)) = \Phi_{V,H[\ell \mapsto v']}(\ell : L^{\vec{s}}(A)) \tag{5.6}$$

Therefore

$$\begin{array}{rcl} \Phi_{V,H}(\Gamma) + q & = & \Phi_{V,H}(x_h{:}A, x_t{:}L^{\lhd(\vec{s})}(A)) + q \\ & \overset{(5.5)}{=} & \Phi_{V,H}(x_h{:}A, x_t{:}L^{\lhd(\vec{s})}(A)) + q' + s_1 + K^{\text{cons}} \\ & \overset{(5.6)}{=} & q' + K^{\text{cons}} + \Phi_{V,H[\ell \mapsto v']}(\ell : L^{\vec{s}}(A)) \end{array}$$

and thus $\Phi_{V,H}(\Gamma) + q - (\Phi_{V,H[\ell \mapsto v']}(\ell{:}L^{\vec{s}}(A)) + q') = K^{\text{cons}} = p - p'$.

(U:MATL) Assume that the type derivation of e ends with an application of the rule U:MATL. Then e is a pattern match of the form $\mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2$ whose evaluation ends with an application of the rule E:MATCONS or E:MATNIL. Assume first that the derivation of the evaluation judgment ends with an application of E:MATCONS.

Then $V(x) = \ell$, $H(\ell) = (v_h, v_t)$, and $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ for $V' = V[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r, r' with

$$(p, p') = K_1^{\text{matC}} \cdot (r, r') \cdot K_2^{\text{matC}} \tag{5.7}$$

Since the derivation of $\Sigma; \Gamma \vdash^{q}_{q'} e{:}B$ ends with an application of U:MATL, we have $\Gamma = \Gamma', x{:}L^{\vec{t}}(A)$, $\Sigma; \Gamma', x_h{:}A, x_t{:}L^{\lhd(\vec{t})}(A) \vdash^{s}_{s'} e_2 : B$ and,

$$q = s + K_1^{\text{matC}} - t_1 \text{ and } q' = s' - K_2^{\text{matC}}\,. \tag{5.8}$$

It follows from the definition of Φ that $\Phi_H(v{:}L^{\vec{t}}(A)) = t_1 + \Phi_H(v_h{:}A) + \Phi_H(v_t{:}L^{\lhd(\vec{t})}(A))$ and therefore

$$\Phi_{V,H}(\Gamma) = t_1 + \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L^{\lhd(\vec{t})}(A))\,. \tag{5.9}$$

Since $H \vDash V' : \Gamma', x_h{:}A, x_t{:}L^{\lhd(\vec{t})}(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ and obtain (with (5.9))

$$r \le \Phi_{V,H}(\Gamma) - t_1 + s \tag{5.10}$$

$$r - r' \le \Phi_{V,H}(\Gamma) - t_1 + s - (\Phi_{H'}(v{:}B) + s') \tag{5.11}$$

Note that $\Phi_{V,H}(\Gamma) - t_1 \geq 0$ and let

$$(u, u') = K_1^{\text{matC}} \cdot (\Phi_{V,H}(\Gamma) - t_1 + s, \Phi_{H'}(v{:}B) + s') \cdot K_2^{\text{matC}} . \tag{5.12}$$

Per definition and from (5.8) it follows that $u = \max(0, \Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matC}})$. From Proposition 3.3.1 applied to (5.10), (5.12) and (5.7) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matC}} \leq 0$ then $u = p = 0$ and $q + \Phi_{V,H}(\Gamma) \geq p$ trivially holds. If $\Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matC}} > 0$ then it follows from (5.8) that

$$q + \Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matC}} = u \geq p .$$

Finally, we apply Proposition 3.3.1 to (5.7) to see that

$$\begin{array}{rcl} p - p' & = & r - r' + K_1^{\text{matC}} + K_2^{\text{matC}} \\ & \overset{(5.11)}{\leq} & \Phi_{V,H}(\Gamma) - t_1 + s - (\Phi_{H'}(v{:}B) + s') + K_1^{\text{matC}} + K_2^{\text{matC}} \\ & = & \Phi_{V,H}(\Gamma) + (s + K_1^{\text{matC}} - t_1) - (\Phi_{H'}(v{:}B) + (s' - K_2^{\text{matC}})) \\ & \overset{(5.8)}{\leq} & \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q') \end{array}$$

Assume now that the derivation of the evaluation judgment ends with an application of E:MatNil. Then $V(x) = \text{Null}$, and $V, H \vdash e_1 \rightsquigarrow v, H' \mid (r, r')$ for some r, r' with

$$(p, p') = K_1^{\text{matN}} \cdot (r, r') \cdot K_2^{\text{matN}} . \tag{5.13}$$

Since the derivation of $\Sigma; \Gamma \vdash^{\frac{q}{q'}} e{:}B$ ends with an application of U:MatL, we have $\Sigma; \Gamma \vdash^{\frac{s}{s'}} e_1 : B$ and

$$q = s + K_1^{\text{matN}} \text{ and } q' = s' - K_2^{\text{matN}} . \tag{5.14}$$

Because $H \vDash V : \Gamma$ we can apply the induction hypothesis to $V, H \vdash e_1 \rightsquigarrow v, H' \mid (r, r')$ and obtain

$$r \leq \Phi_{V,H}(\Gamma) + s \tag{5.15}$$

$$r - r' \leq \Phi_{V,H}(\Gamma) + s - (\Phi_{H'}(v{:}B) + s') \tag{5.16}$$

Now let

$$(u, u') = K_1^{\text{matN}} \cdot (\Phi_{V,H}(\Gamma) + s, \Phi_{H'}(v{:}A) + s') \cdot K_2^{\text{matN}} . \tag{5.17}$$

Per definition and from (5.14) it follows that $u = \max(0, \Phi_{V,H}(\Gamma) + s + K_1^{\text{matN}})$. From Proposition 3.3.1 applied to (5.15), (5.17) and (5.13) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma) + s + K_1^{\text{matN}} \leq 0$ then $u = p = 0$ and $q + \Phi_{V,H}(\Gamma) \geq p$ trivially holds. If $\Phi_{V,H}(\Gamma) + s + K_1^{\text{matN}} > 0$ then it follows from (5.14) that

$$q + \Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma) + s + K_1^{\text{matN}} = u \geq p .$$

Finally, we apply Proposition 3.3.1 to (5.13) to see that

$$\begin{array}{rcl} p - p' & = & r - r' + K_1^{\text{matN}} + K_2^{\text{matN}} \\ & \overset{(5.16)}{\leq} & \Phi_{V,H}(\Gamma) + s - (\Phi_{H'}(v{:}B) + s') + K_1^{\text{matN}} + K_2^{\text{matN}} \\ & = & \Phi_{V,H}(\Gamma) + (s + K_1^{\text{matN}}) - (\Phi_{H'}(v{:}B) + (s' - K_2^{\text{matN}})) \\ & \overset{(5.14)}{\leq} & \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v{:}B) + q') \end{array}$$

(U:LEAF) This case is nearly identical to the case (U:NIL).

(U:NODE) If the type derivation ends with an application of the rule U:NODE then e has the form $\mathit{node}(x_0, x_1, x_2)$ and it has been evaluated with the rule E:NODE. It follows by definition that $V, H \vdash \mathit{node}(x_0, x_1, x_2) \rightsquigarrow \ell, H[\ell \mapsto v' \mid K^{\text{node}},\ x_0, x_1, x_2 \in \text{dom}(V)$, $v = (V(x_0), V(x_1), V(x_2))$, and $\ell \notin \text{dom}(H)$. Thus (if $K^{\text{node}} \geq 0$)

$$p = K^{\text{node}} \text{ and } p' = 0 \tag{5.18}$$

or (if $K^{\text{node}} < 0$)

$$p = 0 \text{ and } p' = -K^{\text{node}}\,. \tag{5.19}$$

Furthermore $B = T^{\vec{s}}(A)$ and the type judgment

$$\Sigma; x_0{:}A, x_1{:}T^{\lhd(\vec{s})}(A), x_2{:}T^{\lhd(\vec{s})}(A) \vdash^{q}_{q'} \mathit{node}(x_0, x_1, x_2) : T^{\vec{s}}(A)$$

has been derived by a single application of the rule U:NODE; thus

$$0 \leq q' = q - s_1 - K^{\text{node}}\,. \tag{5.20}$$

If $p = 0$ then $p \leq \Phi_{V,H}(\Gamma) + q$ holds because of the implicit side condition $q \geq 0$. Otherwise we have $p = K^{\text{node}} \leq q \leq \Phi_{V,H}(\Gamma) + q$.

From the definition of Φ it follows that

$$s_1 + \Phi_{V,H}(x_0{:}A, x_1{:}T^{\lhd(\vec{s})}(A), x_2{:}T^{\lhd(\vec{s})}(A)) = \Phi_{H[\ell \mapsto v']}(\ell : T^{\vec{s}}(A)) \tag{5.21}$$

Therefore

$$\begin{array}{rcl} \Phi_{V,H}(\Gamma) + q & = & \Phi_{V,H}(x_0{:}A, x_1{:}T^{\lhd(\vec{s})}(A), x_2{:}T^{\lhd(\vec{s})}(A)) + q \\ & \overset{(5.20)}{=} & \Phi_{V,H}(x_0{:}A, x_1{:}T^{\lhd(\vec{s})}(A), x_2{:}T^{\lhd(\vec{s})}(A)) + q' + s_1 + K^{\text{node}} \\ & \overset{(5.21)}{=} & q' + K^{\text{node}} + \Phi_{H[\ell \mapsto v']}(\ell : T^{\vec{s}}(A)) \end{array}$$

and thus $\Phi_{V,H}(\Gamma) + q - (\Phi_{H[\ell \mapsto v']}(\ell{:}T^{\vec{s}}(A)) + q') = K^{\text{node}} = p - p'$.

(U:MATT) Assume that the type derivation of e ends with an application of the rule U:MATT. Then e is a pattern match $\mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2$ whose evaluation ends with an application of the rule E:MATNODE or E:MATLEAF. The

case E:MATLEAF is similar to the case E:MATNIL. So assume that the derivation of the evaluation judgment ends with an application of E:MATNODE.

Then $V(x) = \ell$, $H(\ell) = (v_0, v_1, v_2)$, and $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ for $V' = V[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2]$ and some r, r' with

$$(p, p') = K_1^{\text{matTL}} \cdot (r, r') \cdot K_2^{\text{matTL}} \tag{5.22}$$

Since the derivation of $\Sigma; \Gamma \vdash^{\frac{q}{q'}} e{:}B$ ends with an application of U:MATT, we have $\Gamma = \Gamma', x{:}T^{\vec{t}}(A)$, $\Sigma; \Gamma', x_0{:}A, x_1{:}T^{\lhd(\vec{t})}(A), x_2{:}T^{\lhd(\vec{t})}(A) \vdash^{\frac{s}{s'}} e_2 : B$, and

$$q = s + K_1^{\text{matTL}} - t_1 \text{ and } q' = s' - K_2^{\text{matTL}} . \tag{5.23}$$

It follows from the definition of Φ that $\Phi_H(v{:}T^{\vec{t}}(A)) = t_1 + \Phi_H(v_0{:}A) + \Phi_H(v_1{:}T^{\lhd(\vec{t})}(A)) + \Phi_H(v_2{:}T^{\lhd(\vec{t})}(A))$ and therefore

$$\Phi_{V,H,V}(\Gamma) = t_1 + \Phi_{V,H,V'}(\Gamma', x_0{:}A, x_1{:}T^{\lhd(\vec{t})}(A), x_2{:}T^{\lhd(\vec{t})}(A)) . \tag{5.24}$$

Because we have $H \vDash V' : \Gamma', x_0{:}A, x_1{:}T^{\lhd(\vec{t})}(A), x_2{:}T^{\lhd(\vec{t})}(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ and obtain (with (5.24))

$$\begin{aligned} r &\leq \Phi_{V,H}(\Gamma) - t_1 + s && (5.25)\\ r - r' &\leq \Phi_{V,H}(\Gamma) - t_1 + s - (\Phi_{V,H'}(v{:}B) + s') && (5.26) \end{aligned}$$

Since the matched tree contains at least one node, we have $\Phi_{V,H}(\Gamma) - t_1 \geq 0$. Let

$$(u, u') = K_1^{\text{matTL}} \cdot (\Phi_{V,H}(\Gamma) - t_1 + s, \Phi_{V,H'}(v{:}A) + s') \cdot K_2^{\text{matTL}} . \tag{5.27}$$

Per definition and from (5.23) it follows that $u = \max(0, \Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matTL}})$. From Proposition 3.3.1 applied to (5.25), (5.27), and (5.22) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matTL}} \leq 0$ then $u = p = 0$ and $q + \Phi_{V,H}(\Gamma) \geq p$ trivially holds. If $\Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matTL}} > 0$ then it follows from (5.23) that

$$q + \Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma) - t_1 + s + K_1^{\text{matTL}} = u \geq p .$$

Finally, we apply Proposition 3.3.1 to (5.22) to see that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{matTL}} + K_2^{\text{matTL}} \\ &\overset{(5.26)}{\leq} \Phi_{V,H}(\Gamma) - t_1 + s - (\Phi_{V,H'}(v{:}B) + s') + K_1^{\text{matTL}} + K_2^{\text{matTL}} \\ &= \Phi_{V,H}(\Gamma) + (s + K_1^{\text{matTL}} - t_1) - (\Phi_{V,H'}(v{:}B) + (s' - K_2^{\text{matTL}})) \\ &\overset{(5.23)}{\leq} \Phi_{V,H}(\Gamma) + q - (\Phi_{V,H'}(v{:}B) + q') \end{aligned}$$

■

5.4 Type Inference

The basis of the type inference for the univariate polynomial system is type inference algorithm for the linear system which is described in Section 4.4.

A further challenge for the inference of polynomial bounds is the need to deal with *resource-polymorphic recursion*, which is required to type most programs that are not tail recursive. It seems to be a hard problem to infer general resource-polymorphic typings, even for the original linear system.

In Section 5.4.2, I present a pragmatic approach to resource-polymorphic recursion that works well and efficiently in practice. It infers types for most functions that admit a type-derivation, including all useful programs that we implemented. Nevertheless, it is not complete with respect to the general resource-polymorphic typing rules. Section 5.4.3 contains a somewhat artificial function with a linear heap-space consumption that admits a resource-polymorphic typing that can neither be inferred by the algorithm I present here nor in the classic linear system [HJ03].

To begin with, I explain in Section 5.4.1 by example why resource-polymorphic recursion is needed frequently in the polynomial system and informally introduce the idea of the inference algorithm.

5.4.1 Resource-Polymorphic Recursion

Recall the function *attach* that has been introduced in Section 2.2.1. It takes an integer and a list of integers and returns a list of pairs of integers in which the first argument is paired with each element of the list.

```
attach(x,l) = match l with | nil  →  nil
                           | (y::ys)  →  (x,y)::(attach (x,ys))
```

To infer the potential annotations for attach we use the inference algorithm for the linear system from Section 4.4. First, we annotate the type of *attach* with a priori unknown resource-annotations s, s', q and p that range over non-negative rational numbers.

$$attach \quad : \quad (int, L^q(int)) \xrightarrow{s/s'} L^p(int, int)$$

We then use the type system to derive linear constraints on the potential annotations. To informally explain the constraints for *attach*, expressions of type list are annotated with variables $q, p, r, \ldots$ that range over $\mathbb{Q}_0^+$. The intended meaning of e^q is that e is of type $L^q(A)$ for some type A.

$$\begin{array}{l}\texttt{attach(x,l}^q\texttt{) = match l}^{q'}\texttt{ with | nil } \rightarrow \texttt{ nil}^p \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{| (y::ys}^r\texttt{) } \rightarrow \texttt{ ((x,y)::(attach (x,ys}^q\texttt{))}^p\texttt{)}^p\end{array}$$

If we assume that a list element for a pair of integers has size 3 (two cells to store the integers and one for the pointer to the next element) then the heap-space usage of an evaluation of *attach(x,l)* is $3|\ell|$ memory cells.

The syntax-directed inference then computes inequalities like $q' + s \geq 3 + p + s$. It expresses the fact that the potential q' of the first list element and the initial potential s

must cover the costs for the cons operation (3 memory cells), the potential p of a list element of the result, and the input potential s of the recursive call.

To pay the cost during the recursion we require the annotation of the function arguments and the result of the recursive call to match their specification ($s = q$ and $t = p$ in the case of *attach*). The function is then used *resource-monomorphically*, that is, with the same annotations as in the result and the arguments of the outer call.

Note that there are functions with linear resource usage that cannot be typed resource-monomorphically. You find an example in Section 5.4.3. Nevertheless, the inference algorithm for the linear system from Section 4.4 infers resource-monomorphic type derivations only. This is unproblematic since most linearly bounded functions that appear in practice do not require resource-polymorphic recursion.

In contrast, many non-tail-recursive functions with a super-linear resource behavior can often be typed *resource-polymorphically* only, that is, with different resource annotations in the recursive calls.

To understand why, consider the function *pairs* from Section 2.2.2, which computes the two-element subsets of a given set. It allows for a resource-monomorphic type derivation but can be turned into a function that needs resource-polymorphic recursion by a small modification.

```
pairs l = match l with | nil  → nil
            | (x::xs)  → append(attach(x,xs),pairs xs)
```

The evaluation of the expression *pairs(l)* consumes 6 memory cells per element of every sub-list (suffix) of ℓ. The type that our system infers resource-monomorphically for *pairs* is $L^{(0,6)}(int) \xrightarrow{0/0} L^{(0)}(int, int)$.

To infer the potential annotations, we start with an annotation of the list types with resource variables as before.

```
pairs l = match l^(q1,q2) with | nil  → nil
            | (x::xs^(p1,p2))  → append(attach(x,xs^(r1,r2)),pairs xs^(s1,s2))
```

The constraints that our type system computes include $q_2 \geq p_2$ and $q_1+q_2 \geq p_1$ (additive shift); $p_1=r_1+s_1$ and $p_2=r_2+s_2$ (sharing between two variables); $r_1 \geq 6$ (pay for non-recursive function calls); $q_1=s_1$, $q_2=s_2$ (pay for the recursive call). This system is solvable by $q_2 = s_2 = p_1 = p_2 = r_1 = 6$ and $q_1 = s_1 = r_2 = 0$.

As in the linear case, we require in the constraint system that the type of the recursive call to *pairs* matches its specification ($q_i = s_i$). Since the resulting constraint system is solvable, the function *pairs* can be typed resource-monomorphically. But in contrast to the linear case, such a resource-monomorphic approach results in an unsolvable linear program for many non-tail-recursive functions with a super linear resource behavior.

Consider for example the function *pairs'* that is a modification of *pairs* in which we permute the arguments of *append* and hence replace the expression in the cons-branch of the pattern match with *append(pairs' xs,attach(x,xs))*.

```
pairs' l = match l with | nil  → nil
             | (x::xs)  → append(pairs' xs,attach(x,xs))
```

The heap-space usage of *pairs'* is $3\binom{n}{2}+3\binom{n}{3}$ since *append* is called with the intermediate results of *pairs'* in the first argument and thus consumes $\sum_{2\le i<n}\binom{i}{2}=\binom{n}{3}$ memory cells.

A resource-polymorphic type derivation establishes an exact heap-space bound for the function *pairs'* by establishing the typing *pairs'*: $L^{(0,3,3)}(int)\xrightarrow{0/0}L^{(0)}(int,int)$. Similar to the case of *pairs*, the additive shift assigns the type $L^{(3,6,3)}(int)$ to *xs* in the cons-branch. The linear potential $xs{:}L^{(3,0,0)}(int)$ is passed on to the occurrence of *xs* in *attach*. But in order to pay the costs of *append* we have to assign a linear potential to the result of the recursive call and thus use the alternate typing *pairs'*: $L^{(0,6,3)}(int)\xrightarrow{0/0}L^{(3)}(int,int)$.

The need of passing on potential of degree at most $k-1$ to the output of a function with a resource consumption of degree k is quite common in typical functions. It is present in the derivation of time bounds for most non-tail-recursive functions that we considered, for example, quick sort and insertion sort. The classic (resource-monomorphic) inference approach of requiring the type of the recursive call to match its specification fails for these functions and it was a non-trivial problem to address it with an efficient solution.

Inference with Cost-Free Types

Our pragmatic approach to infer type derivations with resource-polymorphic recursion is the use of the special *cost-free* resource metric that assigns zero costs to every evaluation step. A cost-free function type *f*: $A\xrightarrow{a/a'}B$ then describes how to pass potential from x to $f(x)$ without paying for resource usage. Any concrete typing for a given resource metric can be superposed with a *cost-free* typing to obtain another typing for the given resource metric. This is similar to the solution of inhomogeneous systems by superposition with homogeneous solutions in linear algebra.

I illustrate the idea using *pairs'* again. First, we derive the cost-free types *attach:* $(int,L^{(3)}(int))\xrightarrow{0/0}L^{(3)}(int,int)$ and *append:* $(L^{(3)}(int,int),L^{(3)}(int,int))\xrightarrow{0/0}L^{(3)}(int,int)$. The type inference for, say, *attach* works as outlined above with the inequality $q'+s\ge 3+p+s$ replaced with $q'+s\ge p+s$. Similar, we can assign *pairs'* the cost-free type $L^{(0,3)}(int)\xrightarrow{0/0}L^{(3)}(int,int)$. The typing $xs{:}L^{(3,3)}(int)$ that results from the additive shift is then used as $xs{:}L^{(3,0)}(int)$ in *attach* and as $xs{:}L^{(0,3)}(int)$ in the recursive call.

If we now aim to infer the type of a function with respect to some cost metric then we deal with recursive calls by requiring them to match the type specification of the function and to optionally pass potential to the result via a cost-free type. The cost-free type is then inferred resource-monomorphically. In the case of the heap-space consumption of *pairs'* we would first infer that the recursive call has to be of the form $L^{(0+q_1,3+q_2,3)}(int)\to L^{(0+p_1)}(int,int)$, where $L^{(q_1,q_2)}(int)\to L^{(p_1)}(int,int)$ is a cost-free type. We then infer like in the linear case that $q_1=0$ and $q_2=p_1=3$.

This method cannot infer every resource-polymorphic typing with respect to declarative type derivations with polymorphic recursion. This would mean to start with a (possibly infinite) set of annotated types for each function and to justify each function type with a type derivation that uses types from the initial set. With respect to this

declarative view, the inference algorithm in this section can compute every set of types for a function f that has the form

$$\Sigma(f) = \{T + q \cdot T_i \mid q \in \mathbb{Q}_0^+, 1 \le i \le m\}$$

for a resource-annotated function of type T, cost-free function types T_i, and m recursive calls of f in its function body. Since many resource-polymorphic type derivations feature a set of function types of this format, this approach leads to an effective inference method.

5.4.2 Inference Algorithm

The inference algorithm is mainly defined by algorithmic versions of the type rules from Section 5.2, which are described in detail in a conference paper [HH10a]. Like in the linear case, it works like a standard type inference in which each type is annotated with resource variables and the corresponding linear constraints are collected as each type rule is applied.

Algorithmic Type Rules

The derivation of the algorithmic rules is similar as described in Section 4.4. The main innovation in comparison to the classic algorithm for the linear system [HJ03] is the resource-polymorphic recursion enabled by the algorithmic versions of the rule U:APP.

$$\frac{\Sigma(f) = A \xrightarrow{p/p'} B \qquad q = p + c + K_1^{\text{app}} \qquad q' = p' + c - K_2^{\text{app}}}{\Sigma; x{:}A \vdash^{q}_{q'} f(x) : B} \text{ (A:APPCF)}$$

$$\frac{\begin{array}{l} q = p + p_{cf} + c + K_1^{\text{app}} \qquad q' = p' + p'_{cf} + c - K_2^{\text{app}} \qquad A \curlyvee (A', A_{cf}) \qquad B \curlyvee (B', B_{cf}) \\ \Sigma_{cf}; y_f{:}A_{cf} \vdash^{p_{cf}}_{p'_{cf}} e_f{:}B_{cf} \qquad \Sigma_{cf}(f) = A_{cf} \xrightarrow{p_{cf}/p'_{cf}} B_{cf} \qquad \Sigma(f) = A' \xrightarrow{p/p'} B' \end{array}}{\Gamma, x{:}A \vdash^{q}_{q'} f(x) : B} \text{ (A:APP)}$$

The rule A:APPCF is essentially the rule U:APP from section Section 5.2. It is used for the cost-free metric and leads to a resource-monomorphic typing of recursive calls.

The rule A:APP is used for function applications in all other resource metrics and enables resource-polymorphic recursion. It states that one can add any cost-free typing of the function body to the function type that is given by the signature Σ. Note that $(e_f, y_f)_{f \in \text{dom}(\Sigma_{cf})}$ must be a valid RAML program with cost-free types of smaller degree. The annotated signature Σ_{cf} used can differ in every application of the rule.

The idea is as follows. In order to pay for the resource costs of a function call $f(x)$, the available potential $(\Phi(x{:}A) + q)$ must meet the requirements of the signature of the function $(\Phi(x{:}A') + p)$. Additionally available potential $(\Phi(x{:}A_{cf}) + p_{cf})$ can be passed to a cost-free typing of the function body. The potential after the function call

$(\Phi(f(x){:}B) + q')$ is then the sum of the potentials that are assigned by the cost-free typing $(\Phi(f(x){:}B_{cf}) + p_{cf})$ and by the function signature $(\Phi(f(x){:}B') + p)$. As a result, $f(x)$ can be used resource-polymorphically with a specific typing for each recursive call while the resource monomorphic function signature enables an efficient type inference.

The Algorithm

To ensure that the constraint system is finite the user has to provide a maximal degree of the bounds in the search space. The number of computed constraints grows linearly in the maximal degree that has been provided by the user.

There is a trade-off between the quality of the analysis and the size of the constraint system. The reason is that one sometimes has to analyze function applications context-sensitively with respect to the call stack.

In our implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph. In a nutshell, the algorithm computes inequalities for annotations of degree k for a strongly connected component (SCC) F of the call graph as follows.

1. Annotate the signature of each function $f \in F$ with fresh resource variables.
2. Use the algorithmic type rules [HH10a] to type the corresponding expressions e_f. Introduce fresh resource variables for each type annotation in the derivation and collect the corresponding inequalities.
   (a) For a function application $g \in F$: if the maximal degree is 1 or in the cost-free case use the function resource-monomorphically with the signature from (1) using the rule A:APPCF. Otherwise, go to (1) and derive a cost-free typing of e_g with a fresh signature. Store the arising inequalities and use the resource variables from the obtained typing together with the signature from (1) in the rule A:APP.
   (b) For a function application $g \notin F$: repeat the algorithm for the SCC of g. Store the arising inequalities and use the obtained annotated type of g.

The context sensitivity in the algorithm can lead to an exponential blow up of the constraint system if there is a sequence of function $f_1, \ldots, f_n$ such that f_i calls f_{i+1} several times. But such sequences are not very long in most programs. It would not be a substantial limitation in practice to restrict oneself to programs that feature a collapsed call graph with a fixed maximal path length to certainly obtain a constraint system that is linear in the program size.

5.4.3 Incompleteness

The inference algorithm works very efficiently and infers resource-polymorphic types for all programs that we manually typed in our system. However, it is not complete

with respect to full resource-polymorphism. This would mean to start with a (possibly infinite) set of annotated function types for each function and to justify each type with a type derivation that uses some first-order types from the initial set.

For example, the inference algorithm does not compute a resource-annotated type for the function *round*:$L(unit) \to L(unit)$ which computes a list of length $\max\{2^i - 1 \mid 2^i - 1 \leq n\}$ if n is the length of the input list. The function *round* is implemented in RAML as follows.

```
half l = match l with | nil  → nil
          | x1::xs  → match xs with | nil  → nil
                       | x2::xs'  → x1::(half xs')

double l = match l with | nil  → nil
            | x::xs  → x::x::(double xs)


round l = match l with | nil  → nil
           | x::xs  → x::double (round (half xs))
```

The function *half* deletes every second element and the function *double* doubles every element a list. With the cost-free metric, the following types can be (resource-monomorphically) inferred for *half* and *double*.

$$\begin{aligned} \mathit{half} &: L^1(\mathit{unit}) \xrightarrow{0/0} L^2(\mathit{unit}) \\ \mathit{double} &: L^2(\mathit{unit}) \xrightarrow{0/0} L^1(\mathit{unit}) \end{aligned}$$

The linear resource-annotated type systems allows the derivation of the typing

$$\mathit{round} \quad : \quad L^a(\mathit{unit}) \xrightarrow{0/0} L^a(\mathit{unit})$$

for every $a \in \mathbb{Q}_0^+$. In the derivation this function type for a given $a \in \mathbb{Q}_0^+$, we need the type resource-polymorphic type

$$\mathit{round} \quad : \quad L^{2a}(\mathit{unit}) \xrightarrow{0/0} L^{2a}(\mathit{unit})\,.$$

Since the linear cost-free type already requires resource polymorphism, our algorithm can not infer a typing for *round*. For every $q \in \mathbb{Q}_0^+$ one can create functions where one would need to multiply some resource annotations with q in cost-free typing of the recursive call. So it is unlikely that there is a method to infer a typing for such functions that uses only linear constraints.

To deal with them one could move to quadratic constraints to address the problem but the efficiency of such an approach is unclear.

5.5 Examples

In this section, I demonstrate the univariate polynomial analysis on example programs. To start with, I present a canonical family of functions with a univariate polynomial

resource analysis in Section 5.5.1. For each $k \geq 2$ there is a function that computes all subsets of size k from a given list if we view the list as a set.

In Section 5.5.2, I show that the analysis works well on the sorting algorithms quick sort, merge sort, insertion sort, and selection sort. To give a representative example of a program for which the analysis terminates without computing a bound, I also implement the sorting algorithm bubble sort.

Finally, I describe in Section 5.5.3 the analysis of a program that computes the transitive closure of a tree.

5.5.1 Subsets of Fixed Sizes

Canonical examples with polynomial heap-space consumption result from the following problem: view a given list as a set and compute the subsets of size k for a given k. The size of the output is a polynomial of degree k.

Below I define the subset functions for $k = 2$ and $k = 3$. You shall then see how it works for $k > 3$. The function *attach(x,l)* computes a list of pairs so that x is paired with every element in the list ℓ. The function *pairs(l)* computes a list of all (unordered) pairs that can be built from the elements of *l* and the function *triples(l)* computes a list of all (unordered) triples. For example, the expression *triples [1,2,3,4]* evaluates to *[(1,(2,3)),(1,(2,4)),(1,(3,4)),(2,(3,4))]*.

```
pairs: L(int) → L(int,int)

pairs(l) = match l with | nil  →  nil
           | x::xs  → append(attach(x,xs),pairs xs);

attach: (int,L(int)) → L(int,int)

attach(n,l) = match l with | nil  → nil
              | x::xs  → (n,x)::attach(n,xs);

append: (L(int,int),L(int,int)) → L(int,int)

append(l1,l2) = match l1 with | nil  → l2
                | x::xs  → x::append(xs,l2);

triples : L(int) → L(int,(int,int))

triples(l) = match l with | nil  → nil
              | x::xs  → append3(attach3(x,pairs xs),triples xs);

attach3: (int,L(int,int)) → L(int,(int,int))

attach3(n,l) = match l with | nil  → nil
               | x::xs  → (n,x)::attach3(n,xs);
```

```
append3: (L(int,(int,int)),L(int,(int,int))) → L(int,(int,int))

append3(l1,l2) = match l1 with | nil → l2
                               | x::xs → x::append3(xs,l2);
```

Since the heap-space consumption of *attach* and *append* depends on their types, I implemented one version of the functions for every type that is needed. The code of the functions is however identical. It would also be possible to allow polymorphic functions and to analyze them once for each concrete data type they are used with.

The following resource-annotated types are computed with the heap-space metric.

$$
\begin{array}{rcl}
\mathit{pairs} & : & L^{(0,6,0)}(\mathit{int}) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},\mathit{int}) \\
\mathit{triples} & : & L^{(0,0,14)}(\mathit{int}) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},(\mathit{int},\mathit{int})) \\
\mathit{attach} & : & (\mathit{int}, L^{(3,0,0)}(\mathit{int})) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},\mathit{int}) \\
\mathit{attach3} & : & (\mathit{int}, L^{(4,0,0)}(\mathit{int},\mathit{int})) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},(\mathit{int},\mathit{int})) \\
\mathit{append} & : & (L^{(3,0,0)}(\mathit{int},\mathit{int}), L^{(0,0,0)}(\mathit{int},\mathit{int})) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},\mathit{int}) \\
\mathit{append3} & : & (L^{(4,0,0)}(\mathit{int},(\mathit{int},\mathit{int})), L^{(0,0,0)}(\mathit{int},(\mathit{int},\mathit{int}))) \xrightarrow{0/0} L^{(0,0,0)}(\mathit{int},(\mathit{int},\mathit{int}))
\end{array}
$$

The computed heap-space bounds for the functions *pairs* and *triples* are $3n^2-3n$ and $2.\bar{3}n^3-7n^2+4.\bar{6}n$, respectively. Our experiments (see Chapter 7) show that the computed bounds match exactly the measured resource consumption of the functions.

5.5.2 Sorting

A classic way to demonstrate quantitative resource analysis is to analyze the run-time behavior of sorting algorithms. In the book *The Art of Computer Programming* [Knu97], Knuth manually determines worst-case bounds for many well-known sorting algorithms that are implemented in an assembly language for the MIX architecture. Among the analyzed algorithms are quick sort, which uses at most $2n^2+37n+3$ MIX cycles, insertion sort, at most $9\binom{n}{2}+7n-6 = 4.5n^2+2.5n-6$ MIX cycles, selection sort, at most $5\binom{n}{2}+3\lfloor\frac{n^4}{2}\rfloor+12n-11$,and merge sort, roughly $10n\log n+4.92n$ MIX cycles[1] (n is the size of the input).

As a result of a careful and elaborate analysis, the bounds are tight in the sense that they exactly match the actual worst-case behavior of the functions.

In the remainder of this section I implement the four sorting algorithms in RAML to automatically determine a bound on the number of evaluation steps they use. The experimental evaluation that I present in Chapter 7 shows that the computed bounds for insertion sort and quick sort exactly match the measured worst-case behavior of the functions. The bound for selections sort is asymptotically tight and the constant factors are quite precise. The bound for merge sort is quadratic but the actual worst-case behavior of the function is $O(n\log n)$.

[1] The actual worst-case bound is more complicated and presented in a form that is only meaningful in combination with the source code.

To give an example for which the analysis does not compute a bound, I also implement the sorting algorithm bubble sort and describe why the analysis fails.

Insertion Sort

Below is the implementation of insertion sort in RAML. The same implementation may also be given in a textbook.

```
insert(x,l) = match l with | nil  → [x]
                | y::ys  → if y < x then y::insert(x,ys)
                                    else x::y::ys;

isort l = match l with | nil  → nil
            | x::xs  → insert (x,isort xs);
```

If we instantiate our type system with the evaluation-step metric then the prototype implementation automatically computes the following types.

$$\begin{aligned} \mathit{insert} &: (\mathit{int}, L^{(12,0)}(\mathit{int})) \xrightarrow{5/0} L^{(0,0)}(\mathit{int}) \\ \mathit{isort} &: L^{(12,12)}(\mathit{int}) \xrightarrow{3/0} L^{(0,0)}(\mathit{int}) \end{aligned}$$

The typing express that *insert* needs at most $5+12n$ evaluation steps and *isort* needs at most $3+6n+6n^2$ if n is the size of the respective input list.[2] In the type derivation of *isort* we need resource-polymorphic recursion since the result of the recursive call has to contain potential to pay for the following evaluation of *insert*. The type of the recursive call is $\mathit{isort}{:}L^{(24,12)}(\mathit{int}) \xrightarrow{3/0} L^{(12,0)}(\mathit{int})$.

Quick Sort

Quick sort can also be implemented in RAML in the usual way.

```
append(l,ys) = match l with | nil  → ys
                 | x::xs  → x::append(xs,ys);

split(p,l) = match l with | nil  → (nil,nil)
               | x::xs  → let (ls,rs) = split (p,xs) in
                          if x > p then (ls,x::rs) else (x::ls,rs);

quicksort l = match l with | nil  → nil
            | (x::xs)  → let (ls,rs) = split (x,xs) in
                         append(quicksort ls, x::(quicksort rs));
```

With the evaluation-step metric, the prototype infers the following types.

$$\begin{aligned} \mathit{append} &: (L^{(8,0)}(\mathit{int}), L^{(0,0)}(\mathit{int})) \xrightarrow{0,0} L^{(0,0)}(\mathit{int}) \\ \mathit{split} &: L^{(50,24)}(\mathit{int}) \xrightarrow{5,0} (L^{(34,24)}(\mathit{int}), L^{(26,24)}(\mathit{int})) \\ \mathit{quicksort} &: L^{(26,24)}(\mathit{int}) \xrightarrow{3,0} L^{(0,0)}(\mathit{int}) \end{aligned}$$

[2] Note that these symbolic bounds are also part of the output of the analysis in our prototype implementation.

Thus *quicksort* uses at most $3+14n+12n^2$ evaluation steps. The function is typed resource-monomorphically in the first recursive call *quicksort rs* and resource-polymorphically in the second recursive call *quicksort ls*. The typing

$$\mathit{quicksort} \quad : \quad L^{(34,24)}(\mathit{int}) \xrightarrow{3,0} L^{(8)}(\mathit{int})$$

is used there to cover the cost of *append*.

As the computed bounds indicate, insertion sort indeed admits a better worst-case behavior than quick sort. The reason is that there is an (expensive) call to *append* at each recursive call to *quicksort*. Below is a tail-recursive version of quick sort that does not use *append*.

```
q_aux(l,acc) = match l with | nil  → acc
               | x::xs  → let (ls,rs) = split (x,xs) in
                          let acc' = x::q_aux(rs,acc)
                          in q_aux(ls,acc');
quicksort2 l = q_aux(l,[]);
```

The prototype infers the following types.

$$\begin{aligned} \mathit{q\_aux} \quad &: \quad (L^{(26,16)}(\mathit{int}), L^{(0,0)}(\mathit{int})) \xrightarrow{3,0} L^{(0,0)}(\mathit{int}) \\ \mathit{quicksort2} \quad &: \quad L^{(26,16)}(\mathit{int}) \xrightarrow{7,0} L^{(0,0)}(\mathit{int}) \end{aligned}$$

The bound for *quicksort2* is $7+18n+8n^2$. It improves the bound of *quicksort* in the quadratic part. The reduced potential in the second position of the type annotation of the argument corresponds directly to the costs for the calls of *append*. However, insertion sort has still a slightly better bound. Since *quicksort2* is tail recursive, there is no need to use resource-polymorphic recursion in the type derivation.

Selection Sort

Selection sort is implemented as follows.

```
findmin l = match l with | nil  → nil
              | x::xs  → match findmin xs with
                           | nil  → [x]
                           | y::ys  → if x < y then x::y::ys
                                      else y::x::ys;

selsort l = match findMin l with | nil  → nil
             | x::xs  → x::selsort(xs);
```

If we instantiate our type system with the evaluation-step metric then the prototype implementation automatically computes the following types.

$$\begin{aligned} \mathit{findmin} \quad &: \quad L^{(14,0)}(\mathit{int}) \xrightarrow{3/0} L^{(0,0)}(\mathit{int}) \\ \mathit{selsort} \quad &: \quad L^{(24,14)}(\mathit{int}) \xrightarrow{7/0} L^{(0,0)}(\mathit{int}) \end{aligned}$$

The typing express that *findmin* needs at most $3+14n$ and *selsort* needs at most $7+24n+14n^2$ evaluation steps if n is the size of the respective input list.

Merge Sort

The next sorting algorithm I implement is merge sort.

```
msplit l = match l with | nil → (nil,nil)
            | x1::xs → match xs with | nil → ([x1],nil)
                        | x2::xs' → let (l1,l2) = msplit xs' in
                                    (x1::l1,x2::l2);

merge (l1,l2) = match l1 with | nil → l2
                | x::xs → match l2 with | nil → (x::xs)
                          | y::ys → if x<y
                                    then x::merge(xs,y::ys)
                                    else y::merge(x::xs,ys);

msort l = match l with | nil → nil
          | x1::xs → match xs with | nil → l
                      | x2::xs' → let (l1,l2) = msplit l in
                                  merge (msort l1, msort l2);
```

The following types are computed with the evaluation-step metric.

$$\begin{aligned} \mathit{msplit} &: L^{(23,46)}(\mathit{int}) \xrightarrow{25,0} (L^{(16,92)}(\mathit{int}), L^{(16,92)}(\mathit{int})) \\ \mathit{merge} &: (L^{(16,0)}(\mathit{int}), L^{(16,0)}(\mathit{int})) \xrightarrow{3,0} L^{(0,0)}(\mathit{int}) \\ \mathit{msort} &: L^{(0,92)}(\mathit{int}) \xrightarrow{5,0} L^{(0,0)}(\mathit{int}) \end{aligned}$$

The evaluation-step bound for *msort* is $5-46n+46n^2$. In both recursive calls, the function is used resource-polymorphically with the alternate typing

$$\mathit{msort} \;:\; L^{(16,92)}(\mathit{int}) \xrightarrow{5,0} L^{(16,0)}(\mathit{int})\,.$$

Although our system cannot express an asymptotically tight $O(n \log n)$ bound for the function, it doubles the quadratic potential in the result of *msplit* and thus implicitly infers that *msplit* divides a list into two sublists of about equal length.

Bubble Sort

Finally, I implement bubble sort in RAML as follows.

```
bubble l = match l with
    | nil → (nil,false)
    | x1::xs → match xs with
       | nil → (l,false)
       | x2::xs' → if x1 > x2 then
          let (ys,flag) = bubble (x1::xs') in (x2::ys,true)
         else
          let (ys,flag) = bubble (x2::xs') in (x1::ys,flag);

bubblesort l = let (l',flag) = bubble l in
               if flag then bubblesort l' else l';
```

With the evaluation-step metric, the prototype computes the following typing for the function *bubble*. It states that the evaluation of the expression *bubble(l,n)* needs at most $5+18|\ell|$ evaluation steps

$$bubble \quad : \quad L^{(18,0)}(int) \xrightarrow{5,0} (L^{(0,0)}(int), bool)$$

However, the prototype terminates without providing a bound for *bubblesort*. To prove a quadratic evaluation-step bound, one would have to argue that the value *flag* returned by *bubble* is *true* after at most $n-1$ recursive calls of *bubblesort*, where n is the length of the input list.

The example is representative for a class of functions whose worst-case resource bounds can only be proved by using domain-specific knowledge. Admittedly, our type-based approach allows for the manual annotation of such functions using types that represent bounds determined by a user. So you can still profit from the automatic inference of the bound for *bubble* and the manually derived bound for *bubblesort* can be used to infer bounds for a larger program. However, it would be beneficial to develop a program logic to prove the correctness of such user-annotated types.

5.5.3 Transitive Closure

The function *trans* that is defined below is an example that uses potential of a tree. For a binary tree t the expression *trans(t,[])* evaluates to a list ℓ such that (x,y) is in ℓ if and only if x is an ancestor of y in t. In other words *trans(t,[])* computes the transitive closure of t.

```
attach : (int,T(int),L(int,int)) → L(int,int)

attach(y,t,acc) = match t with | leaf → acc
   | node(x,t1,t2) → let acc1 = attach(y,t1,acc) in
                     let acc2 = attach(y,t2,acc1) in (y,x)::acc2;

trans : (T(int),L(int,int)) → L(int,int)

trans(t,acc) = match t with | leaf → acc
    | node(x,t1,t2) → let acc1 = attach(x,t1,acc) in
                      let acc2 = attach(x,t2,acc1) in
                      let acc3 = trans(t1,acc2) in
                      trans(t2,acc3);
```

The following types are inferred with the heap-space metric.

$$\begin{aligned} attach \quad &: \quad (int, T^{(3,0)}(int), L^{(0,0)}(int,int)) \xrightarrow{0/0} L^{(0,0)}(int,int) \\ trans \quad &: \quad (T^{(0,3)}(int), L^{(0,0)}(int,int)) \xrightarrow{0/0} L^{(0,0)}(int,int) \end{aligned}$$

According to Lemma 5.1.4, the bound that is implied by the type of *trans* is

$$\sum_{i=2}^{h} n_i \cdot 3 \cdot (i-1)$$

where n_i is the number of nodes on level i. If the input tree is balanced then

$$\begin{aligned}\sum_{i=2}^{h} n_i \cdot 3 \cdot (i-1) &= \sum_{i=2}^{\lfloor log_2 n \rfloor} 2^i \cdot 3 \cdot (i-1) \\ &\leq 3 \cdot (\lfloor log_2 n \rfloor - 1) \cdot \sum_{i=2}^{\lfloor log_2 n \rfloor} 2^i \\ &\leq 3 \cdot (\lfloor log_2 n \rfloor - 1) \cdot (n-1)\end{aligned}$$

Note that $3\binom{n}{2} = 1.5n^2 - 1.5n$ is an upper bound for this function.

Mathematical analysis is as extensive as nature itself; it defines all perceptible relations, measures times, spaces, forces, temperatures; this difficult science is formed slowly, but it preserves every principle which it has once acquired; it grows and strengthens itself incessantly in the midst of the many variations and errors of the human mind.

JOSEPH FOURIER
The Analytical Theory of Heat (1878)

6 Multivariate Polynomial Potential

The univariate polynomial amortized analysis that I presented in Chapter 5 extends linear automatic amortized analysis to polynomial bounds while preserving most of the features that make the linear system practicable. However, the inability of the univariate system to express mixed multiplicative bounds such as $n \cdot m$ hampers both utilization in practice and compositionality.

In this chapter, I describe a *multivariate polynomial amortized resource analysis* that extends the univariate system. It preserves the principles of the univariate system while expanding the set of potential functions so as to express a wide range of dependencies between different data structures. The presentation is based on an article that appeared at the 38th ACM Symposium on Principles of Programming Languages (POPL'11) [HAH11].

Section 6.1 introduces *resource polynomials*, the multivariate potential functions that I use in the chapter. In Section 6.2, I show how data types can be annotated with resource polynomials. In contrast to the previous chapters, there is one global resource polynomial for tuple types. Section 6.3 contains multivariate shift operations as well as type rules that are used to derive annotated type judgments. In Section 6.4, I prove the soundness of the multivariate analysis. Section 6.5 explains the type inference and Section 6.6 demonstrates the analysis with several example programs.

6.1 Resource Polynomials

A resource polynomial maps a value of some data type to a nonnegative rational number. Potential functions in this section are always given by such resource polynomials.

In the case of an inductive tree-like data type, a resource polynomial will only depend on the list of entries of the data structure in pre-order. Thus, if $D(A)$ is such a data type with entries of type A, that is, A-labelled binary trees, and v is a value of type

$D(A)$ then we write $\text{elems}(v) = [a_1, \ldots, a_n]$ for this list of entries.

An analysis of typical polynomial computations operating on a data structure v with $\text{elems}(v) = [a_1, \ldots, a_n]$ shows that it consists of operations that are executed for every k-tuple $(a_{i_1}, \ldots, a_{i_k})$ with $1 \le i_1 < \cdots < i_k \le n$. The simplest examples are linear map operations that perform some operation for every a_i. Another example are common sorting algorithms that perform comparisons for every pair (a_i, a_j) with $1 \le i < j \le n$ in the worst case.

Base Polynomials

For each data type A, I now define a set $P(A)$ of functions $p : [\![A]\!] \to \mathbb{N}$ that map values of type A to natural numbers. The resource polynomials for type A are then given as nonnegative rational linear combinations of these *base polynomials*. We define $P(A)$ as follows.

$$\begin{aligned} P(A) &= \{a \mapsto 1\} \text{ if } A \text{ is an atomic type} \\ P(A_1, A_2) &= \{(a_1, a_2) \mapsto p_1(a_1) \cdot p_2(a_2) \mid p_i \in P(A_i)\} \\ P(D(A)) &= \Big\{v \mapsto \sum_{1 \le j_1 < \cdots < j_k \le n} \prod_{1 \le i \le k} p_i(a_{j_i}) \mid k \in \mathbb{N}, p_i \in P(A)\Big\} \end{aligned}$$

In the last clause we have $[a_1, \ldots, a_n] = \text{elems}(v)$. Every set $P(A)$ contains the constant function $v \mapsto 1$. In the case of $D(A)$ this arises for $k = 0$ (one element sum, empty product).

For example, the function $\ell \mapsto \binom{|\ell|}{k}$ is in $P(L(A))$ for every $k \in \mathbb{N}$; simply take $p_1 = \ldots = p_k = 1$ in the definition of $P(D(A))$. The function $(\ell_1, \ell_2) \mapsto \binom{|\ell_1|}{k_1} \cdot \binom{|\ell_2|}{k_2}$ is in $P(L(A), L(B))$ for every $k_1, k_2 \in \mathbb{N}$, and the function $[\ell_1, \ldots, \ell_n] \mapsto \sum_{1 \le i < j \le n} \binom{|\ell_i|}{k_1} \cdot \binom{|\ell_j|}{k_2}$ is in $P(L(L(A)))$ for every $k_1, k_2 \in \mathbb{N}$.

Resource Polynomials

A *resource polynomial* $p : [\![A]\!] \to \mathbb{Q}_0^+$ for a data type A is a non-negative linear combination of base polynomials, that is,

$$p = \sum_{i=1,\ldots,m} q_i \cdot p_i$$

for $q_i \in \mathbb{Q}_0^+$ and $p_i \in P(A)$. We write $R(A)$ for the set of resource polynomials for A.

An instructive, but not exhaustive, example is given by $R_n = R(L(\mathit{int}), \ldots, L(\mathit{int}))$. The set R_n is the set of linear combinations of products of binomial coefficients over variables $x_1, \ldots, x_n$, that is, $R_n = \{\sum_{i=1}^m q_i \prod_{j=1}^n \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$. These expressions naturally generalize the univariate polynomials from Chapter 5 and meet two conditions that are important to efficiently manipulate polynomials during the analysis. Firstly, the polynomials are non-negative, and secondly, they are closed under the discrete

difference operators Δ_i for every i. The discrete derivative $\Delta_i\, p$ is defined through $\Delta_i\, p(x_1,\ldots,x_n) = p(x_1,\ldots,x_i+1,\ldots,x_n) - p(x_1,\ldots,x_n)$.

As in [HH10b] it can be shown that R_n is the largest set of polynomials enjoying these closure properties. It would be interesting to have a similar characterisation of $R(A)$ for arbitrary A. So far, we know that $R(A)$ is closed under sum and product (see Lemma 6.2.1) and are compatible with the construction of elements of data structures in a very natural way (see Lemmas 6.2.3 and 6.2.4). This provides some justification for their choice and canonicity. An abstract characterization would have to take into account the fact that our resource polynomials depend on an unbounded number of variables, e.g., sizes of inner data structures, and are not invariant under permutation of these variables. It seems that some generalization of infinite symmetric polynomials to subgroups of the symmetric group could be useful, but this would not serve our immediate goal of accurate multivariate resource analysis.

6.2 Annotated Types

The resource polynomials described in Section 6.1 are non-negative linear combinations of base polynomials. The rational coefficients of the linear combination are present as type annotations in our type system. To relate type annotations to resource polynomials we systematically describe base polynomials and resource polynomials for data of a given type.

If one considers only univariate polynomials then their description is straightforward. Every inductive data structure of size n has a potential of the form $\sum_{1\le i\le k} q_i \binom{n}{i}$. So we can describe the potential function with a vector $\vec{q} = (q_1,\ldots,q_k)$ in the corresponding recursive type. For instance we can write $L^{\vec{q}}(A)$ for annotated list types. Since each annotation refers to the size of one input part only, univariatly annotated types can be directly composed. For example, an annotated type for a pair of lists has the form $(L^{\vec{q}}(A), L^{\vec{p}}(A))$. See Chapter 5 for details.

In this chapter, I use multivariate potential functions, that is, functions that depend on the sizes of different parts of the input. For a pair of lists of lengths n and m we have, for instance, a potential function of the form $\sum_{0\le i+j\le k} q_{ij}\binom{n}{i}\binom{m}{j}$, which can be described by the coefficients q_{ij}. But I also would like to describe potential functions that refer to the sizes of different lists inside a list of lists, etc. That is why I need to describe a set of indexes $I(A)$ that enumerate the basic resource polynomials p_i and the corresponding coefficients q_i for a data type A. These type annotations can be, in a straight forward way, automatically transformed into usual easily understood polynomials. This is done in our prototype to present the bounds to the user at the end of the analysis.

Names For Base Polynomials

To assign a unique name to each base polynomial I define the *index set* $I(A)$ to denote resource polynomials for a given data type A. Interestingly, but as I find coincidentally,

$I(A)$ is essentially the meaning of A with every atomic type replaced by *unit*.

$$\begin{aligned} I(A) &= \{*\} \text{ if } A \in \{\mathit{int}, \mathit{bool}, \mathit{unit}\} \\ I(A_1, A_2) &= \{(i_1, i_2) \mid i_1 \in I(A_1) \text{ and } i_2 \in I(A_2)\} \\ I(L(B)) = I(T(B)) &= \{[i_1, \ldots, i_k] \mid k \geq 0, i_j \in I(B)\} \end{aligned}$$

The *degree* $\deg(i)$ of an index $i \in I(A)$ is defined as follows.

$$\begin{aligned} \deg(*) &= 0 \\ \deg(i_1, i_2) &= \deg(i_1) + \deg(i_2) \\ \deg([i_1, \ldots, i_k]) &= k + \deg(i_1) + \cdots + \deg(i_k) \end{aligned}$$

Define $I_k(A) = \{i \in I(A) \mid \deg(i) \leq k\}$. The indexes $i \in I_k(A)$ are an enumeration of the base polyonomials $p_i \in P(A)$ of degree at most k. For each $i \in I(A)$, I define a base polynomial $p_i \in P(A)$ as follows: If $A \in \{\mathit{int}, \mathit{bool}, \mathit{unit}\}$ then

$$p_*(v) = 1\,.$$

If $A = (A_1, A_2)$ is a pair type and $v = (v_1, v_2)$ then

$$p_{(i_1,i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2)\,.$$

If $A = D(B)$ (in our type system D is either lists or binary node-labelled trees) is a data structure and $\mathrm{elems}(v) = [v_1, \ldots, v_n]$ then

$$p_{[i_1,\ldots,i_m]}(v) = \sum_{1 \leq j_1 < \cdots < j_m \leq n} p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m})\,.$$

I use the notation 0_A (or just 0) for the index in $I(A)$ such that $p_{0_A}(a) = 1$ for all a. We have $0_{\mathit{int}} = *$ and $0_{(A_1,A_2)} = (0_{A_1}, 0_{A_2})$ and $0_{D(B)} = []$. If $A = D(B)$ for B a data type then the index $[0, \ldots, 0] \in I(A)$ of length n is denoted by just n. For convenience, I identify the index (i_1, i_2, i_3, i_4) with the index $(i_1, (i_2, (i_3, i_4)))$.

For a list $i = [i_1, \ldots, i_k]$ I write $i_0{::}i$ to denote the list $[i_0, i_1, \ldots, i_k]$. Furthermore, I write $i\,i'$ for the concatenation of two lists i and i'.

Recall that $R(A)$ denotes the set of nonnegative rational linear combinations of the base polynomials.

Lemma 6.2.1 Let $p, p' \in R(A)$ be resource polynomials. Then we have $p + p', p \cdot p' \in R(A)$, $\deg(p + p') = \max(\deg(p), \deg(p'))$, and $\deg(p \cdot p') = \deg(p) + \deg(p')$.

PROOF By linearity it suffices to show this lemma for base polynomials. For them, the claim follows by structural induction. ∎

Corollary 6.2.2 For every $p \in R(A, A)$ there exists $p' \in R(A)$ with $\deg(p') = \deg(p)$ and $p'(a) = p(a, a)$ for all $a \in [\![A]\!]$.

PROOF The proof follows directly from Lemma 6.2.1 noticing that base polynomials $p \in P(A,A)$ take the form $p_i \cdot p_{i'}$. ∎

Lemma 6.2.3 Let $a \in [\![A]\!]$ and $\ell \in [\![L(A)]\!]$ be a list. Let furthermore $k \geq 0$ and let $i_0, \ldots, i_k \in I(A)$ indexes for type A. Then we have

$$\begin{aligned} p_{[i_0,i_1,\ldots,i_k]}([]) &= 0 \\ p_{[i_0,i_1,\ldots,i_k]}(a::\ell) &= p_{i_0}(a) \cdot p_{[i_1,\ldots,i_k]}(\ell) + p_0(a) \cdot p_{[i_0,i_1,\ldots,i_k]}(\ell)\,. \end{aligned}$$

PROOF Let $\ell = [v_1, \ldots, v_n]$. Writing v_0 for a we compute as follows.

$$\begin{aligned} lccl p_{[i_0,i_1,\ldots,i_k]}(a::\ell) &= \sum_{0 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\ &= \sum_{1 \leq j_1 < \cdots < j_m \leq n} p_{i_o}(v_0) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\ &+ \sum_{1 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\ &= p_{i_o}(a) \cdot \sum_{1 \leq j_1 < \cdots < j_m \leq n} p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\ &+ \sum_{1 \leq j_0 < j_1 < \cdots < j_m \leq n} p_{i_o}(v_{j_0}) \cdot p_{i_1}(v_{j_1}) \cdots p_{i_m}(v_{j_m}) \\ &= p_{i_0}(a) \cdot p_{[i_1,\ldots,i_k]}(\ell) + p_0(a) \cdot p_{[i_0,i_1,\ldots,i_k]}(\ell) \end{aligned}$$

The statement $p_{[i_0,i_1,\ldots,i_k]}([]) = 0$ is obvious as the sum in the definition of the corresponding base polynomial is over the empty index set. ∎

Lemma 6.2.4 characterizes concatenations of lists (written as juxtaposition) as they will occur in the construction of tree-like data. Note that we have for instance that $\mathrm{elems}(\mathrm{node}(a, t_1, t_2)) = a::\mathrm{elems}(t_1)\,\mathrm{elems}(t_2)$.

Lemma 6.2.4 Let $\ell_1, \ell_2 \in [\![L(A)]\!]$ be lists of type A. Then we have $\ell_1\ell_2 \in [\![L(A)]\!]$ and $p_{[i_1,\ldots,i_k]}(\ell_1\ell_2) = \sum_{t=0}^{k} p_{[i_1,\ldots,i_t]}(\ell_1) \cdot p_{[i_{t+1},\ldots,i_k]}(\ell_2)$ for all indexes $i_j \in I(A)$.

This can be proved by induction on the length of ℓ_1 using Lemma 6.2.3 or else by a decomposition of the defining sum according to which indices hit the first list and which ones hit the second.

Annotated Types and Potential Functions

I use the indexes and base polynomials to define type annotations and resource polynomials. I then give examples to illustrate the definitions.

A *type annotation* for a data type A is defined to be a family

$$Q_A = (q_i)_{i \in I(A)} \text{ with } q_i \in \mathbb{Q}_0^+\,.$$

I say that Q_A is of *degree (at most) k* if $q_i = 0$ for every $i \in I(A)$ with $deg(i) > k$. An *annotated data type* is a pair (A, Q_A) of a data type A and a type annotation Q_A of some degree k.

Let H be a heap and let v be a value with $H \vDash v \mapsto a : A$ for a data type A. Then the type annotation Q_A defines the *potential*

$$\Phi_H(v{:}(A, Q_A)) = \sum_{i \in I(A)} q_i \cdot p_i(a) \,.$$

Usually, I define type annotations Q_A by only stating the values of the non-zero coefficients q_i. However, it is sometimes handy to write annotations $(q_0, \ldots, q_n)$ for a list of atomic types just as a vector. Similarly, I write annotations $(q_0, q_{(1,0)}, q_{(0,1)}, q_{(1,1)}, \ldots)$ for pairs of lists of atomic types sometimes as a triangular matrix.

If $a \in [\![A]\!]$ and Q is a type annotation for A then I also write $\Phi(a : (A, Q))$ for $\sum_i q_i p_i(a)$.

Examples

The simplest annotated types are those for atomic data types like integers. The indexes for *int* are $I(int) = \{*\}$ and thus each type annotation has the form (int, q_0) for a $q_0 \in \mathbb{Q}_0^+$. It defines the constant potential function $\Phi_H(v{:}(int, q_0)) = q_0$. Similarly, tuples of atomic types feature a single index of the form $(*, \ldots, *)$ and a constant potential function defined by some $q_{(*,\ldots,*)} \in \mathbb{Q}_0^+$.

More interesting examples are lists of atomic types like, that is, $L(int)$. The set of indexes of degree k is then

$$I_k(L(int)) = \{[], [*], [*, *], \ldots, [*, ..., *]\}$$

where the last list contains k unit elements. Since we identify a list of i unit elements with the integer i we have $I_k(L(int)) = \{0, 1, \ldots, k\}$. Consequently, annotated types have the form $(L(int), (q_0, \ldots, q_k))$ for $q_i \in \mathbb{Q}_0^+$. The defined potential function is $\Phi([a_1, \ldots, a_n]{:}(L(int), (q_0, \ldots, q_n))) = \sum_{0 \le i \le k} q_i \binom{n}{i}$.

The next example is the type $(L(int), L(int))$ of pairs of integer lists. The set of indexes of degree k is

$$I_k(L(int), L(int)) = \{(i, j) \mid i + j \le k\}$$

if we identify lists of units with their lengths as usual. Annotated types are then of the form $((L(int), L(int)), Q)$ for a triangular $k \times k$ matrix Q with non-negative rational entries. If $\ell_1 = [a_1, \ldots, a_n]$, $\ell_2 = [b_1, \ldots, b_m]$ are two lists then the potential function is $\Phi((\ell_1, \ell_2), ((L(int), L(int)), (q_{(i,j)}))) = \sum_{0 \le i+j \le k} q_{(i,j)} \binom{n}{i} \binom{m}{j}$.

Finally, consider the type $A = L(L(int))$ of lists of lists of integers. The set of indexes of degree k is then

$$I_k(L(L(int))) = \left\{[i_1, \ldots, i_m] \mid m \le k, i_j \in \mathbb{N}, \sum_{j=1,\ldots,m} i_j \le k - m\right\} .$$

Thus we have $I_k(L(L(int))) = \{0,\ldots,k\} \cup \{[1],\ldots,[k-1]\} \cup \{[0,1],\ldots\} \cup \cdots$. Let for instance $\ell = [[a_{11},\ldots,a_{1m_1}],\ldots,[a_{n1},\ldots,a_{nm_n}]]$ be a list of lists and $Q = (q_i)_{i \in I_k(L(L(int)))}$ be a corresponding type annotation. The defined potential function is then

$$\Phi(\ell,(L(L(int)),Q)) = \sum_{[i_1,\ldots,i_l] \in I_k(A)} \sum_{1 \le j_1 < \cdots < j_l \le n} q_{[i_1,\ldots,i_l]} \binom{m_{j_1}}{i_1} \cdots \binom{m_{j_l}}{i_l}.$$

In practice the potential functions are usually not very complex since most of the q_i are zero. Note that the resource polynomials for binary trees are identical to those for lists.

The Potential of a Context

For use in the type system, I have to extend the definition of resource polynomials to typing contexts. I treat a context like a tuple type.

Let $\Gamma = x_1{:}A_1,\ldots,x_n{:}A_n$ be a typing context and let $k \in \mathbb{N}$. The index set $I_k(\Gamma)$ is defined through

$$I_k(\Gamma) = \left\{(i_1,\ldots,i_n) \mid i_j \in I_{m_j}(A_j), \sum_{j=1,\ldots,n} m_j \le k\right\}.$$

A *type annotation* Q of degree k for Γ is a family

$$Q = (q_i)_{i \in I_k(\Gamma)} \text{ with } q_i \in \mathbb{Q}_0^+ .$$

I denote a *resource-annotated context* with $\Gamma;Q$. Let H be a heap and V be a stack with $H \vDash V : \Gamma$ where $H \vDash V(x_j) \rightarrowtail a_{x_j} : \Gamma(x_j)$. The potential of $\Gamma;Q$ with respect to H and V is

$$\Phi_{V,H}(\Gamma;Q) = \sum_{(i_1,\ldots,i_n) \in I_k(\Gamma)} q_{\vec{i}} \prod_{j=1}^{n} p_{i_j}(a_{x_j})$$

In particular, if $\Gamma = \emptyset$ then $I_k(\Gamma) = \{()\}$ and $\Phi_{V,H}(\Gamma;q_0) = q_0$. I sometimes also write q_0 for $q_{()}$.

6.3 Type Rules

Before I describe the multivariate type system, I formalize some facts about the potential method that are useful to explain some of the ideas I describe later.

If $f : [\![A]\!] \to [\![B]\!]$ is a function computed by some program and $K(a)$ is the cost of the evaluation of $f(a)$ then our type system will essentially try to identify resource polynomials $p \in R(A)$ and $\bar{p} \in R(B)$ such that $p(a) \ge \bar{p}(f(a)) + K(a)$. The key aspect of such amortized cost accounting is that it interacts well with composition.

Proposition 6.3.1 Let $p \in R(A)$, $\check{p} \in R(B)$, and $\bar{p} \in R(C)$ be resource polynomials. Let $f : [\![A]\!] \to [\![B]\!]$, $g : [\![B]\!] \to [\![C]\!]$, $K_1 : [\![A]\!] \to \mathbb{Q}$, and $K_2 : [\![B]\!] \to \mathbb{Q}$. If $p(a) \ge \check{p}(f(a)) + K_1(a)$ and $\check{p}(b) \ge \bar{p}(g(b)) + K_2(b)$ for all a,b then $p(a) \ge \bar{p}(g(f(a))) + K_1(a) + K_2(f(a))$ for all a.

Notice that if we merely had $p(a) \geq K_1(a)$ and $\check{p}(b) \geq K_2(b)$ then no bound could be directly obtained for the composition.

Interaction with parallel composition, that is, $(a,c) \mapsto (f(a),c)$, is more complex than in the univariate system due to the presence of mixed multiplicative terms in the resource polynomials.

Proposition 6.3.2 Let $p \in R(A,C), \bar{p} \in R(B,C)$, $f : [\![A]\!] \to [\![B]\!]$, and $K : [\![A]\!] \to \mathbb{Q}$. For each $j \in I(C)$ let $p^{(j)} \in R(A)$ and $\bar{p}^{(j)} \in R(B)$ be such that $p(a,c) = \sum_j p^{(j)}(a)p_j(c)$ and $\bar{p}(b,c) = \sum_j \bar{p}^{(j)}(b)p_j(c)$. If $p^{(0)}(a) \geq \bar{p}^{(0)}(f(a)) + K(a)$ and $p^{(j)}(a) \geq \bar{p}^{(j)}(f(a))$ holds for all a and $j \neq 0$ then $p(a,c) \geq \bar{p}(f(a),c) + K(a)$.

In fact, the situation is more complicated due to the accounting for high watermarks as opposed to merely additive cost, and also due to the fact that functions are recursively defined and may be partial. Furthermore, we have to deal with contexts and not merely types. To gain an intuition for the development to come, the above simplified view should, however, prove helpful.

Type Judgments

The declarative type rules for RAML expressions (see Figure 6.1 and Figure 6.2) define a *multivariate resource-annotated typing judgment* of the form

$$\Sigma; \Gamma; Q \vdash e : (A, Q')$$

where e is a RAML expression, Σ is a resource-annotated signature (see below), $\Gamma; Q$ is a resource-annotated context and (A, Q') is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $\Phi(\Gamma; Q)$ resource units available then this is sufficient to pay for the cost of the evaluation e. In addition, there are more than $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

Programs with Annotated Types

Multivariate resource-annotated first-order types have the form $(A, Q) \to (B, Q')$ for annotated data types (A, Q) and (B, Q'). A *multivariate resource-annotated signature* Σ is a finite, partial mapping of function identifiers to *sets of* resource-annotated first-order types.

Like in the univariate and linear cases, a RAML program with (multivariate) resource-annotated types consists of a (multivariate) resource-annotated signature Σ and a family of expressions with variables identifiers $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ such that $\Sigma; y_f{:}A; Q \vdash e_f : (B, Q')$ for every function type $(A, Q) \to (B, Q') \in \Sigma(f)$.

Notations

Families that describe type and context annotations are denoted with upper case letters $Q, P, R, \ldots$ with optional superscripts. I use the convention that the elements of the

families are the corresponding lower case letters with corresponding superscripts, that is, $Q = (q_i)_{i \in I}$, $Q' = (q'_i)_{i \in I}$, and $Q^x = (q^x_i)_{i \in I}$.

Let Q, Q' be two annotations with the same index set I. I write $Q \leq Q'$ if $q_i \leq q'_i$ for every $i \in I$. For $K \in \mathbb{Q}$ I write $Q = Q' + K$ to state that $q_{\vec{0}} = q'_{\vec{0}} + K \geq 0$ and $q_i = q'_i$ for $i \neq \vec{0} \in I$. Let $\Gamma = \Gamma_1, \Gamma_2$ be a context, let $i = (i_1, \ldots, i_k) \in I(\Gamma_1)$ and $j = (j_1, \ldots, j_l) \in I(\Gamma_2)$. I write (i, j) to denote the index $(i_1, \ldots, i_k, j_1, \ldots, j_l) \in I(\Gamma)$.

Like in the other systems, I write

$$\Sigma; \Gamma; Q \vdash^{\mathrm{cf}} e : (A, Q')$$

to refer to multivariate cost-free type judgments where all constants K in the rules from Figure 6.1 and Figure 6.2 are zero. I use it to assign potential to an extended context in the let rule. More explanations will follow later.

Let Q be an annotation for a context Γ_1, Γ_2. For $j \in I(\Gamma_2)$, I define the *projection* $\pi_j^{\Gamma_1}(Q)$ of Q to Γ_1 to be the annotation Q' with $q'_i = q_{(i,j)}$. The essential properties of the projections are stated by Propositions 6.3.2 and 6.3.3; they show how the analysis of juxtaposed functions can be broken down to individual components.

Proposition 6.3.3 Let $\Gamma, x{:}A; Q$ be an annotated context. Let furthermore $H \vDash V : \Gamma, x{:}A$ and $H \vDash V(x) \mapsto a : A$. Then it is true that $\Phi_{V,H}(\Gamma, x{:}A; Q) = \sum_{j \in I(A)} \Phi_{V,H}(\Gamma; \pi_j^{\Gamma}(Q)) \cdot p_j(a)$.

Additive Shift

A key notion in the type system is the multivariate *additive shift* that is used to assign potential to typing contexts that result from a pattern match or from an application of a constructor of an inductive data type. I first define the additive shift, then illustrate the definition with examples and finally state and prove the soundness of the operation.

Let $\Gamma, y{:}L(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, y:L(A))}$ be a context annotation of degree k. The *additive shift for lists* $\lhd_L(Q)$ of Q is an annotation $\lhd_L(Q) = (q'_i)_{i \in I(\Gamma, x:A, xs:L(A))}$ of degree k for a context $\Gamma, x{:}A, xs{:}L(A)$ that is defined through

$$q'_{(i,j,\ell)} = \begin{cases} q_{(i,j::\ell)} + q_{(i,\ell)} & j = 0 \\ q_{(i,j::\ell)} & j \neq 0 \end{cases}$$

Let $\Gamma, t{:}T(A)$ be a context and let $Q = (q_i)_{i \in I(\Gamma, t:T(A))}$ be a context annotation of degree k. The *additive shift for binary trees* $\lhd_T(Q)$ of Q is an annotation $\lhd_T(Q) = (q'_i)_{i \in I(\Gamma')}$ of degree k for a context $\Gamma' = \Gamma, x{:}A, xs_1{:}T(A), xs_2{:}T(A)$ that is defined by

$$q'_{(i,j,\ell_1,\ell_2)} = \begin{cases} q_{(i,j::\ell_1\ell_2)} + q_{(i,\ell_1\ell_2)} & j = 0 \\ q_{(i,j::\ell_1\ell_2)} & j \neq 0 \end{cases}$$

The definition of the additive shift is short but substantial. I begin by illustrating its effect in some example cases. Consider for instance a context $\ell{:}L(\mathit{int})$ with a single integer list that features an annotation $(q_0, \ldots, q_k) = (q_{[]}, \ldots, q_{[0,\ldots,0]})$. The shift operation $\lhd_L$ for lists produces an annotation for a context of the form $x{:}\mathit{int}, xs{:}L(\mathit{int})$, namely

$\lhd_L(q_0,\ldots,q_k) = (q_{(0,0)},\ldots,q_{(0,k)})$ such that $q_{(0,i)} = q_i + q_{i+1}$ for all $i < k$ and $q_{(0,k)} = q_k$. This is exactly the additive shift that I used in Chapter 5. Like in the univariate system, we use it in a context where ℓ points to a list of length $n+1$ and xs is the tail of ℓ. It reflects the fact that $\sum_{i=0,\ldots,k} q_i\binom{n+1}{i} = \sum_{i=0,\ldots,k-1} q_{i+1}\binom{n}{i} + \sum_{i=0,\ldots,k} q_i\binom{n}{i}$.

Now consider the annotated context $t{:}T(int);(q_0,\ldots,q_k)$ with a single variable t that points to a tree with $n+1$ nodes. The additive shift $\lhd_T$ produces an annotation for a context of the form $x{:}int, t_1{:}T(int), t_2{:}T(int)$. We have $\lhd_T(q_0,\ldots,q_k) = (q_{(0,i,j)})_{i+j\le k}$ where $q_{(0,i,j)} = q_{i+j} + q_{i+j+1}$ if $i+j<k$ and $q_{(0,i,j)} = q_{i+j}$ if $i+j=k$. The intention is that t_1 and t_2 are the subtrees of t which have n_1 and n_2 nodes, respectively ($n_1 + n_2 = n$). The definition of the additive shift for trees incorporates the convolution $\binom{n+m}{k} = \sum_{i+j=k}\binom{n}{i}\binom{m}{j}$ for binomials. It is true that $\sum_{i=0,\ldots,k} q_i\binom{n+1}{i} = \sum_{i=0,\ldots,k-1}(q_i + q_{i+1})\binom{n}{i} + q_k\binom{n}{k} = \sum_{i=0}^{k-1}\sum_{j_1+j_2=i}(q_i+q_{i+1})\binom{n_1}{j_1}\binom{n_2}{j_2} + \sum_{j_1+j_2=k} q_i\binom{n_1}{j_1}\binom{n_2}{j_2}$.

As a last example consider the context $\ell_1{:}L(int), \ell_2{:}L(int); Q$ where $Q = (q_{(i,j)})_{i+j\le k}$, ℓ_1 is a list of length m, and ℓ_2 is a list of length $n+1$. The additive shift results in an annotation for a context of the form $\ell_1{:}L(int), x{:}int, xs{:}L(int)$ and the intention is that xs is the tail of ℓ_2, that is, a list of length n. From the definition it follows that $\lhd_L(Q) = (q_{(i,0,j)})_{i+j\le k}$ where $q_{(i,0,j)} = q_{(i,j)} + q_{(i,j+1)}$ if $i+j<k$ and $q_{(i,0,j)} = q_{(i,j)}$ if $i+j=k$. The soundness follows from the fact that $\sum_{j=1}^{k-i} q_{(i,j)}\binom{m}{i}\binom{n+1}{j} = \binom{m}{i}\big(\sum_{j=0}^{k-i-1}(q_{(i,j)} + q_{(i,j+1)})\binom{n}{i} + q_{(i,k-i)}\binom{n}{k}\big)$ for every $i \le k$.

Lemmas 6.3.4 and 6.3.5 state the soundness of the shift operations.

Lemma 6.3.4 Let $\Gamma, \ell{:}L(A); Q$ be an annotated context, $H \vDash V : \Gamma, \ell{:}L(A)$, $H(\ell) = (v_1, \ell')$ and let $V' = V[x_h \mapsto v_1, x_t \mapsto \ell']$. Then $H \vDash V' : \Gamma, x_h{:}A, x_t{:}L(A)$ and $\Phi_{V,H}(\Gamma, \ell{:}L(A); Q) = \Phi_{V',H}(\Gamma, x_h{:}A, x_t{:}L(A); \lhd_L(Q))$.

Lemma 6.3.4 is a consequence of Lemma 6.2.3. One takes the linear combination of instances of its second equation and regroups the right hand side according to the base polynomials for the resulting context.

PROOF It follows directly from the assumptions that $H \vDash V' : \Gamma, x_h{:}A, x_t{:}L(A)$. Let $\ell = [v_1,\ldots,v_n]$ and let $q_i \in \mathbb{Q}_0^+$ for each $i \in I(L(A))$. Then $\sum_{[i_1,\ldots,i_k]\in I(L(A))} q_{[i_1,\ldots,i_k]} \cdot p^H_{[i_1,\ldots,i_k]}(\ell)$

$$
\begin{aligned}
&= \sum_{[i_1,\ldots,i_k]} q_{[i_1,\ldots,i_k]} \cdot \Big(\sum_{1\le j_1<\cdots<j_k\le n} p^H_{i_1}(v_{j_1})\cdots p^H_{i_k}(v_{j_k})\Big)\\
&= \sum_{[i_1,\ldots,i_k]} q_{[i_1,\ldots,i_k]} \Big(\sum_{2\le j_1<\cdots<j_k\le n} p^H_{i_1}(v_{j_1})\cdots p^H_{i_k}(v_{j_k})\\
&\quad + q_{[i_1,\ldots,i_k]} \cdot p^H_{i_1}(v_1)\Big(\sum_{2\le j_2<\cdots<j_k\le n} p^H_{i_2}(v_{j_2})\cdots p^H_{i_k}(v_{j_k})\Big)\Big)\\
&= \sum_{[i_1,\ldots,i_k]} \Big(q_{[i_1,\ldots,i_k]} \cdot p^H_{(0,[i_1,\ldots,i_k])}(v_1,\ell') + q_{[i_1,\ldots,i_k]} \cdot p^H_{(i_1,[i_2,\ldots,i_k])}(v_1,\ell')\Big)\\
&= \sum_{[0,i_2,\ldots,i_k]} (q_{[i_2,\ldots,i_k]} + q_{[0,i_2,\ldots,i_k]}) \cdot p^H_{(0,[i_2,\ldots,i_k])}(v_1,\ell') + \sum_{[i_1,\ldots,i_k], i_1\neq 0} q_{[i_1,\ldots,i_k]} \cdot p^H_{(i_1,[i_2,\ldots,i_k])}(v_1,\ell')
\end{aligned}
$$

Let $\Gamma = x_1{:}A_1, \ldots, x_m{:}A_m$, $j_i \in I(A_i)$ and $j = (j_1, \ldots, j_m)$. We write $p^H_j(V(\Gamma))$ instead of $\prod_{i=1}^m p^H_{j_i}(V(x_i))$. Let $\Gamma' = \Gamma, x_h{:}A, x_t{:}L(A)$. From the above equation it follows that

$$\begin{aligned}
\Phi_{V,H}(\Gamma, \ell{:}(A); Q) &= \sum_{(j,i)\in I(\Gamma, L(A))} q_{(j,i)} \cdot p^H_j(V(\Gamma)) \cdot p^H_i(\ell) \\
&= \sum_{j\in I(\Gamma)} p^H_j(V(\Gamma)) \cdot \Big(\sum_{i\in I(L(A))} q_{(j,i)} \cdot p^H_i(\ell) \Big) \\
&= \sum_{j\in I(\Gamma)} p^H_j(V(\Gamma)) \Big(\sum_{[i_1,\ldots,i_k], i_1\neq 0} q_{(j,[i_1,\ldots,i_k])} \cdot p^H_{(i_1,[i_2,\ldots,i_k])}(v_1, \ell') \\
&\quad + \sum_{[0,i_2,\ldots,i_k]} (q_{(j,[i_2,\ldots,i_k])} + q_{(j,[0,i_2,\ldots,i_k])}) \cdot p^H_{(0,[i_2,\ldots,i_k])}(v_1, \ell') \Big) \\
&= \sum_{(j,i_1,[i_2,\ldots,i_k]), i_1\neq 0} q_{(j,[i_1,\ldots,i_k])} \cdot p^H_{(j,i_1,[i_2,\ldots,i_k])}(V'(\Gamma')) \\
&\quad + \sum_{(j,0,[i_2,\ldots,i_k])} (q_{(j,[i_2,\ldots,i_k])} + q_{(j,[0,i_2,\ldots,i_k])}) p^H_{(j,0,[i_2,\ldots,i_k])}(V'(\Gamma')) \\
&= \Phi_{V',H}(\Gamma'; \lhd_L(Q))
\end{aligned}$$

■

Lemma 6.3.5 Let $\Gamma, t{:}T(A); Q$ be an annotated context, let $H \vDash V : \Gamma, t{:}T(A)$, $H(t) = (v_1, t_1, t_2)$, and $V' = V[x_0 \mapsto v_1, x_1 \mapsto t_1, x_2 \mapsto t_2]$. If $\Gamma' = \Gamma, x{:}A, x_1{:}T(A), x_2{:}T(A)$ then $H \vDash V' : \Gamma'$ and $\Phi_{V,H}(\Gamma, t{:}T(A); Q) = \Phi_{V',H}(\Gamma'; \lhd_T(Q))$.

PROOF Remember that the potential of a tree only depends on the list of nodes in pre-order. So, we can think of the context splitting as done in two steps. First the head is separated, as in Lemma 6.3.4, and then the list of remaining elements into two lists. Lemma 6.3.5 is then proved like the previous one by regrouping terms using Lemma 6.2.3 for the first separation and Lemma 6.2.4 for the second one. ■

Sharing

Let $\Gamma, x_1{:}A, x_2{:}A; Q$ be an annotated context. The *sharing operation* $\curlyvee Q$ defines an annotation for a context of the form $\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable. The following lemma shows that sharing is a linear operation that does not lead to any loss of potential.

Lemma 6.3.6 Let A be a data type. Then there are non-negative rational numbers $c^{(i,j)}_k$ for $i, j, k \in I(A)$ and $\deg(k) \leq \deg(i, j)$ such that the following holds. For every context $\Gamma, x_1{:}A, x_2{:}A; Q$ and every H, V with $H \vDash V : \Gamma, x{:}A$ it holds that $\Phi_{V,H}(\Gamma, x{:}A; Q') = \Phi_{V',H}(\Gamma, x_1{:}A, x_2{:}A; Q)$ where $V' = V[x_1, x_2 \mapsto V(x)]$ and $q'_{(\ell,k)} = \sum_{i,j\in I(A)} c^{(i,j)}_k q_{(\ell,i,j)}$.

Lemma 6.3.6 is a direct consequence of Corollary 6.2.2. In fact, inspection of the argument of the underlying Lemma 6.2.1 shows that the coefficients $c^{(i,j)}_k$, are indeed *natural* numbers and can be computed effectively.

For a context $\Gamma, x_1{:}A, x_2{:}A; Q$ we define $\curlyvee Q$ to be the Q' from Lemma 6.3.6.

PROOF The task is to show that for every resource polynomial $p_{(i,j)}((v,v)) = p_i(v) \cdot p_i(v)$ can be written as a sum (possibly with repetitions) of $p_{i'}(v)$'s. We argue by induction on A. If A is an atomic type *bool*, *int*, or *unit*, we can simply write $1 \cdot 1$ as 1. If A is a pair $A = (B, C)$ then we have $p_{(i,j)}((v,w)) \cdot p_{(i',j')}((v,w)) = p_i(v)p_j(w)p_{i'}(v)p_{j'}(w) = (p_i(v)p_{i'}(v))(p_j(w)p_{j'}(w))$. By induction hypothesis, $(p_i(v)p_{i'}(v))$ and $(p_j(w)p_{j'}(w))$ both are sums of elementary resource polynomials for B or C, respectively. So the expression is a sum of terms of the form $p_{i''}(v)p_{j''}(w)$, which is $p_{(i'',j'')}((v,w))$. If A is a list $A = L(B)$ we have to consider

$$\begin{aligned} & p_{[i_1,\ldots,i_k]}([v_1,\ldots,v_n]) \cdot p_{[i'_1,\ldots,i'_{k'}]}([v_1,\ldots,v_n]) \\ = \; & \Big(\sum_{1\le j_1<\ldots<j_k\le n} p_{i_1}(v_{j_1})\ldots p_{i_k}(v_{j_k})\Big)\Big(\sum_{1\le j'_1<\ldots<j'_{k'}\le n} p_{i'_1}(v_{j'_1})\ldots p_{i'_{k'}}(v_{j'_{k'}})\Big) \end{aligned}$$

Using the distributive law, this can be considered as the sum over all possible ways to arrange the $j_1,\ldots,j_k$ and $j'_1,\ldots,j'_{k'}$ relative to each other respecting their respective orders, including the case that some j_i coincide with some $j'_{i'}$. Each of term in this sum of fixed length (independent of the lists) has the shape

$$\sum_{1\le j''_1<\ldots<j''_\ell\le n} q_1(v_{j''_1})\ldots q_\ell(v_{j''_\ell})$$

where each $q_r(v_{j_r})$ is either a $p_{i_s}(v_{j_r})$, a $p_{i'_{s'}}(v_{j_r})$ or a product $p_{i_r}(v_{j_r})p_{i'_{s'}}(v_{j_r})$. The latter can, by induction hypothesis, be written as sum of $p_{i''}(v_{j_r})$'s. Again, this presentation is independent of the actual value of v_{j_r}. Using distributivity again, we obtain a sum of expressions of the form

$$\sum_{1\le j''_1<\ldots<j''_\ell\le n} p_{i''_1}(v_{j''_1})\ldots p_{i''_\ell}(v_{j''_\ell}) = p_{[i''_1,\ldots,i''_\ell]}$$

The case of A being a tree $A = T(B)$ is reduced to the case of A being a list, as the potential for trees is defined to be that of a list—the preorder traversal of the tree. ∎

Type Rules

Figures 6.1 and 6.2 shows the annotated type rules for RAML expressions. I assume a fixed global signature Σ that I omit from the rules. The last four rules are structural rules that apply to every expression. The other rules are syntax-driven and there is one rule for every construct of the syntax. In the implementation we incorporated the structural rules in the syntax-driven ones. The most interesting rules are explained below.

M:SHARE has to be applied to expressions that contain a variable twice (z in the rule). The sharing operation $\curlyvee P$ transfers the annotation P for the context $\Gamma, x{:}A, y{:}A$ into an annotation Q for the context $\Gamma, z{:}A$ without loss of potential (Lemma 6.3.6). This

$$\frac{Q = Q' + K^{\text{var}}}{x{:}B; Q \vdash x : (B, Q')}\ (\text{M:VAR}) \qquad \frac{b \in \{True, False\} \qquad q_0 = q_0' + K^{\text{bool}}}{\emptyset; Q \vdash b : (bool, Q')}\ (\text{M:CONSTB})$$

$$\frac{n \in \mathbb{Z} \qquad q_0 = q_0' + K^{\text{int}}}{\emptyset; Q \vdash n : (int, Q')}\ (\text{M:CONSTI}) \qquad \frac{q_0 = q_0' + K^{\text{unit}}}{\emptyset; Q \vdash () : (unit, Q')}\ (\text{M:CONSTU})$$

$$\frac{op \in \{or, and\} \qquad q_{(0,0)} = q_0' + K^{op}}{x_1{:}bool, x_2{:}bool; Q \vdash x_1\ op\ x_2 : (bool, Q')}\ (\text{M:OPBOOL})$$

$$\frac{op \in \{+, -, *, mod, div\} \qquad q_{(0,0)} = q_0' + K^{op}}{x_1{:}int, x_2{:}int; Q \vdash x_1\ op\ x_2 : (int, Q')}\ (\text{M:OPINT})$$

$$\frac{P + K_1^{\text{app}} = Q \qquad P' = Q' + K_2^{\text{app}} \qquad (A, P) \to (B, P') \in \Sigma(f)}{x{:}A; Q \vdash f(x) : (B, Q')}\ (\text{M:APP})$$

$$\frac{\begin{array}{c}\Gamma_1; P \vdash e_1 : (A, P') \qquad \Gamma_2, x{:}A; R \vdash e_2 : (B, R') \\ P + K_1^{\text{let}} = \pi_{\vec{0}}^{\Gamma_1}(Q) \qquad P' = \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}} \qquad R' = Q' + K_3^{\text{let}} \\ \forall \vec{0} \neq j \in I(\Gamma_2):\ \ \Gamma_1; P_j \overset{\text{cf}}{\vdash} e_1 : (A, P_j') \qquad P_j = \pi_j^{\Gamma_1}(Q) \qquad P_j' = \pi_j^{x:A}(R)\end{array}}{\Gamma_1, \Gamma_2; Q \vdash let\ x = e_1\ in\ e_2 : (B, Q')}\ (\text{M:LET})$$

$$\frac{\begin{array}{lll}\Gamma; P \vdash e_t : (B, P') & P + K_1^{\text{conT}} = \pi_0^{\Gamma}(Q) & P' = Q' + K_2^{\text{conT}} \\ \Gamma; R \vdash e_f : (B, R') & R + K_1^{\text{conF}} = \pi_0^{\Gamma}(Q) & R' = Q' + K_2^{\text{conF}}\end{array}}{\Gamma, x{:}bool; Q \vdash if\ x\ then\ e_t\ else\ e_f : (B, Q')}\ (\text{M:COND})$$

$$\frac{\begin{array}{c}A = (A_1, A_2) \\ \Gamma, x_1{:}A_1, x_2{:}A_2; P \vdash e : (B, P') \qquad P + K_1^{\text{matP}} = Q \qquad P' = Q' + K_2^{\text{matP}}\end{array}}{\Gamma, x{:}A; Q \vdash match\ x\ with\ (x_1, x_2) \to e : (B, Q')}\ (\text{M:MATP})$$

$$\frac{Q = Q' + K^{\text{pair}}}{x_1{:}A_1, x_2{:}A_2; Q \vdash (x_1, x_2) : ((A_1, A_2), Q')}\ (\text{M:PAIR}) \qquad \frac{q_0 = q_{\vec{0}}' + K^{\text{nil}}}{\emptyset; Q \vdash nil : (L(A), Q')}\ (\text{M:NIL})$$

$$\frac{q_0 = q_{\vec{0}}' + K^{\text{leaf}}}{\emptyset; Q \vdash leaf : (T(A), Q')}\ (\text{M:LEAF}) \qquad \frac{Q = \lhd_L(Q') + K^{\text{cons}}}{x_h{:}A, x_t{:}L(A); Q \vdash cons(x_h, x_t) : (L(A), Q')}\ (\text{M:CONS})$$

Figure 6.1: Type rules for annotated types (1 of 2).

$$\frac{Q = \lhd_T(Q') + K^{\text{node}}}{x_0{:}A, x_1{:}T(A), x_2{:}T(A); Q \vdash \mathit{node}(x_0, x_1, x_2) : (T(A), Q')}\ (\text{M:NODE})$$

$$\frac{\begin{array}{c}\Gamma; R \vdash e_1 : (B, R') \qquad R + K_1^{\text{matN}} = \pi_0^\Gamma(Q) \qquad R' = Q' + K_2^{\text{matN}} \\ \Gamma, x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P') \qquad P + K_1^{\text{matC}} = \lhd_L(Q) \qquad P' = Q' + K_2^{\text{matC}}\end{array}}{\Gamma, x{:}L(A); Q \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : (B, Q')}\ (\text{M:MATL})$$

$$\frac{\begin{array}{c}\Gamma; R \vdash e_1 : (B, R') \\ \Gamma, x_0{:}A, x_1{:}T(A), x_2{:}T(A); P \vdash e_2 : (B, P') \qquad R + K_1^{\text{matTL}} = \pi_0^\Gamma(Q) \\ R' = Q' + K_2^{\text{matTL}} \qquad P + K_1^{\text{matTN}} = \lhd_T(Q) \qquad P' = Q' + K_2^{\text{matTN}}\end{array}}{\Gamma, x{:}T(A); Q \vdash \mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 : (B, Q')}\ (\text{M:MATT})$$

$$\frac{\Gamma, x{:}A, y{:}A; P \vdash e : (B, Q') \qquad Q = \curlyvee P}{\Gamma, z{:}A; Q \vdash e[z/x, z/y] : (B, Q')}\ (\text{M:SHARE}) \qquad \frac{\Gamma; \pi_{\vec{0}}^\Gamma(Q) \vdash e : (B, Q')}{\Gamma, x{:}A; Q \vdash e : (B, Q')}\ (\text{M:AUGMENT})$$

$$\frac{\Gamma; P \vdash e : (B, P') \qquad Q \geq P \qquad Q' \leq P'}{\Gamma; Q \vdash e : (B, Q')}\ (\text{M:WEAKEN})$$

$$\frac{\Gamma; P \vdash e : (B, P') \qquad Q = P + c \qquad Q' = P' + c}{\Gamma; Q \vdash e : (B, Q')}\ (\text{M:OFFSET})$$

Figure 6.2: Type rules for annotated types (2 of 2).

is crucial for the accuracy of the analysis since instances of M:SHARE are quite frequent in typical examples. The remaining rules are affine linear in the sense that they assume that every variable occurs at most once.

M:CONS assigns potential to a lengthened list. The additive shift $\lhd_L(Q')$ transforms the annotation Q' for a list type into an annotation for the context $x_h{:}A, x_t{:}L(A)$. Lemma 6.3.4 shows that potential is neither gained nor lost by this operation. The potential Q of the context has to pay for both the potential Q' of the resulting list and the resource cost K^{cons} for list cons.

M:MATL shows how to treat pattern matching of lists. The initial potential defined by the annotation Q of the context $\Gamma, x{:}L(A)$ has to be sufficient to pay the costs of the evaluation of e_1 or e_2 (depending on whether the matched list is empty or not) and the potential defined by the annotation Q' of the result type. To type the expression e_1 of the *nil* case we use the projection $\pi_0^\Gamma(Q)$ that results in an annotation for the context Γ. Since the matched list is empty in this case no potential is lost by the discount of the annotations $q_{(i,j)}$ of Q where $j \neq 0$. To type the expression e_2 of the *cons* case we rely on the shift operation $\lhd_L(Q)$ for lists that results in an annotation for the context $\Gamma, x_h{:}A, x_t{:}L(A)$. Again there is no loss of potential (see Lemma 6.3.4). The equalities relate the potential before and after the evaluation of e_1 or e_2, to the potential before the and after the evaluation of the match operation by incorporating the respective resource cost for the matching.

M:NODE and M:MATT are similar to the corresponding rules for lists but use the shift operator $\lhd_T$ for trees (see Lemma 6.3.5).

M:LET comprises essentially an application of Proposition 6.3.2 (with $f = e_1$ and $C = \Gamma_2$) followed by an application of Proposition 6.3.1 (with f being the parallel composition of e_1 and the identity on Γ_2 and g being e_2). Of course, the rigorous soundness proof takes into account partiality and additional constant costs for dispatching a let. It is part of the inductive soundness proof for the entire type system (Theorem 6.4.1).

The derivation of the type judgment $\Gamma_1, \Gamma_2; Q \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : (B, Q')$ can be explained in two steps. The first starts with the derivation of the judgment $\Gamma_1; P \vdash e_1 : (A, P')$ for the sub-expression e_1. The annotation P corresponds to the potential that is exclusively attached to Γ_1 by the annotation Q plus some resource cost for the *let*, namely $P = \pi_{\vec{0}}^{\Gamma_1}(Q) + K_1^{\text{let}}$. Now we derive the judgment $\Gamma_2, x{:}A; R \vdash e_2 : (B, R')$. The potential that is assigned by R to $x{:}A$ is the potential that resulted from the judgment for e_1 plus some cost that might occur when binding the variable x to the value of e_1, namely $P' = \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}}$. The potential that is assigned by R to Γ_2 is essentially the potential that is assigned by to Γ_2 by Q, namely $\pi_{\vec{0}}^{\Gamma 2}(Q) = \pi_0^{\Gamma_2}(R)$. The second step of the derivation is to relate the annotations in R that refer to mixed potential between $x{:}A$ and Γ_2 to the annotations in Q that refer to potential that is mixed between Γ_1 and Γ_2. To this end we remember that we can derive from a judgment $\Gamma_1; S \vdash e_1 : (A, S')$ that $\Phi(\Gamma_1; S) \geq \Phi(v{:}(A, S'))$ if e_1 evaluates to v. This inequality remains valid if multiplied with a potential for $\phi_{\Gamma_2} = \Phi(\Gamma_2; T)$, that is, $\Phi(\Gamma_1; S) \cdot \phi_{\Gamma_2} \geq \Phi(v{:}(A, S')) \cdot \phi_{\Gamma_2}$. To relate the mixed potential annotations we thus derive a cost-free judgment $\Gamma_1; P_j \vdash^{\text{cf}} e_1 : (A, P'_j)$

for every $\vec{0} \neq j \in I(\Gamma_2)$. (We use cost-free judgments to avoid paying multiple times for the evaluation of e_1.) Then we equate P_j to the corresponding annotations in Q and equate P'_j to the corresponding annotations in R, that is, $P_j = \pi_j^{\Gamma_1}(Q)$ and $P'_j = \pi_j^{x:A}(R)$. The intuition is that j corresponds to ϕ_{Γ_2}. Note that we use a fresh signature Σ in the derivation of each cost-free judgment for e_1.

6.4 Soundness

The main theorem of this chapter states that type derivations establish correct bounds: an annotated type judgment for an expression e shows that if e evaluates to a value v in a well-formed environment then the initial potential of the context is an upper bound on the watermark of the resource usage and the difference between initial and final potential is an upper bound on the consumed resources.

As in Chapter 4 and Chapter 5, I use the partial evaluation judgments to prove that the bounds derived from an annotated type judgment also apply to non-terminating evaluations.

Theorem 6.4.1 (Soundness) Let $H \vDash V{:}\Gamma$ and $\Sigma;\Gamma;Q \vdash e{:}(B,Q')$.

1. If $V, H \vdash e \rightsquigarrow v, H' \mid (p,p')$ then we have $p \geq \Phi_{V,H}(\Gamma;Q)$ and $p - p' \geq \Phi_{V,H}(\Gamma;Q) - \Phi_{H'}(v{:}(B,Q'))$.

2. If $V, H \vdash e \rightsquigarrow \mid p$ then we have $p \geq \Phi_{V,H}(\Gamma;Q)$.

Like the soundness theorems in the previous chapters, Theorem 6.4.1 is proved by a nested induction on the derivation of the evaluation judgment—$V, H \vdash e \rightsquigarrow v, H' \mid (p,p')$ or $V, H \vdash e \rightsquigarrow \mid p$, respectively—and the type judgment $\Gamma;Q \vdash e{:}(B,Q')$. The inner induction on the type judgment is needed because of the structural rules.

Compared to the previous soundness proofs, further complexity arises from the rich multivariate potential annotations. It is mainly dealt with in Lemmas 6.3.4, 6.3.5, and 6.3.6 and the concept of projections as explained in Propositions 6.3.2 and 6.3.3.

Note that I could define an embedding of the linear into the multivariate polynomial system so as to derive the soundness of the linear system as corollary. This would however not be possible for univariate system from Chapter 5 since the univariate potential of trees is not compatible with the potential of trees that I use here.

It follows from Theorem 6.4.1 and Theorem 3.3.9 that run-time bounds also prove the termination of programs. Corollary 6.4.2 states this fact formally.

Corollary 6.4.2 Let the resource constants be instantiated by $K^x = 1$, $K_1^x = 1$ and $K_m^x = 0$ for all x and all $m > 1$. If $H \vDash V{:}\Gamma$ and $\Sigma;\Gamma;Q \vdash e{:}(A,Q')$ then there is an $n \in \mathbb{N}, n \leq \Phi_{V,H}(\Gamma;Q)$ such that $V, H \vdash e \rightsquigarrow v, H' \mid (n,0)$.

Lemma 6.4.3 is used to show the soundness of the rule M:LET. It states that the potential of a context is invariant during the evaluation. This is a consequence of allocated heap-cells being immutable with the language features that I describe in this dissertation.

Lemma 6.4.3 Let $H \vDash V{:}\Gamma$, $\Sigma;\Gamma;Q \vdash e : (B, Q')$ and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$. Then it is true that $\Phi_{V,H}(\Gamma;Q) = \Phi_{V,H'}(\Gamma;Q)$.

PROOF The lemma is a direct consequence of the definition of the potential Φ and the fact that $H'(\ell) = H(\ell)$ for all $\ell \in \mathrm{dom}(H)$ which is proved in Proposition 3.3.2. ∎

Soundness Proof

In the following I prove Theorem 6.4.1. I first present the details of the proof of part 1 and then I describe some cases of the proof of part 2 to convince you that the proof is similar.

PROOF (PART 1) I prove $p \leq \Phi_{V,H}(\Gamma;Q)$ and $p - p' \leq \Phi_{V,H}(\Gamma;Q) - \Phi_{H'}(v{:}(B,Q'))$ by induction on the derivations of $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ and $\Sigma;\Gamma;Q \vdash e : (B, Q')$, where the induction on the evaluation judgment takes priority.

(M:SHARE) Suppose that the derivation of $\Sigma;\Gamma;Q \vdash e : (B, Q')$ ends with an application of the rule M:SHARE. Then $\Gamma = \Gamma', z{:}A$. It follows from the premise that

$$\Gamma', x{:}A, y{:}A; P \vdash e' : (B, Q') \tag{6.1}$$

for a type annotation P with $Q = \curlyvee P$ and an expression e' with $e'[z/x, z/y] = e$. Since $H \vDash V : \Gamma', z{:}A$ and $V, H \vdash e \rightsquigarrow v, H' \mid (p, p')$ it follows that $H \vDash V_{xy} : \Gamma', x{:}A, y{:}A$ and

$$V_{xy}, H \vdash e' \rightsquigarrow v, H' \mid (p, p') \tag{6.2}$$

where $V_{xy} = V \cup \{x \mapsto V(x), y \mapsto V(z)\}$. Thus we can apply the induction hypothesis to (6.1) and (6.2) to derive

$$p \leq \Phi_{V_{xy},H}(\Gamma', x{:}A, y{:}A; P) \tag{6.3}$$

and

$$p - p' \leq \Phi_{V_{xy},H}(\Gamma', x{:}A, y{:}A; P) - (\Phi_{H'}(v{:}(B, Q'))) \,. \tag{6.4}$$

From the definition of the sharing annotation $\curlyvee Q$ (compare Lemma 6.3.6) it follows that

$$\Phi_{V_{xy},H}(\Gamma', x{:}A, y{:}A; P) = \Phi_{V,H}(\Gamma', z{:}A; Q) \,. \tag{6.5}$$

The claim follows from (6.3), (6.4), and (6.5).

(M:AUGMENT) If the derivation of $\Sigma;\Gamma;Q \vdash e : (B, Q')$ ends with an application of the rule M:AUGMENT then we have $\Sigma;\Gamma';Q \vdash e : (B, Q')$ for a context Γ' with $\Gamma', x{:}A = \Gamma$. From the assumption $H \vDash V : \Gamma', x{:}A$ it follows that $H \vDash V : \Gamma'$. Thus we can apply the induction hypothesis to the premise $\Gamma'; \pi_{\vec{0}}^{\Gamma'}(Q) \vdash e : (B, Q')$ of M:AUGMENT. We derive

$$p \leq \Phi_{V,H}(\Gamma'; \pi_{\vec{0}}^{\Gamma'}(Q)) \tag{6.6}$$

and

$$p - p' \leq \Phi_{V,H}\big(\Gamma'; \pi_{\vec{0}}^{\Gamma'}(Q)\big) - (\Phi_{H'}(v{:}(B, Q'))\big) \,. \tag{6.7}$$

Assume that $H \vDash V(x) \mapsto a : A$. From Proposition 6.3.3 it follows that $\Phi_{V,H}(\Gamma'; \pi_{\vec{0}}^{\Gamma'}(Q)) = \Phi_{V,H}(\Gamma'; \pi_{\vec{0}}^{\Gamma}(Q)) \cdot p_{\vec{0}}(a) \leq \sum_{j \in I(A)} \Phi_{V,H}(\Gamma'; \pi_j^{\Gamma'}(Q)) \cdot p_j(a) = \Phi_{V,H}(\Gamma', x{:}A; Q)$. Hence we have

$$\Phi_{V,H}(\Gamma'; \pi_{\vec{0}}^{\Gamma'}(Q)) \leq \Phi_{V,H}(\Gamma; Q) \tag{6.8}$$

and the claim follows from (6.6), (6.7), and (6.8).

(M:WEAKEN) Assume the derivation of the typing judgment ends with an application of the type rule M:WEAKEN. Then we have $\Gamma; P \vdash e : (B, P')$, $Q \geq P$, and $Q' \leq P'$. We can conclude by induction that

$$p \leq \Phi_{V,H}(\Gamma; P) \qquad \text{and} \qquad p - p' \leq \Phi_{V,H}(\Gamma; P) - \Phi_{H'}(v{:}(B, P')) \,. \tag{6.9}$$

From the definition of $\leq$ for type annotations it follows immediately that

$$\Phi_{V,H}(\Gamma; Q) \geq \Phi_{V,H}(\Gamma; P) \qquad \text{and} \qquad \Phi_{H'}(v{:}(B, P')) \leq \Phi_{H'}(v{:}(B, Q')) \,. \tag{6.10}$$

The claim follows then from (6.9) and (6.10).

(M:OFFSET) The case M:OFFSET is similar to the case M:WEAKEN.

(M:VAR) Assume that e is a variable x that has been evaluated with the rule E:VAR. Then it is true that $H = H'$. The type judgment $\Sigma; \Gamma; Q \vdash x{:}(B, Q')$ has been derived by a single application of the rule M:VAR. Thus we have $\Gamma = x{:}B$,

$$\Phi_{V,H}(x{:}B; Q) - \Phi_{V,H'}(x{:}(B, Q')) = K^{\text{var}} \tag{6.11}$$

and in particular $\Phi_{V,H}(x{:}B; Q) \geq K^{\text{var}}$.

Assume first that $K^{\text{var}} \geq 0$. Then it follows by definition that $p = K^{\text{var}}$, $p' = 0$ and thus $p - p' = K^{\text{var}}$. The claim follows from (6.11). Assume now that $K^{\text{var}} < 0$. Then it follows by definition that $p = 0$, $p' = -K^{\text{var}}$. We have again that $p - p' = K^{\text{var}}$ and the claim follows from (6.11). (Remember that we have the implicit side condition that $\Phi_{V,H}(x{:}B; Q) \geq 0$.)

(M:CONST*) Similar to the case (M:VAR).

(M:OPINT) Assume that the type derivation ends with an application of the rule M:OPINT. Then e has the form x_1 *op* x_2 and the evaluation consists of a single application of the rule E:BINOP. From the rule M:OPINT it follows that $\Gamma = x_1{:}\mathit{int}, x_2{:}\mathit{int}$ and $\Phi_{V,H}(x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q) - \Phi_{V,H'}(v : (\mathit{int}, Q')) = q_{(0,0)} - q'_0 = K^{op}$.

If $K^{op} \geq 0$ then $p = K^{op}$ and $p' = 0$. Thus $p = K^{op} \leq q_{(0,0)} = \Phi_{V,H}(x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q)$ and $p - p' = K^{op} = \Phi_{V,H}(x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q) - (\Phi_{V,H'}(v{:}(\mathit{int}, Q'))$.

If $K^{op} < 0$ then $p = 0$ and $p' = -K^{op}$. Thus $p \leq q = \Phi_{V,H}(x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q)$ and $p - p' = K^{op} = \Phi_{V,H}(x_1{:}\mathit{int}, x_2{:}\mathit{int}; Q) - (\Phi_{V,H'}(v : (\mathit{int}, Q')))$.

(M:OPBOOL) The case in which the type derivation ends with an application of M:OPBOOL is similar to the case (M:OPINT).

(M:LET) If the type derivation ends with an application of M:LET then e is a *let* expression of the from *let* $x = e_1$ *in* e_2 that has eventually been evaluated with the rule E:LET. Then it follows that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (r, r')$ and $V', H_1 \vdash e_2 \rightsquigarrow v_2, H_2 \mid (t, t')$ for $V' = V[x \mapsto v_1]$ and r, r', t, t' with

$$(p, p') = K_1^{\text{let}} \cdot (r, r') \cdot K_2^{\text{let}} \cdot (t, t') \cdot K_3^{\text{let}} . \tag{6.12}$$

The derivation of the type judgment for e ends with an application of L:LET. Hence $\Gamma = \Gamma_1, \Gamma_2$, $\quad \Sigma; \Gamma_1; P \vdash e_1 : (A, P')$, $\Sigma; \Gamma_2, x{:}A; R \vdash e_2 : (B, R')$, and

$$\begin{aligned} P + K_1^{\text{let}} &= \pi_{\vec{0}}^{\Gamma_1}(Q) && (6.13)\\ P' &= \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}} && (6.14)\\ R' &= Q' + K_3^{\text{let}} && (6.15) \end{aligned}$$

Furthermore we have for every $\vec{0} \neq j \in I(\Gamma_2)$: $\quad \Gamma_1; P_j \overset{\text{cf}}{\vdash} e_1 : (A, P'_j)$,

$$\begin{aligned} P_j &= \pi_j^{\Gamma_1}(Q) && (6.16)\\ P'_j &= \pi_j^{x:A}(R) && (6.17) \end{aligned}$$

Since $H \vDash V : \Gamma$ we have also $H \vDash V : \Gamma_1$ and can thus apply the induction hypothesis for the evaluation judgment of e_1 to derive

$$\begin{aligned} r &\leq \Phi_{V,H}(\Gamma_1; P) && (6.18)\\ r - r' &\leq \Phi_{V,H}(\Gamma_1; P) - \Phi_{H_1}(v_1{:}(A, P')) && (6.19) \end{aligned}$$

From Theorem 3.3.4 it follows that $H_2 \vDash V' : \Gamma_2, x{:}A$ and thus again by induction

$$\begin{aligned} t &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A; R) && (6.20)\\ t - t' &\leq \Phi_{V',H_1}(\Gamma_2, x{:}A; R) - \Phi_{H_2}(v_2{:}(B, R')) && (6.21) \end{aligned}$$

Furthermore we apply the induction hypothesis to the evaluation judgment for e_1 with the cost-free metric. Then we have $r = r' = 0$ and therefore for every $\vec{0} \neq j \in I(\Gamma_2)$

$$\Phi_{V,H}(\Gamma_1; P_j) \geq \Phi_{H_1}(v_1{:}(A, P'_j)) . \tag{6.22}$$

Let $\Gamma_1 = x_1, \ldots, x_n$, $\Gamma_2 = y_1, \ldots, y_m$, $H \vDash V(x_j) \mapsto a_{x_j} : \Gamma(x_j)$, and $H \vDash V(y_j) \mapsto b_{y_j} : \Gamma(y_j)$. Define

$$\begin{aligned} \phi_P &= \Phi_{V,H}(\Gamma_1; P) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})\\ \phi_{P'} &= \Phi_{H_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{H_1}(v_1{:}(A, P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \end{aligned}$$

We argue that

$$\begin{aligned}
\Phi_{V,H}(\Gamma_1,\Gamma_2;Q) &\overset{\text{Prop. 6.3.3}}{=} \sum_{\vec{j}\in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1;\pi_{\vec{j}}^{\Gamma_1}(Q)) \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&\overset{(6.13,6.16)}{=} \Phi_{V,H}(\Gamma_1;P) + K_1^{\text{let}} + \sum_{\vec{0}\neq\vec{j}\in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1;P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&= \phi_P + K_1^{\text{let}} \qquad (6.23)
\end{aligned}$$

Similarly, we use Proposition 6.3.3, (6.14), and (6.17) to see that

$$\phi_{P'} = \Phi_{V',H_1}(\Gamma_2, x{:}A; R) + K_2^{\text{let}} \qquad (6.24)$$

Additionally we have

$$\begin{aligned}
r - r' &\overset{(6.19)}{\leq} \Phi_{V,H}(\Gamma_1;P) - \Phi_{H_1}(v_1{:}(A,P')) \\
&\overset{(6.22)}{\leq} \Phi_{V,H}(\Gamma_1;P) - \Phi_{H_1}(v_1{:}(A,P')) + \sum_{\vec{0}\neq\vec{j}\in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1;P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&\quad - \sum_{\vec{0}\neq\vec{j}\in I_k(\Gamma_2)} \Phi_{H_1}(v_1{:}(A,P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&= \phi_P - \phi_{P'} \qquad (6.25)
\end{aligned}$$

Now let

$$(u,u') = K_1^{\text{let}} \cdot (\phi_P, \phi_{P'}) \cdot K_2^{\text{let}} \cdot (\Phi_{V',H_1}(\Gamma_2, x{:}A; R), \Phi_{H_2}(v_2{:}(B,R'))) \cdot K_3^{\text{let}}$$

Then it follows that

$$\begin{aligned}
(u,u') &\overset{(6.15,6.24)}{=} K_1^{\text{let}} \cdot (\phi_P, \phi_{P'} - K_2^{\text{let}}) \cdot (\Phi_{V',H_1}(\Gamma_2, x{:}A; R), \Phi_{H_2}(v_2{:}(B,R')) - K_3^{\text{let}}) \\
&\overset{(6.24)}{=} K_1^{\text{let}} \cdot (\phi_P, v')
\end{aligned}$$

for some $v' \in \mathbb{Q}_0^+$. Now we can conclude that

$$u \leq \max(0, \phi_P + K_1^{\text{let}}) \overset{(6.23)}{\leq} \Phi_{V,H}(\Gamma;Q)$$

Finally, it follows with Proposition 3.3.1 applied to (6.18), (6.25), (6.20), (6.21), and (6.12) that $u \geq p$.

For the second part of the statement we apply Proposition 3.3.1 to (6.12) and derive the following.

$$\begin{aligned}
p - p' &= r - r' + t - t' + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
&\overset{(6.21,6.25)}{\leq} \phi_P - \phi_{P'} + \Phi_{V',H_1}(\Gamma_2, x{:}A; R) - \Phi_{H_2}(v_2{:}(B,R')) + K_1^{\text{let}} + K_2^{\text{let}} + K_3^{\text{let}} \\
&\overset{(6.24)}{=} \phi_P - \Phi_{H_2}(v_2{:}(B,R')) + K_1^{\text{let}} + K_3^{\text{let}} \\
&\overset{(6.23)}{=} \Phi_{V,H}(\Gamma;Q) - \Phi_{H_2}(v_2{:}(B,R')) + K_3^{\text{let}} \\
&\overset{(6.15)}{\leq} \Phi_{V,H}(\Gamma;Q) - \Phi_{H_2}(v_2{:}(B,Q'))
\end{aligned}$$

(M:APP) Assume that e is a function application of the form $f(x)$. The evaluation of e then ends with an application of the rule E:APP. Thus we have $V(x) = v'$ and $[y_f \mapsto v'], H \vdash e_f \rightsquigarrow v, H' \mid (r, r')$ for some r, r' with

$$(p, p') = K_1^{\text{app}} \cdot (r, r') \cdot K_2^{\text{app}} \tag{6.26}$$

The derivation of the type judgment for e ends with an application of M:FUN. Therefore it is true that $\Gamma = x{:}A; Q$, $(A, P) \to (B, P') \in \Sigma(f)$, and

$$P + K_1^{\text{app}} = Q \qquad \text{and} \qquad P' = Q' + K_2^{\text{app}}\,. \tag{6.27}$$

In order to apply the induction hypothesis to the evaluation of the function body e_f we recall from the definition of a well-formed program that $(A, P) \to (B, P') \in \Sigma(f)$ implies that $\Sigma; y_f{:}A; P \vdash e_f{:}P'$. Since $H \vDash V : x{:}A$ and $V(x) = v'$ it follows $H \vDash [y_f \mapsto v'] : y_f{:}A$. We obtain by induction that

$$\begin{aligned} r &\leq \Phi_{[y_f \mapsto v'],H}(y_f{:}A; P) && (6.28)\\ r - r' &\leq \Phi_{[y_f \mapsto v'],H}(y_f{:}A; P) - \Phi_{H'}(v{:}(B, P')) && (6.29) \end{aligned}$$

Now define

$$(u, u') = K_1^{\text{app}} \cdot (\Phi_{[y_f \mapsto v'],H}(y_f{:}A; P), \Phi_{H'}(v{:}(B, P'))) \cdot K_2^{\text{app}}\,. \tag{6.30}$$

From (6.27) it follows that $\Phi_{H'}(v{:}(B, P') \geq K_2^{\text{app}}$ and hence we obtain $u = \max(0, K_1^{\text{app}} + \Phi_{[y_f \mapsto v'],H}(y_f{:}A; P))$. We apply Proposition 3.3.1 to (6.26), (6.28), (6.29), (6.30) and obtain $p \leq u$. If $u = 0$ then $p = 0 \leq \Phi_{V,H}(x{:}A; Q)$. Otherwise we have $u = \Phi_{[y_f \mapsto v']H}(y_f{:}A; P)) + K_1^{\text{app}}$. Furthermore it follows from (6.27) that $\Phi_{[y_f \mapsto v'],H}(y_f{:}A; P) + K_1^{\text{app}} = \Phi_{V,H}(x{:}A; Q)$ and therefore $p \leq \Phi_{V,H}(x{:}A; Q)$.

For the second part for the statement observe that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{app}} + K_2^{\text{app}} \\ &\overset{(6.29)}{\leq} \Phi_{[y_f \mapsto v'],H}(y_f{:}A; P) - \Phi_{H'}(v{:}(B, P')) + K_1^{\text{app}} + K_2^{\text{app}} \\ &\overset{(6.27)}{=} \Phi_{V,H}(x{:}A; Q) - \Phi_{H'}(v{:}(B, Q')) \end{aligned}$$

(M:NIL) If the type derivation ends with an application of M:NIL then we have $e = \mathit{nil}$, $\Gamma = \emptyset$, $B = L(A)$ for some A, and $q_0 = q'_{\vec{0}} + K^{\text{nil}}$. The corresponding evaluation rule E:NIL has been applied to derive the evaluation judgment and hence $v = \text{NULL}$.

If $K^{\text{nil}} \geq 0$ then $p = K^{\text{nil}}$ and $p' = 0$. Thus $p = K^{\text{nil}} \leq q_0 = \Phi_{V,H}(\emptyset, Q)$. Furthermore it follows from the definition of Φ that $\Phi_{V,H'}(\text{NULL}{:}(L(A), Q')) = q'_{\vec{0}}$. Thus $p - p' = K^{\text{nil}} = \Phi_{V,H}(\emptyset; Q) - \Phi_{V,H'}(\text{NULL}{:}(L(A), Q'))$. If $K^{\text{nil}} < 0$ then $p = 0$ and $p' = -K^{\text{nil}}$. Then clearly $p \leq \Phi_{V,H}(\emptyset, Q)$ and again $p - p' = K^{\text{nil}}$.

(M:CONS) If the type derivation ends with an application of the rule M:CONS then e has the form $\mathit{cons}(x_h, x_t)$ and it has been evaluated with the rule E:CONS. It

follows by definition that $V, H \vdash \mathit{cons}(x_h, x_t) \rightsquigarrow \ell, H[\ell \mapsto v'] \mid K^{\text{cons}}$, $x_h, x_t \in \text{dom}(V)$, $v' = (V(x_h), V(x_t))$, and $\ell \notin \text{dom}(H)$. Thus

$$p = K^{\text{cons}} \text{ and } p' = 0 \qquad \text{or (if } K^{\text{cons}} < 0) \qquad p = 0 \text{ and } p' = -K^{\text{cons}}$$

Furthermore $B = L(A)$ and the judgment $\Sigma; x_h{:}A, x_t{:}L(A); Q \vdash \mathit{cons}(x_h, x_t) : (L(A), Q')$ has been derived by a single application of the rule M:CONS; thus

$$Q = \lhd_L(Q') + K^{\text{cons}} \,. \tag{6.31}$$

If $p = 0$ then $p \le \Phi_{V,H}(\Gamma; Q)$ follows because potential is always non-negative. Otherwise we have $p = K^{\text{cons}} \le \Phi_{V,H}(\Gamma; Q)$ from (6.31).

From Lemma 6.3.4 it follows that $\Phi_{V,H}(x_h{:}A, x_t{:}L(A); \lhd_L(Q')) = \Phi_{V,H'}(\ell{:}(L(A), Q'))$ and therefrom with (6.31) $\Phi_{V,H}(x_h{:}A, x_t{:}L(A); Q) - \Phi_{V,H[\ell\mapsto v']}(\ell{:}(L(A), Q')) = K^{\text{cons}} = p - p'$.

(M:MATL) Assume that the type derivation of e ends with an application of the rule M:MATL. Then e is a pattern match of the form $\mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2$ whose evaluation ends with an application of the rule E:MATCONS or E:MATNIL. Assume first that the derivation of the evaluation judgment ends with an application of E:MATCONS.

Then $V(x) = \ell$, $H(\ell) = (v_h, v_t)$, and $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ for $V' = V[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r, r' with

$$(p, p') = K_1^{\text{matC}} \cdot (r, r') \cdot K_2^{\text{matC}} \tag{6.32}$$

Since the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q)$ ends with an application of M:MATL, we have $\Gamma = \Gamma', x{:}L(A)$, $\Sigma; \Gamma', x_h{:}A, x_t{:}L(A); P \vdash e_2 : (B, P')$,

$$P + K_1^{\text{matC}} = \lhd_L(Q) \qquad \text{and} \qquad P' = Q' + K_2^{\text{matC}} \tag{6.33}$$

It follows from Lemma 6.3.4 that

$$\Phi_{V,H}(\Gamma; Q) = \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q)) \,. \tag{6.34}$$

Since $H \vDash V' : \Gamma', x_h{:}A, x_t{:}L(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ and obtain

$$r \le \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) \tag{6.35}$$

$$r - r' \le \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) - \Phi_{H'}(v{:}(B, P')) \tag{6.36}$$

We define

$$(u, u') = K_1^{\text{matC}} \cdot (\Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P), \Phi_{H'}(v{:}(B, P'))) \cdot K_2^{\text{matC}} \,. \tag{6.37}$$

Per definition and from (6.33) it follows that $\Phi_{H'}(v{:}(B, P')) \ge K_2^{\text{matC}}$ and thus we have $u = \max(0, \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}})$. From Proposition 3.3.1 applied to (6.35),

(6.36), (6.37) and (6.32) we derive $u \geq p$. If $\Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} \leq 0$ then $u = p = 0$ and $\Phi_{V,H}(\Gamma; Q) \geq p$ trivially holds. If $\Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} > 0$ then it follows from (6.33) and (6.34) that

$$\Phi_{V,H}(\Gamma; Q) = \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) + K_1^{\text{matC}} = u \geq p\,.$$

Finally, we apply Proposition 3.3.1 to (6.32) to see that

$$\begin{array}{rcl} p - p' & = & r - r' + K_1^{\text{matC}} + K_2^{\text{matC}} \\ & \overset{(6.36)}{\leq} & \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); P) - \Phi_{H'}(v{:}(B, P')) + K_1^{\text{matC}} + K_2^{\text{matC}} \\ & \overset{(6.33)}{=} & \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q))) - \Phi_{H'}(v{:}(B, Q')) \\ & \overset{(6.34)}{=} & \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(v{:}(B, Q')) \end{array}$$

Assume now that the derivation of the evaluation judgment ends with an application of E:MatNil. Then $V(x) = \text{Null}$, and $V, H \vdash e_1 \rightsquigarrow v, H' \mid (r, r')$ for some r, r' with

$$(p, p') = K_1^{\text{matN}} \cdot (r, r') \cdot K_2^{\text{matN}}\,. \tag{6.38}$$

Since the derivation of $\Sigma; \Gamma; q \vdash e : (B, Q')$ ends with an application of M:MatL, we have $\Sigma; \Gamma; R \vdash e_1 : (B, R')$,

$$R + K_1^{\text{matN}} = \pi_0^{\Gamma}(Q) \qquad \text{and} \qquad R' = Q' + K_2^{\text{matN}} \tag{6.39}$$

From Proposition 6.3.3 it follows that

$$\Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}} \leq \Phi_{V,H}(\Gamma; Q) \tag{6.40}$$

Because $H \vDash V : \Gamma$ we can apply the induction hypothesis to $V, H \vdash e_1 \rightsquigarrow v, H' \mid (r, r')$ and obtain

$$r \leq \Phi_{V,H}(\Gamma; R) \tag{6.41}$$

$$r - r' \leq \Phi_{V,H}(\Gamma; R) - \Phi_{H'}(v{:}(B, R')) \tag{6.42}$$

Now let

$$(u, u') = K_1^{\text{matN}} \cdot (\Phi_{V,H}(\Gamma; R), \Phi_{H'}(v{:}(B, R'))) \cdot K_2^{\text{matN}}\,. \tag{6.43}$$

Per definition and from (6.39) it follows that $u = \max(0, \Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}})$. From Proposition 3.3.1 applied to (6.41), (6.42), (6.43) and (6.38) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}} \leq 0$ then $u = p = 0$ and $\Phi_{V,H}(\Gamma; Q) \geq p$ trivially holds. If $\Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}} > 0$ then it follows from (6.40) that

$$\Phi_{V,H}(\Gamma; Q) \geq \Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}} = u \geq p\,.$$

Finally, we apply Proposition 3.3.1 to (6.38) to see that

$$\begin{array}{rcl} p - p' & = & r - r' + K_1^{\mathrm{matN}} + K_2^{\mathrm{matN}} \\ & \overset{(6.42)}{\leq} & \Phi_{V,H}(\Gamma; R) - \Phi_{H'}(v{:}(B, R')) + K_1^{\mathrm{matN}} + K_2^{\mathrm{matN}} \\ & \overset{(6.40)}{\leq} & \Phi_{V,H}(\Gamma; Q) - (\Phi_{H'}(v{:}(B, R)) - K_2^{\mathrm{matN}}) \\ & \overset{(6.39)}{=} & \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(v{:}(B, Q')) \end{array}$$

(M:LEAF) This case is nearly identical to the case (M:NIL).

(M:NODE) If the type derivation ends with an application of the rule M:NODE then e has the form $\mathit{node}(x_0, x_1, x_2)$ and it has been evaluated with the rule E:NODE. It follows from the definition that $V, H \vdash \mathit{node}(x_0, x_1, x_2) \rightsquigarrow \ell, H[\ell \mapsto v'] \mid K^{\mathrm{node}}$, $v' = (V(x_0), V(x_1), V(x_2))$, and $\ell \notin \mathrm{dom}(H)$. Thus

$$p = K^{\mathrm{node}} \text{ and } p' = 0 \qquad \text{or (if } K^{\mathrm{node}} < 0) \qquad p = 0 \text{ and } p' = -K^{\mathrm{node}}$$

Furthermore we have $B = T(A)$ and the type judgment $x_0{:}A, x_1{:}T(A), x_2{:}T(A); Q \vdash \mathit{node}(x_0, x_1, x_2) : (T(A), Q')$ has been derived by a single application of the rule M:NODE; thus

$$Q = \lhd_T(Q') + K^{\mathrm{node}}\,. \tag{6.44}$$

If $p = 0$ then clearly $p \leq \Phi_{V,H}(\Gamma; Q)$. Otherwise we have $p = K^{\mathrm{node}} \leq \Phi_{V,H}(\Gamma; Q)$ from (6.44). From Lemma 6.3.5 it follows that

$$\Phi_{V,H}(x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q')) = \Phi_{V,H'}(\ell{:}(T(A), Q'))$$

and therefrom with (6.44)

$$\Phi_{V,H}(x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q')) - \Phi_{V,H[\ell\mapsto v']}(\ell{:}(T(A), Q')) = K^{\mathrm{node}} = p - p'\,.$$

(M:MATT) Assume that the type derivation of e ends with an application of the rule M:MATT. Then e is a pattern match $\mathit{match}\ x\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2$ whose evaluation ends with an application of the rule E:MATNODE or E:MATLEAF. The case E:MATLEAF is similar to the case E:MATNIL. So assume that the derivation of the evaluation judgment ends with an application of E:MATNODE.

Then $V(x) = \ell$, $H(\ell) = (v_0, v_1, v_2)$, and $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ for $V' = V[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2]$ and some r, r' with

$$(p, p') = K_1^{\mathrm{matTL}} \cdot (r, r') \cdot K_2^{\mathrm{matTL}}\,. \tag{6.45}$$

Since the derivation of $\Sigma; \Gamma; Q \vdash e : (B, Q)$ ends with an application of M:MATT, we have $\Gamma = \Gamma', x{:}T(A)$, $\Sigma; \Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P \vdash e_2 : (B, P')$,

$$P + K_1^{\mathrm{matTN}} = \lhd_L(Q) \qquad \text{and} \qquad P' = Q' + K_2^{\mathrm{matTN}}\,. \tag{6.46}$$

It follows from Lemma 6.3.5 that

$$\Phi_{V,H}(\Gamma; Q) = \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_T(Q))\,. \tag{6.47}$$

Since we have $H \vDash V' : \Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow v, H' \mid (r, r')$ and obtain

$$\begin{aligned} r &\leq \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) && (6.48)\\ r - r' &\leq \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) - \Phi_{H'}(v{:}(B, P')) && (6.49)\end{aligned}$$

We define

$$(u, u') = K_1^{\text{matTN}} \cdot (\Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P), \Phi_{H'}(v{:}(B, P'))) \cdot K_2^{\text{matTN}}\,. \tag{6.50}$$

Per definition and from (6.46) it follows that $\Phi_{H'}(v{:}(B, P')) \geq K_2^{\text{matTN}}$ and thus $u = \max(0, \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}})$. From Proposition 3.3.1 applied to (6.48), (6.49), (6.50) and (6.45) we derive $u \geq p$. If $\Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} \leq 0$ then $u = p = 0$ and $\Phi_{V,H}(\Gamma; Q) \geq p$ trivially holds. If we otherwise have $\Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} > 0$ then it follows from (6.46) and (6.47) that

$$\Phi_{V,H}(\Gamma; Q) = \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) + K_1^{\text{matTN}} = u \geq p\,.$$

Finally, we apply Proposition 3.3.1 to (6.45) to see that

$$\begin{aligned} p - p' &= r - r' + K_1^{\text{matTN}} + K_2^{\text{matTN}}\\ &\overset{(6.49)}{\leq} \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); P) - \Phi_{H'}(v{:}(B, P')) + K_1^{\text{matTN}} + K_2^{\text{matTN}}\\ &\overset{(6.46)}{=} \Phi_{V',H}(\Gamma', x_1{:}A, x_2{:}T(A), x_3{:}T(A); \lhd_L(Q))) - \Phi_{H'}(v{:}(B, Q'))\\ &\overset{(6.47)}{=} \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(v{:}(B, Q'))\end{aligned}$$

(M:PAIR) This case is similar to the case in which the type derivation ends with an application of the rule M:CONS.

(M:MATP) This case is proved like the case M:MATL.

(M:COND) This case is similar to (but also simpler than) the case M:MATL. ∎

PROOF (PART 2) The proof of part 2 is similar but simpler than the proof of part 1. However, it uses part 1 in the case of the rule P:LET2. Like in the proof of part 1, we prove $p \leq \Phi_{V,H}(\Gamma; Q)$ by induction on the derivations of $V, H \vdash e \rightsquigarrow \mid p$ and $\Sigma; \Gamma; Q \vdash e : (B, Q')$, where the induction on the partial evaluation judgment takes priority.

I only present a few cases to show that the proof is similar to the poof of part 1.

(M:VAR) Assume that e is a variable x and the type judgment $\Sigma;\ Q \vdash x : (B, Q')$ has been derived by a single application of the rule M:VAR. Thus we have $\Gamma = x{:}B$,

$$\Phi_{V,H}(x{:}B; Q) - \Phi_{V,H'}(x{:}(B, Q')) = K^{\text{var}}$$

and in particular $\Phi_{V,H}(x{:}B;Q) \geq K^{\text{var}}$.

Furthermore e has been evaluated with a single application of the rule P:VAR and it follows by definition that $p = \max(K^{\text{var}}, 0)$. (Remember that $V, H \vdash x \rightsquigarrow | K^{\text{var}}$ is an abbreviation for $V, H \vdash x \rightsquigarrow | \max(K^{\text{var}}, 0)$ in P:VAR.)

Assume first that $K^{\text{var}} \geq 0$. Then we have $p = K^{\text{var}} \leq \Phi_{V,H}(x{:}B;Q)$. Assume now that $K^{\text{var}} < 0$. Then it follows by definition that $p = 0$ and $p \leq \Phi_{V,H}(x{:}B;Q)$ trivially holds.

(M:MATL) Assume that the type derivation of e ends with an application of the rule M:MATL. Then e is a pattern match of the form $\mathit{match}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2$ whose evaluation ends with an application of the rule P:MATCONS or P:MATNIL. Assume first that the derivation of the evaluation judgment ends with an application of P:MATCONS.

Then $V(x) = l$, $H(\ell) = (v_h, v_t)$, and $V', H \vdash e_2 \rightsquigarrow | r$ for $V' = V[x_h \mapsto v_h, x_t \mapsto v_t]$ and some r with

$$p = \max(K_1^{\text{matC}} + r, 0)\,. \tag{6.51}$$

Since the derivation of $\Sigma;\Gamma;Q \vdash e : (B, Q)$ ends with an application of M:MATL, we have $\Gamma = \Gamma', x{:}L(A)$, $\Sigma;\Gamma', x_h{:}A, x_t{:}L(A);P \vdash e_2 : (B, P')$,

$$P + K_1^{\text{matC}} = \lhd_L(Q) \tag{6.52}$$

It follows from Lemma 6.3.4 that

$$\Phi_{V,H}(\Gamma;Q) = \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q))\,. \tag{6.53}$$

Since $H \vDash V' : \Gamma', x_h{:}A, x_t{:}L^t(A)$ we can apply the induction hypothesis to $V', H \vdash e_2 \rightsquigarrow | r$ and obtain

$$r \quad \leq \quad \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A);P) \tag{6.54}$$

If $p = 0$ then the claim follows immediately. Thus assume that $p = K_1^{\text{matC}} + r$. Then it follows that

$$\begin{aligned} p = K_1^{\text{matC}} + r &\overset{(6.54)}{\leq} K_1^{\text{matC}} + \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A);P) \\ &\overset{(6.52)}{\leq} K_1^{\text{matC}} + \Phi_{V',H}(\Gamma', x_h{:}A, x_t{:}L(A); \lhd_L(Q)) \\ &\overset{(6.53)}{\leq} \Phi_{V,H}(\Gamma;Q) \end{aligned}$$

Assume now that the derivation of the evaluation judgment ends with an application of P:MATNIL. Then $V, H \vdash e_1 \rightsquigarrow | r$ for an r with

$$p = \max(K_1^{\text{matN}} + r, 0)$$

Since the derivation of $\Sigma;\Gamma;q \vdash e : (B, Q')$ ends with an application of M:MATL, we have $\Sigma;\Gamma;R \vdash e_1 : (B, R')$,

$$R + K_1^{\text{matN}} = \pi_0^{\Gamma}(Q)$$

From Proposition 6.3.3 it follows that

$$\Phi_{V,H}(\Gamma; R) + K_1^{\text{matN}} \leq \Phi_{V,H}(\Gamma; Q) \tag{6.55}$$

Since $H \vDash V : \Gamma'$ we can apply the induction hypothesis to $V, H \vdash e_1 \rightsquigarrow \mid r$ and obtain

$$r \quad \leq \quad \Phi_{V,H}(\Gamma; R) \tag{6.56}$$

If $p = 0$ then the claim follows immediately. So assume that $p = K_1^{\text{matN}} + r$. Then it follows from (6.55) and (6.56) that

$$p = K_1^{\text{matN}} + r \leq K_1^{\text{matN}} + \Phi_{V,H}(\Gamma; R) \leq \Phi_{V,H}(\Gamma; Q)\,.$$

(M:LET) If the type derivation ends with an application of M:LET then e is a let expression of the from *let* $x = e_1$ *in* e_2 that has eventually been evaluated with the rule P:LET1 or with the rule P:LET2.

The case P:LET1 is similar to the case P:MATCONS. So assume that the evaluation judgment ends with an application of the rule P:LET2. Then it follows that $V, H \vdash e_1 \rightsquigarrow v_1, H_1 \mid (r, r')$ and $V', H_1 \vdash e_2 \rightsquigarrow \mid t$ for $V' = V[x \mapsto v_1]$ and r, r', t with

$$(p, p') = K_1^{\text{let}} \cdot (r, r') \cdot K_2^{\text{let}} \cdot (t, 0) \tag{6.57}$$

The derivation of the type judgment for e ends with an application of L:LET. Hence $\Gamma = \Gamma_1, \Gamma_2$, $\quad \Sigma; \Gamma_1; P \vdash e_1 : (A, P')$, $\Sigma; \Gamma_2, x{:}A; R \vdash e_2 : (B, R')$ and

$$P + K_1^{\text{let}} \quad = \quad \pi_{\vec{0}}^{\Gamma_1}(Q) \tag{6.58}$$

$$P' \quad = \quad \pi_{\vec{0}}^{x:A}(R) + K_2^{\text{let}} \tag{6.59}$$

Furthermore we have for every $\vec{0} \neq j \in I(\Gamma_2)$: $\quad \Gamma_1; P_j \overset{\text{cf}}{\vdash} e_1 : (A, P'_j)$,

$$P_j = \pi_j^{\Gamma_1}(Q) \tag{6.60}$$

$$P'_j = \pi_j^{x:A}(R) \tag{6.61}$$

Since $H \vDash V : \Gamma$ we have also $H \vDash V : \Gamma_1$ and can thus apply part 1 of the soundness theorem to the evaluation judgment of e_1 and derive

$$r \quad \leq \quad \Phi_{V,H}(\Gamma_1; P) \tag{6.62}$$

$$r - r' \quad \leq \quad \Phi_{V,H}(\Gamma_1; P) - \Phi_{H_1}(v_1{:}(A, P')) \tag{6.63}$$

From Theorem 3.3.4 it follows that $H_2 \vDash V' : \Gamma_2, x{:}A$. Thus we can apply the induction hypothesis of part 2 to the partial evaluation judgment for e_2 and obtain

$$t \quad \leq \quad \Phi_{V',H_1}(\Gamma_2, x{:}A; R) \tag{6.64}$$

Furthermore we apply part 1 of the theorem to the evaluation judgment for e_1 with the cost-free metric. Then we have $r = r' = 0$ and therefore for every $\vec{0} \neq j \in I(\Gamma_2)$

$$\Phi_{V,H}(\Gamma_1; P_j) \geq \Phi_{H_1}(v_1{:}(A, P'_j)) \,. \tag{6.65}$$

Let $\Gamma_1 = x_1, \ldots, x_n$, $\Gamma_2 = y_1, \ldots, y_m$, $H \vDash V(x_j) \mapsto a_{x_j} : \Gamma(x_j)$, and $H \vDash V(y_j) \mapsto b_{y_j} : \Gamma(y_j)$. Define

$$\begin{aligned}
\phi_P &= \Phi_{V,H}(\Gamma_1; P) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
\phi_{P'} &= \Phi_{H_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{H_1}(v_1{:}(A, P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k})
\end{aligned}$$

We argue that

$$\begin{aligned}
\Phi_{V,H}(\Gamma_1, \Gamma_2; Q) &\overset{\text{Prop. 6.3.3}}{=} \sum_{\vec{j} \in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1; \pi^{\Gamma_1}_{\vec{j}}(Q)) \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&\overset{(6.58,6.60)}{=} \Phi_{V,H}(\Gamma_1; P) + K_1^{\text{let}} + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&= \phi_P + K_1^{\text{let}}
\end{aligned} \tag{6.66}$$

Similarly, we use Proposition 6.3.3, (6.59), and (6.61) to see that

$$\phi_{P'} = \Phi_{V',H_1}(\Gamma_2, x{:}A; R) + K_2^{\text{let}} \,. \tag{6.67}$$

Additionally we have

$$\begin{aligned}
r - r' &\overset{(6.63)}{\leq} \Phi_{V,H}(\Gamma_1; P) - \Phi_{H_1}(v_1{:}(A, P')) \\
&\overset{(6.65)}{\leq} \Phi_{V,H}(\Gamma_1; P) - \Phi_{H_1}(v_1{:}(A, P')) + \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{V,H}(\Gamma_1; P_{\vec{j}}) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&\quad - \sum_{\vec{0} \neq \vec{j} \in I_k(\Gamma_2)} \Phi_{H_1}(v_1{:}(A, P'_{\vec{j}})) \cdot \prod_{k=1}^{m} p_{j_k}(b_{x_k}) \\
&= \phi_P - \phi_{P'}
\end{aligned} \tag{6.68}$$

Now let

$$(u, u') = K_1^{\text{let}} \cdot (\phi_P, \phi_{P'}) \cdot K_2^{\text{let}} \cdot (\Phi_{V',H_1}(\Gamma_2, x{:}A; R), 0) \,.$$

Then it follows that

$$\begin{aligned}
(u, u') &\overset{(6.67)}{=} K_1^{\text{let}} \cdot (\phi_P, \phi_{P'} - K_2^{\text{let}}) \cdot (\Phi_{V',H_1}(\Gamma_2, x{:}A; R), 0) \\
&\overset{(6.67)}{=} K_1^{\text{let}} \cdot (\phi_P, 0)
\end{aligned}$$

Now we can conclude that

$$u \leq \max(0, \phi_P + K_1^{\text{let}}) \overset{(6.66)}{\leq} \Phi_{V,H}(\Gamma; Q)$$

Finally, it follows with Proposition 3.3.1 applied to (6.62), (6.68), (6.64), and (6.57) that $u \geq p$. ∎

6.5 Type Inference

The type-inference algorithm for the multivariate system extends the algorithm that I presented the univariate polynomial system in Section 5.4. As for the inference methods in the previous chapters, it is not complete with respect to the type rules in Section 6.3 but it works well for the example programs we tested.

Its basis is a classic type inference generating simple linear constraints for the annotations that are collected during the inference, and that can be solved later by linear programming. In order to obtain a finite set of constraints one has to provide a maximal degree of the resource bounds. If the degree is too low then the generated linear program is unsolvable. The maximal degree can either be specified by the user or can be incremented successively after an unsuccessful analysis.

A main challenge in the inference is the handling of resource-polymorphic recursion which I believe to be of very high complexity if not undecidable in general. To deal with it practically, I employ the same heuristic that I presented for the univariate system in Chapter 5. In a nutshell, a function is allowed to invoke itself recursively with a type different from the one that is being justified (polymorphic recursion) provided that the two types differ only in lower-degree terms. In this way, one can successively derive polymorphic type schemes for higher and higher degrees; for details, see Chapter 5. The generalisation of this approach to the multivariate setting poses no extra difficulties.

The number of multivariate polynomials our type system takes into account (e.g., $nm, n\binom{m}{2}, n\binom{m}{3}, m\binom{n}{2}, m\binom{n}{3}, \binom{n}{2}\binom{m}{2}$ for a pair of integer lists if the max. degree is 4) grows exponentially in the maximal degree. Thus the number of inequalities we collect for a fixed program grows also exponentially in the given maximal degree.

Moreover, one often has to analyze function applications context-sensitively with respect to the call stack. Consider for example the expression *append(a,append(b,c))* you have to use two different types for *append*. In our prototype implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph.

To obtain a type inference that produces linear constaints, I have to develop algorithmic versions of the type rules from Section 6.3. This is described in detail for the univariate system in another article [HH10a]. It works similar for the multivariate system in this chapter. Basically, the structural type rules have to be integrated in the syntax directed rules. If the syntax-directed rules implicitly assume that two resource annotations are equal or differ by a fixed constant, an integration of the rules M:OFFSET

$$\frac{P + c + K_1^{\text{app}} = \pi_{\vec{0}}^{x:A}(Q) \qquad P' = Q' + c + K_2^{\text{app}} \qquad \Sigma(f) = (A,P) \to (A',P')}{\Gamma, x{:}A; Q \vdash^{1} f(x) : (A',Q')} \text{(A:APP1)}$$

$$\frac{\begin{array}{c} P + P_{\text{cf}} + K_1^{\text{app}} = \pi_{\vec{0}}^{x:A}(Q) \qquad P' + P'_{\text{cf}} = Q' + K_2^{\text{app}} \qquad \Sigma(f) = (A,P) \to (A',P') \\ \Sigma_{\text{cf}}(f) = (A,P_{\text{cf}}) \to (A',P'_{\text{cf}}) \qquad y_f{:}A; P_{\text{cf}} \vdash^{\text{cf}(k-1)} (A',P'_{\text{cf}}) \qquad k > 1 \end{array}}{\Gamma, x{:}A; Q \vdash^{k} f(x) : (A',Q')} \text{(A:APP)}$$

Figure 6.3: Algorithmic type rules for function application.

and M:WEAKEN enable the analysis of a wider range of programs. The rule M:AUGMENT can be eliminated by formulating the ground rules such as M:VAR or M:CONSTU for arbitrary contexts.

A difference to standard type systems is the sharing rule M:SHARE that has to be applied if the same free variable is used more than once in an expression. The rule is not problematic for the type inference and there are several ways to deal with it in practice. The easiest way is maybe to transform input programs into programs that make sharing explicit with a syntactic construct before the type inference. Such a transformation is straightforward: Each time a free variable x occurs twice in an expression e, we replace the first occurrence of x with x_1 and the second occurrence of x with x_2 obtaining a new expression e'. We then replace e with $\mathit{share}(x, x_1, x_2)\ \mathit{in}\ e'$. In this way, the sharing rule becomes a normal syntax directed rule in the type inference. Another possibility is to integrate sharing directly into the type rule for *let* expression as we did in an earlier work [HH10a]. Then you have to ensure a variable only occurs once in each function or constructor call.

Key rules for the type inference are the algorithmic versions of the rule M:APP in Figure 6.3. In contrast to the declarative versions, signatures map function names to a single function type. The judgment $\Gamma; Q \vdash^{k} e : (A,Q')$ denotes that $\Gamma; Q \vdash e : (A,Q')$ and that all type annotation in the corresponding derivation of a maximal degree of at most k. The judgment $\Gamma; Q \vdash^{\text{cf}(k)} e : (A,Q')$ states that we have $\Gamma; Q \vdash^{k} e : (A,Q')$ for the cost-free resource metric.

The rule A:APP1 is essentially the rule M:APP from section Section 6.3. It is used if the maximal degree is one and leads to a resource-monomorphic typing of recursive calls.

The rule A:APP is used if the maximal degree is greater than one. It enables resource-polymorphic recursion. More precisely, it states that one can add a cost-free typing of the function body to the function type that is given by the signature Σ. Note that $(e_f, y_f)_{f \in \text{dom}(\Sigma_{cf})}$ must be a valid RAML program with cost-free types of degree at most $k-1$. The annotated signature Σ_{cf} used can differ in every application of the rule. The idea is as follows. To pay for the resource costs of a function call $f(x)$, the available potential $(\Phi(x{:}A; \pi_{\vec{0}}^{x:A}(Q)))$ must meet the requirements of the signa-

ture of the function ($\Phi(x{:}A;P)$). Additionally available potential ($\Phi(x{:}A;P_{\text{cf}})$) can be passed to a cost-free typing of the function body. The potential after the function call ($\Phi(f(x){:}(A',Q'))$) is then the sum of the potentials that are assigned by the cost-free typing ($\Phi(f(x){:}(A',P_{\text{cf}}))$) and by the function signature ($\Phi(f(x){:}(A',P))$). As a result, $f(x)$ can be used resource-polymorphically with a specific typing for each recursive call while the resource monomorphic function signature enables an efficient type inference.

The inference can be informally described as follows.

1. Annotate the signature of each function $f \in F$ with fresh resource variables.
2. Use the algorithmic type rules to type the corresponding expressions e_f. Introduce fresh resource variables for each type annotation in the derivation and collect the corresponding inequalities.
   (a) For a function application $g \in F$: if the maximal degree is 1 use the function resource-monomorphically with the signature from 1. using the rule A:APP1. If the maximal degree is greater than 1, go to 1. and derive a cost-free typing of e_g with a fresh signature. Store the arising inequalities and use the resource variables from the obtained typing together with the signature from 1. in the rule A:APP.
   (b) For a function application $g \notin F$: repeat the algorithm for the strongly connected component of g. Store the arising inequalities and use the obtained annotated type of g.

In contrast to the univariate system (Chapter 5), cost-free type derivations also depend on resource-polymorphic recursion to assign super-linear potential to function results. A simple example is the function *append*.

```
append(l,ys) = match l with | nil  → ys
                            | (x::xs)  → x::append(xs,ys)
```

The following linear cost-free type can be derived resource-monomorphically.

$$\mathit{append}\colon ((L(\mathit{int}),L(\mathit{int})),\begin{pmatrix}0&1&0\\1&0&\\0&&\end{pmatrix}) \to (L(\mathit{int}),(0,1,0))$$

To derive the quadratic cost-free type

$$\mathit{append}\colon ((L(\mathit{int}),L(\mathit{int})),\begin{pmatrix}0&0&1\\0&1&\\1&&\end{pmatrix}) \to (L(\mathit{int}),(0,0,1))$$

one needs however the resource-polymorphic typing

$$\mathit{append}\colon ((L(\mathit{int}),L(\mathit{int})),\begin{pmatrix}0&1&1\\1&1&\\1&&\end{pmatrix}) \to (L(\mathit{int}),(0,1,1))$$

in the recursive call.

6.6 Examples

In the following, I demonstrate the multivariate analysis with several example programs. The aim of the examples is to illustrate *how* the analysis works. You can find more realistic example programs in Chapter 7.

Multivariate Tuples

In Section 5.5, I presented canonical examples with a (univariate) polynomial heap-space consumption. Namely, functions that compute the subsets of size k for a given set (represented as a list) and a fixed k.

The canonical examples with a multivariate polynomial heap-space consumption implement the following functions. Given a fixed $k \in \mathbb{N}$ and k lists $\ell_1, \ldots, \ell_k$, compute a list of all k-tuples $(a_1, \ldots, a_k)$ such that a_i is an element of the list ℓ_i.

I define the functions for $k = 2$ (*mPairs*) and $k = 3$ (*mTriples*). The expression *mPairs([1,2],[3,4])* evaluates for instance to *[(1,3),(1,4),(2,3),(2,4)]*. You can then already see how you can implement similar functions for larger k.

```
attach2: (int,L(int)) → L(int,int)

attach2(n,l) = match l with | nil → nil
                | (x::xs) → (n,x)::attach2(n,xs);

append2: (L(int,int),L(int,int)) → L(int,int)

append2(l1,l2) = match l1 with | nil → l2
                 | (x::xs) → x::append2(xs,l2);

mPairs : (L(int),L(int)) → L(int,int)

mPairs (l1,l2) = match l1 with | nil → nil
           | (x::xs) → append2(attach2(x,l2),mPairs(xs,l2));

attach3: (int,L(int,int)) → L(int,(int,int))

attach3(n,l) = match l with | nil → nil
               | (x::xs) → (n,x)::attach3(n,xs);

append3: (L(int,(int,int)),L(int,(int,int))) → L(int,(int,int))

append3(l1,l2) = match l1 with | nil → l2
                 | (x::xs) → x::append3(xs,l2);

mTriples : (L(int),L(int),L(int)) → L(int,(int,int))

mTriples(l1,l2,l3) = match l1 with | nil → nil
    | (x::xs) → let triples = attach3(x,mPairs(l2,l3)) in
                append3 (triples,mTriples(xs,l2,l3));
```

With the heap-space metric, we can derive the following typings. I only mention the coefficients in the type annotations that are not zero.

$$\begin{array}{rcl}
\mathit{mPairs} & : & ((L(\mathit{int}),L(\mathit{int})),\{(1,1)\mapsto 6\}) \to (L(\mathit{int},\mathit{int}),\emptyset) \\
\mathit{mTriples} & : & ((L(\mathit{int}),L(\mathit{int}),L(\mathit{int})),\{(1,1,1)\mapsto 14\}) \to (L(\mathit{int},(\mathit{int},\mathit{int})),\emptyset) \\
\mathit{attach2} & : & ((\mathit{int},L(\mathit{int})),\{(*,1)\mapsto 3\}) \to (L(\mathit{int},\mathit{int}),\emptyset) \\
\mathit{attach3} & : & ((\mathit{int},L(\mathit{int},\mathit{int})),\{(*,1)\mapsto 4\}) \to (L(\mathit{int},(\mathit{int},\mathit{int})),\emptyset) \\
\mathit{append2} & : & ((L(\mathit{int},\mathit{int}),L(\mathit{int},\mathit{int})),\{(1,0)\mapsto 3\}) \to (L(\mathit{int},\mathit{int}),\emptyset) \\
\mathit{append3} & : & ((L(\mathit{int},(\mathit{int},\mathit{int})),L(\mathit{int},(\mathit{int},\mathit{int}))),\{(1,0)\mapsto 4\}) \to (L(\mathit{int},(\mathit{int},\mathit{int})),\emptyset)
\end{array}$$

For instance, the typing for *mPairs* states that the heap-space usage of the function is bound by $6nm$ where n is the length of the first input list and m is the length of the second input list. Similarly, the type of *mTriples* states that $14nmx$ bounds the heap-space usage of the function where n,m, and x are the lengths of the three arguments. Both bounds exactly match the actual worst-case behavior of the functions.

Compositionality

A primary feature of the multivariate analysis system is its high compositionality. I demonstrate this by using the functions that I defined in the previous subsection.

First, consider the function *pairs* from Section 5.5 again. The function *appPairs* uses *append* to concatenate two lists and then passes the result to *pairs*.

```
pairs: L(int) → L(int,int)

pairs(l) = match l with | nil  → nil
           | (x::xs) → append2(attach2(x,xs),pairs xs);

append: (L(int),L(int)) → L(int)

append(l1,l2) = match l1 with | nil → l2
                | (x::xs) → x::append(xs,l2);

appPairs : (L(int),L(int)) → L(int,int)

appPairs (l1,l2) = pairs(append(l1,l2));
```

With the heap-space metric, we obtain the following types.

$$\begin{array}{rcl}
\mathit{pairs} & : & (L(\mathit{int}),\{2\mapsto 6\}) \to (L(\mathit{int},\mathit{int}),\emptyset) \\
\mathit{append} & : & ((L(\mathit{int}),L(\mathit{int})),\{(1,0)\mapsto 2\}) \to (L(\mathit{int}),\emptyset) \\
\mathit{appPairs} & : & ((L(\mathit{int}),L(\mathit{int})),\left\{\begin{array}{ll}(0,2)\mapsto 6, & (1,1)\mapsto 6,\\ (2,0)\mapsto 6, & (1,0)\mapsto 2\end{array}\right\}) \to (L(\mathit{int},\mathit{int}),\emptyset)
\end{array}$$

The (optimal) computed heap-space for the function *appPairs* is therefore $3m^2+6mn-3m+3n^2-n$. The typing of *append* that is used for the call in the body of *appPairs*

passes potential from the arguments to the result without loss.

$$\mathit{append} \quad : \quad ((L(\mathit{int}), L(\mathit{int})), \left\{ \begin{array}{ll} (0,2) \mapsto 6, & (1,1) \mapsto 6, \\ (2,0) \mapsto 6, & (1,0) \mapsto 2 \end{array} \right\}) \to (L(\mathit{int}), \{2 \mapsto 6\})$$

Note that the potential $2n$, represented by the mapping $(1,0) \mapsto 2$, is used to pay for the resource consumption of *append*.

Similarly, we can also concatenate an arbitrary number of lists (i.e., the inner lists of a list of lists) and pass the result to *pairs*.

```
appendAll : L(L(int)) → L(int)

appendAll l = match l with | nil → nil
              | (l1::ls) → append(l1,appendAll ls);

appAllPairs : (L(L(int))) → L(int,int)

appAllPairs l = pairs(appendAll(l));
```

For the function *appAllPairs* we derive the following type which implies the heap-space bound $3n^2m^2 - nm$ if n is the length of the outer list and m is the maximal length of the inner lists.

$$\mathit{appAllPairs} \quad : \quad (L(L(\mathit{int})), \{[1,1] \mapsto 6,\ [1] \mapsto 2,\ [2] \mapsto 6\}) \to (L(\mathit{int}, \mathit{int}), \emptyset)$$

The last example in this subsection combines the function *appendAll* with *mPairs*.

```
appAllMPairs : (L(L(int)),L(L(int))) → L(int,int)

appAllMPairs (l1,l2) = mPairs(appendAll(l1),appendAll(l2));
```

With the heap-space metric, we derive the following type.

$$\mathit{appAllMPairs} \quad : \quad ((L(L(\mathit{int})), L(L(\mathit{int}))), \left\{ \begin{array}{c} ([1],0) \mapsto 2, \\ ([1],[1]) \mapsto 6, \\ (0,[1]) \mapsto 2 \end{array} \right\}) \to (L(\mathit{int}, \mathit{int}), \emptyset)$$

Eliminating Duplicates

A typical example that illustrates the advantages of resource polynomials is duplicate elimination in a list of lists. To find duplicates, we compare every element with every other element in the list.[1] Since the equality test for lists is linear in the lengths of the inputs, the running time of the program is $O(n \cdot m^2)$.

[1] There are more clever ways of eliminating duplicates but for the purpose of this example the naive algorithm is fine.

```
eq : (L(int),L(int)) → bool

eq (l1,l2) = match l1 with
    | nil     →  match l2 with | nil  → true
                    | (y::ys)  → false
    | (x::xs)  → match l2 with | nil  → false
                    | (y::ys)  → (x == y) and (eq (xs,ys));

nub : L(L(int)) → L(L(int))

nub l = match l with | nil  → nil
          | (x::xs)  → x::nub( remove(x,xs) );

remove : (L(int),L(L(int))) → L(L(int))

remove (x,l) = match l with | nil  → nil
               | (y::ys)  → if eq (x,y) then remove(x,ys)
                            else y::remove(x,ys);
```

The function *eq* implements an equality test for integer lists. The evaluation of the function call *remove(x,ℓ)* eliminates all elements from list ℓ that are equal to x and the function *nub* implements the actual duplicate elimination.

We derive the following types evaluation using the evaluation-step metric. The bound that is implied for *nub* is $6n^2m+9n^2-6nm+3n+3$.

$$
\begin{array}{rcl}
eq & : & ((L(int),L(int)),\{(0,0)\mapsto 5,\ (0,1)\mapsto 12\}) \to (bool,\emptyset) \\
remove & : & ((L(int),L(L(int))),\{(0,0)\mapsto 3,\ (0,[1])\mapsto 12,\ (0,1)\mapsto 18\}) \to (L(L(int)),\emptyset) \\
nub & : & (L(L(int)),\left\{\begin{array}{ll} 0\mapsto 3, & 1\mapsto 12, \\ 2\mapsto 18, & [1,0]\mapsto 12 \end{array}\right\}) \to (L(L(int)),\emptyset)
\end{array}
$$

One can always in principle find out how a particular system will behave just by running an experiment and watching what happens. But the great historical successes of theoretical science have typically revolved around finding mathematical formulas that instead directly allow one to predict the outcome.

STEPHEN WOLFRAM
A New Kind of Science (2002)

7 Experimental Evaluation

Klaus Aehlig and I implemented the multivariate analysis system from Chapter 6. In this chapter, I describe this prototype implementation as well as an experimental evaluation of the precision and the efficiency of the analysis.

In the prototype, we extended the syntax of Resource Aware ML to make it easier to use. Section 7.1 gives an overview of the implementation, defines the extended syntax, and explains how to use the prototype to analyze programs. In Section 7.2, I evaluate the performance of the analysis on a wide range of example programs. I compare the computed bounds with the measured worst-case behavior of the programs and report the time that is needed to compute the bounds. Finally, in Section 7.3, I present four case studies: lexicographic sorting of lists of lists, longest common subsequence via dynamic programming, split and sorting, and breadth-first traversal with matrix multiplication.

7.1 Prototype Implementation

Together with Klaus Aehlig, I implemented a prototype of Resource Aware ML. It is written in Haskell and consists of

- a parser (546 lines of code),
- a standard type checker (490 lines of code),
- an interpreter (333 lines of code),
- an LP solver interface (301 lines of code),
- and the multivariate analysis system from Chapter 6 (1637 lines of code).

Our emphasis in the prototype implementation was on correctness and extensibility rather than efficiency. That is why Haskell was a natural choice. In particular, the

comprehensive standard library of the Glasgow Haskell Compiler and the handy syntax for monadic computations proved helpful.

To implement the parser for RAML, we used the monadic parser combinator library Parsec[1]. The implementation of the type checker and the interpreter are straightforward. In the interpreter, we used the state monad to keep track of the resource consumption. We can track the resource consumption according to one or multiple metrics. We support two LP solvers in the implementation: Clp (Coin-or linear programming)[2] and lp_solve[3].

The main part of the implementation is the actual analyzer. In a nutshell, it works like a usual type checker that uses the state monad to store linear constraints while the individual syntactic constructs are type-checked. The main complexity arises from the manipulation of the rich indexes in the annotations. Altogether, we needed 4.5 man-month for the implementation of the analysis.

7.1.1 Extended Syntax

The RAML syntax in the prototype implementation differs from the syntax described in Chapter 3. For example, expressions are not restricted to let normal from. We also have more built-in operators and allow a destructive pattern matching *matchD* that deallocates the memory cell associated with the matched node of the data structure.

The following EBNF grammars describe the syntax of RAML programs. I skip the standard definitions of integer constants $n \in \mathbb{Z}$, variable identifiers $x \in$ VID, and function identifiers $f \in$ VID. Identifiers start with a letter and are built of numbers, letters, underscore, and prime.

A RAML program P consists of a (possibly empty) list of declarations followed by a main expression M. A declaration is either a type declaration D_T or a function definition D_F. There must be exactly one type declaration for every function definition. For every identifier, at most one type declaration and at most one function definition is allowed.

$$\begin{aligned} P &::= (D_T \mid D_F)^* M \\ D_F &::= f(x_1,\ldots,x_n) = e\,; \\ D_T &::= f : \tau_1 \to \tau_2 \\ M &::= \mathit{main} = e \end{aligned}$$

Data types τ are trees, lists, integers, Booleans, units, and tuples as defined by the following grammar.

$$\tau ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid (\tau_1,\ldots,\tau_n) \mid L(\tau) \mid T(\tau)$$

The next EBNF grammar defines expressions e. The reserved function *tick* is used in a special *tick metric* which is described later in this section. The argument q of *tick*

[1] `http://legacy.cs.uu.nl/daan/parsec.html`
[2] `https://projects.coin-or.org/Clp`
[3] `http://lpsolve.sourceforge.net`

denotes a floating point number. Also note that—in contrast to the rule in the grammar—the order of the patterns in the pattern matches can be arbitrary in the implementation. Instead of *cons(x,xs)* you can alternatively write *x::xs*, and instead of *nil* you can write *[]* in the patterns.

$$
\begin{aligned}
e ::= {} & () \mid \mathit{True} \mid \mathit{False} \mid n \mid x \mid \mathit{tick}(q) \\
& \mid e_1\ \mathit{binop}\ e_2 \mid \mathit{unop}\ e \mid f(e_1,\ldots,e_n) \\
& \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{if}\ e\ \mathit{then}\ e_t\ \mathit{else}\ e_f \\
& \mid [] \mid [e_1,\ldots,e_n] \mid (e_1,\ldots,e_n) \\
& \mid \mathit{nil} \mid \mathit{cons}(e_h,e_t) \mid \mathit{leaf} \mid \mathit{node}(e_0,e_1,e_2) \\
& \mid \mathit{match}\ e_1\ \mathit{with}\ (x_1,\ldots,x_n) \to e_2 \\
& \mid \mathit{let}\ (x_1,\ldots,x_n) = e_1\ \mathit{in}\ e_2 \\
& \mid \mathit{match}\ e\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \\
& \mid \mathit{match}\ e\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0,x_1,x_2) \to e_2 \\
& \mid \mathit{matchD}\ e\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \\
& \mid \mathit{matchD}\ e\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0,x_1,x_2) \to e_2
\end{aligned}
$$

$$
\begin{aligned}
\mathit{binop} &::= + \mid - \mid * \mid \mathit{mod} \mid \mathit{div} \mid \mathit{and} \mid \mathit{or} \mid :: \mid <= \mid >= \mid == \mid < \mid > \\
\mathit{unop} &::= + \mid - \mid \mathit{not}
\end{aligned}
$$

The expression *x::xs* is equivalent to the expression *cons(x,xs)*. An expression such as *[e1,e2,e3]* is another way of writing *e1::e2::e3::nil*. Furthermore, an expression such as *let (x1,x2,x3) = e1 in e2* is equivalent to *match e1 with (x1,x2,x3) → e2*.

Comments start with *(** and end with **)*. Like in SML there are no line comments.

Note that you have to provide a monomorphic type for every function in a program. The reason why we avoid polymorphic functions is that the resource consumption of a function depends on its type. For instance, the heap-space consumption of *append:*$(L(\mathit{int}),L(\mathit{int}))\to L(\mathit{int})$ might be $2n$ where n is the length of the first input list. In contrast, the heap-space consumption of *append:*$(L(\mathit{int},\mathit{int}),L(\mathit{int},\mathit{int}))\to L(\mathit{int},\mathit{int})$ might be $3n$. Alternatively, we could allow polymorphic functions and analyze a function for each concrete type it is used with in the program.

Destructive Pattern Match

A destructive pattern match—written using *matchD*—can be used to deallocate memory cells. For instance, let ℓ be a location in a heap H that contains a list element (v,ℓ') and let x be a variable that points to ℓ. Then the evaluation of the expression *matchD x with | nil → e1 | cons(x,xs) → e2* typically results in a heap H' that does not contain the location ℓ. This is provably true if there a no allocations during the evaluation of *e2*.

If memory cells are allocated during the evaluation of *e2* then the location ℓ may be used to store a new value in the resulting heap H'. So if a deallocated value is accessed during the evaluation of an expression then the behavior of the program is undefined. The following expression is an example that would cause a run-time error. The reason is that the deallocated list ℓ is accessed in the inner pattern match.

```
matchD l with
  | nil  → 0
  | (x::xs)  → match l with
                | nil  → 0
                | (y::ys)  → 1
```

If carefully used, destructive pattern matches can be used to define programs that use memory very efficiently. The following version of quick sort consumes for instance only $2n$ heap cells if n is the length of the input list. If we would replace the destructive pattern matches with the usual ones then we would have a quadratic heap-space consumption.

```
quicksortD : L(int)  → L(int)

quicksortD l = match l with | nil  → nil
               | (z::zs)  → let (xs,ys) = splitD (z,zs) in
                            appendD(quicksortD xs, z::(quicksortD ys));


splitD : (int,L(int))  → (L(int),L(int))

splitD(pivot,l) = matchD l with | nil  →  (nil,nil)
          | (x::xs)  → let (ls,rs) = splitD (pivot,xs) in
                       if x > pivot then (ls,x::rs) else (x::ls,rs);

appendD : (L(int),L(int))  → L(int)

appendD(l,ys) = matchD l with | nil  → ys
                | (x::xs)  → x::appendD(xs,ys);
```

Transformation to Let Normal Form

To perform the resource analysis as described in Chapter 6, we have to transform the unrestricted RAML expressions of the prototype implementation into expressions in let normal form as defined in Chapter 3. Furthermore, we make sharing of variables explicit to enable type inference (compare the discussion in Section 4.4).

The transformation to let normal from uses a special form of a let expression—called *freelet*—that does not consume any resources. For every expression that occurs in a position where only variables are allowed, we introduce a new variable with a *freelet*. For technical reasons we also introduce a new variable if the expression in such a *variable only position* in the source program is a variable itself. In this way, it becomes easy to preserve the resource cost of the source program because we know that all variables in

the *variable only positions* have been introduced by a *freelet*. Thus we never count the resource consumption K^{var} for the evaluation of variables in these places.

Consider for instance the expression *cons(cons(x,xs),nil)* which is not in let normal form. We would transform this expression as follows.

```
freelet x3 = x in
freelet x4 = xs in
freelet x1 = cons(x3,x4) in
freelet x2 = nil in
cons(x1,x2)
```

The resource cost we account for the evaluation of *cons(x3,x4)* is K^{cons} rather than $K^{\text{cons}} + K^{\text{var}} + K^{\text{var}}$ since we ensure that $2K^{\text{var}}$ has been accounted before in the free-let expressions.

To make sharing explicit, we add an additional syntactic construct to the expression each time a variable occurs multiple times. If a free variable x occurs twice in an expression e, we replace the first occurrence of x with x_1 and the second occurrence of x with x_2, obtaining a new expression e'. We then replace e with $\mathit{share}(x, x_1, x_2)\ \mathit{in}\ e'$. In this way, the sharing rule becomes a conventional syntax directed rule in the type inference. Consider for example the expression *let t = leaf in node(1,node(2,t,t),t)*. It is transformed by the prototype implementation as follows.

```
let t = leaf in
share (x9,x10) = t in
freelet x6 = 1 in
freelet x7 = share (x4,x5) = x9 in
             freelet x1 = 2 in
             freelet x2 = x4 in
             freelet x3 = x5 in
             node(x1,x2,x3) in
freelet x8 = x10 in
node(x6,x7,x8)
```

7.1.2 Usage

The prototype implementation is well documented and publically available. You can download the source code of the latest RAML version on the web site of the project[4]. It can be used to evaluate RAML programs and to compute resource bounds.

Resource Metrics

We included three resource metrics in the prototype and it is easy to define more by instantiating the resource constants.

The first metric that we included is the evaluation-step metric that counts the number of evaluation steps in the big-step operational semantics described in Section 3.3.

[4] `http://raml.tcs.ifi.lmu.de`

	Evaluation Steps	Heap Space	Ticks
K^{var}	1	0	0
K^{unit}	1	0	0
K^{int}	1	0	0
K^{bool}	1	0	0
K_1^{app}	1	0	0
K^{op}	1	0	0
K_1^{conT}	1	0	0
K_1^{conF}	1	0	0
K_1^{let}	1	0	0
K^{pair}	1	0	0
K_1^{matP}	1	0	0
K^{nil}	1	0	0
$K^{\text{cons}}(A)$	1	1+size(A)	0
K_1^{matN}	1	0	0
K_1^{matC}	1	0	0
K^{leaf}	1	0	0
$K^{\text{node}}(A)$	1	2+size(A)	0
K_1^{matTL}	1	0	0
K_1^{matTN}	1	0	0
K_1^{matND}	1	0	0
$K_1^{\text{matCD}}(A)$	1	-(1+size(A))	0
K_1^{matTLD}	1	0	0
$K_1^{\text{matTND}}(A)$	1	-(2+size(A))	0
$K^{\text{tick}}(q)$	1	0	q

Table 7.1: Resource Constants in the Implemented Metrics.

The second metric we included is the heap-space metric. The heap-space used by a node of a data structure depends on the type of the elements of the data structure. That is why we allow the resource constants to depend on the types of the respective expressions in the prototype. For instance, we do not simply have K^{cons} which defines the resource usage of a *cons* but rather $K^{\text{cons}}(A)$ where A is the type of the elements of the list. We define

$$\text{size}(A) \quad = \quad \begin{cases} n & \text{if } A = (A_1, \ldots, A_n) \\ 1 & \text{otherwise} \end{cases}$$

Then $K^{\text{cons}}(A) = \text{size}(A) + 1$ is the number of memory cells that are used to store a node of a list of type $L(A)$. Similarly, $-K_1^{\text{matCD}}(A) = \text{size}(A) + 1$ memory cells become available in a destructive pattern match.

Since the types $L(A)$ are known at compile time, it makes no difference for the analysis whether the constants depend on data types. I excluded this dependency from the type rules to simplify the type systems in the previous chapters. Moreover, the values of the constants can depend on everything that is statically known about the program, not only the types.

The third metric that we implemented measures the number of ticks that occur in an evaluation. To this end, a programmer can insert expressions such as *tick(3.5)* or *tick(−4)* into the code. Every time the expression *tick(q)* is evaluated, q resources are consumed, or $-q$ resources are restituted if q is negative.

The tick metric can be used to manually model specific resource metrics and is also helpful for testing.

Table 7.1 on page 144 shows the values of the constants in the evaluation-step, heap-space, and ticks metric. Constants that are zero in all metrics are not mentioned. The constant K^{op} stands for the constants of all operators *op*. These constants (K^+, K^-, $K^{\leq}$, etc.) are identical in all three metrics. The constants $K^{\text{cons}}(A)$ and $K^{\text{node}}(A)$ depend on the type $L(A)$ or $T(A)$ of the respective data structure; $K^{\text{tick}}(q)$ depends on the lengths q of the tick. K_1^{matND}, $K_1^{\text{matCD}}(A)$, K_1^{matTLD}, and $K_1^{\text{matTND}}(A)$ are the constants for the destructive pattern matches.

Compilation

To compile the prototype you need the Glasgow Haskell Compiler (GHC)[5]. We successfully compiled it with GHC 6.8, GHC 6.10, GHC 6.12, and GHC 7. To produce the binary *raml* you can execute the following command in the directory of the source code.

```
> ghc -O --make Main.hs -o raml
```

Alternatively you can use RAML interactively with ghci as follows.

```
> ghci
GHCi, version 6.10.4: http://www.haskell.org/ghc/ :? for help
```

[5] `http://haskell.org/ghc`

```
Prelude> :l Main.hs
*Main> analyseFile "examples/sorting.raml" heapSpace 2
```

The function *analyseFile* is documented in detail in the file *Main.hs*.

The prototype uses the LP solver Clp by default. You can also use lp_solve. However, Clp seems to be much faster. The RAML implementation expects that the respective binaries, namely *clp* or *lp_solve*, are in the path.

Web Interface

On the RAML website[6] you can download the source code of the prototype or use it directly on the web. Figure 7.1 shows the web interface of the prototype implementation. You see two text fields.

In the first text field you can provide an input program or select an example file from the drop-down menu above the text field. Click on the button *load file* to load the example into the field. The second text field contains the output of the RAML prototype. You can use it to compute resource bounds for your program or to evaluate the main expression.

To evaluate the main expression click on the button *Evaluate Main* which you find on the right-hand side between the two text fields. The result of a successful evaluation is the value of the main expression as well as the number of heap cells, the number of evaluation steps, and the number of ticks that have been used to evaluate the main expression. Note that the evaluation on the server will be terminated after 2 minutes.

To analyze the RAML program in the input text field, click on the button *Infer Types* that is located on the left-hand side between the input and the output text fields. The output is either a list of annotated types—one for each function in the program including the main expression—or an error message. If the program is type correct then the only error that should occur is the message *the linear program is infeasible*. It indicates that the LP solver finished unsuccessful.

Next to the *Infer Types* button you can choose different options:

1. The resource metric that you want to use in the analysis. It can either be heap-space consumption, evaluation steps or ticks.

2. An upper bound on the maximal degree that can occur in the resource bounds. If the degree is too low then the analysis reports that *the linear program is infeasible*. The only problem with a too high degree is that the analysis will take longer.

3. Whether you would like to have a verbose output. The verbose output shows for instance the function definitions in let normal form.

[6] `http://raml.tcs.ifi.lmu.de`

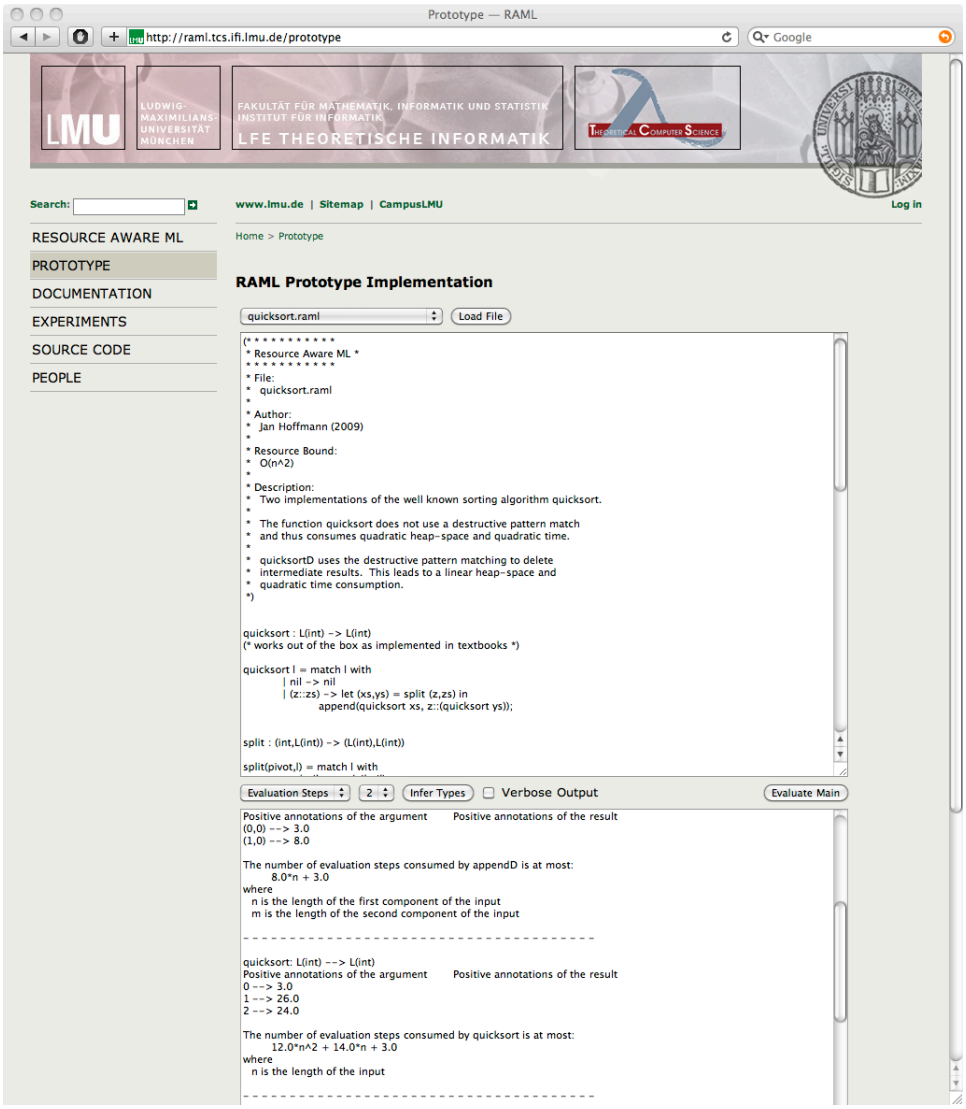

Figure 7.1: The web interface of the RAML prototype implementation at `http://raml.tcs.ifi.lmu.de`. You can analyze or evaluate predefined examples or own example programs directly on the web.

Command Line Interface

On the command line, the same options as in the web form are available. The general pattern is the following.

```
raml ACTION OPTIONS1 FILE OPTIONS2
```

At the position *OPTIONS1* are the options for the selected action and *OPTIONS2* contains global options. *FILE* is the path to the RAML program.

The actions that are implemented are *analyse* and *evaluate*. The options for *analyse* are *heap−space*, *eval−steps*, or *ticks* as well as a positive integer *n* that specifies the maximal degree of the bounds. There are no local options for the action *evaluate*.

The following global options are available.

- *−−verbose*
- *−−tempdir=TEMPDIR*
- *−−time*
- *−−lp_solve*
- *−−clp*

As you might guess, *TEMPDIR* defines which directory is used for temporary files. The temporary files will be deleted after the execution.

The option *−−time* is used to measure the run time of the prototype. It is not very exact since it just subtracts the system time at the start of the execution from the system time at the end of the execution.

The option *−−lp_solve* enables the use of the LP solver lp_solve. The binary lp_solve is then assumed to be in the path. Similarly, *−−clp* (default) enables the use of the LP solver Clp. It is assumed that the binary clp is in the path.

Below are some typical usage examples.

```
raml analyse heap-space 2 quicksort.raml

raml evaluate quicksort.raml

raml analyse eval-steps 2 quicksort.raml --lp_solve --time

raml analyse ticks 2 quicksort.raml --clp --time
```

Output of the Analysis

Below is the result of the evaluation of the main expression in the file *quicksort.raml*. It contains the type of the main expression, the value of the main expression, and the resource usage according to the heap-space, evaluation-step, and ticks metrics.

```
> raml evaluate quicksort.raml --time
main : L(int)

main = [0,1,2,3,4,5,6,7,8,9]

Resource Usage:
     112.0 heap cells
     807.0 evaluation steps
     0.0 ticks

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Run time of the prototype: 0.006705s
```

The resource analysis of a RAML program computes a resource bound for every function in the program. The output for the function *quicksort* and the evaluation step metric is for instance the following.

```
> raml analyse eval-steps 3 quicksort.raml
quicksort: L(int) → L(int)
Positive annotations of the argument    Pos. annos. of the result
0 → 3.0
1 → 26.0
2 → 24.0

The number of evaluation steps consumed by quicksort is at most:
         12.0*n^2 + 14.0*n + 3.0
where
   n is the length of the input
```

It contains the type of the function and the non-zero potential annotations of the argument type and the result type. Finally, the resource annotations are converted into a usual polynomial at the convenience of the user. This transformation is straightforward.

7.2 Experiments

We performed experiments to evaluate the performance and accuracy of the prototype.

We used the LP solver Clp[7] in the experiments which seems to be much faster than lp_solve. Further speed-up would be possible by using a commercial LP solver and by optimizing our Haskell implementation. However, we decided that accessibility and maintainability take precedence over performance in the prototype implementation.

More profound improvement is possible by finding a suitable heuristic that is in between the (maybe too) flexible multivariate analysis and the inference for the univariate system, which also works efficiently with high maximal degree for large programs.

[7] Clp version 1.14. See `https://projects.coin-or.org/Clp`

For example, we could set certain coefficients q_i to zero before even generating the constraints. Alternatively, we could limit the number of different types for each function.

Table 7.2 on page 151 shows a compilation of the computed evaluation-step bounds for several example functions. Table 7.3 on page 152 contains heap-space bounds. The tables list the computed resource polynomials, the simplified bounds, the actual asymptotic worst-case behavior of the functions, and the run time of the analysis on a 3.6 GHz Intel Core 2 Duo iMac with 4 GB RAM. The run times are between 2.00 s and 0.02 s and depend on both the maximal degree that is needed and the size of the program.

Function names that end with a D indicate that the destructive pattern match was used in the program. The variables that appear in the bounds are defined as follows.

- n is the size of the first argument
- m_i are the sizes of the elements of the first argument
- x is the size of the second argument
- y_i are the sizes of the elements of the second
- $m = \max_{1 \le i \le n} m_i$
- $y = \max_{1 \le i \le x} y_i$

Most bounds are asymptotically tight. Exceptions are the evaluation-step bound for *mergesort* and the heap-space bounds for *matrixmultT* and *matrixmultAcc*.

To determine the precision of the constant factors, we manually identified worst-case inputs for the functions and compared the computed bounds with the measured resource consumption. Our experiments show that the constant factors in the computed bounds are generally quite tight and even match the measured worst-case running times of many functions. I briefly discuss the results of the experiments for every function.

Quick Sort The code of *quicksort* is given in Section 5.5.2 and you can find the code of *quicksortD* in Section 7.1. The worst-case resource behavior of quick sort emerges if the input list is reversely sorted. Figure 7.2 compares the computed bound with the measured number of evaluation steps that *quicksort* needed for these lists. Our experiments show that both the heap-space and the evaluation-step bound match exactly the measured worst-case behavior.

Insertion Sort You find the code of *insertionsort* in Section 5.5.2. The destructive version *insertionsortD* replaces the pattern match in the function *insert* with a destructive one. Our experiments show that the constants in both bounds are optimal.

We also implemented insertion sort for lists of lists. The computed bounds are also tight. The program code and more detail can be found in Section 7.3.

Function / Type	Computed Evaluation-Step Bound / Simplified Computed Bound	Asymptotic Behavior	Run Time
quicksort : $L(int)\to L(int)$	$24\binom{n}{2}+26n+3$ $12n^2+14n+3$	$O(n^2)$	0.08 s
insertionsort : $L(int)\to L(int)$	$12\binom{n}{2}+12n+3$ $6n^2+6n+3$	$O(n^2)$	0.05 s
mergesort : $L(int)\to L(int)$	$73.3\binom{n}{2}+7.3n+3$ $36.6n^2-29.3n+3$	$O(n\log n)$	0.07 s
pairs : $L(int)\to L(int,int)$	$18\binom{n}{2}+16n+3$ $9n^2+7n+3$	$O(n^2)$	0.08 s
triples : $L(int)\to L(int,int,int)$	$36\binom{n}{3}+16\binom{n}{2}+20n+3$ $6n^3-10n^2+24n+3$	$O(n^3)$	0.43 s
quadruples : $L(int)\to L(int,int,int,int)$	$54\binom{n}{4}+16\binom{n}{3}+20\binom{n}{2}+20n+3$ $2.2n^4-10.8n^3+26.7n^2+1.8n+3$	$O(n^4)$	2.00 s
isortlist : $L(L(int))\to L(L(int))$	$\sum_{1\le i<j\le n}16m_i+16\binom{n}{2}+12n+3$ $8n^2m+8n^2-8nm+4n+3$	$O(n^2m)$	0.19 s
nub : $L(L(int))\to L(L(int))$	$\sum_{1\le i<j\le n}12m_i+18\binom{n}{2}+12n+3$ $6n^2m+9n^2-6nm+3n+3$	$O(n^2m)$	0.21 s
transpose : $L(L(int))\to L(L(int))$	$\sum_{1\le i\le n}32m_i+2n+13$ $32nm+2n+13$	$O(nm)$	0.10 s
matrixmultT : $(L(L(int)),L(L(int)))\to L(L(int))$	$(\sum_{1\le i\le x}y_i)(32+28n)+14n+2x+21$ $28xyn+32xy+2x+14n+21$	$O(nxy)$	0.70 s
matrixmultAcc : $(L(L(int)),L(L(int)))\to L(L(int))$	$\sum_{1\le i\le n}15m_i+\sum_{1\le i\le x}15ny_i+15n+3$ $15xyn+16nm+15n+3$	$O(nxy)$	0.41 s
dyad : $(L(int),L(int))\to L(L(int))$	$10nx+14n+3$ $10nx+14n+3$	$O(nx)$	0.02 s
lcs : $(L(int),L(int))\to int$	$39nx+6x+21n+19$ $39nx+6x+21n+19$	$O(nx)$	0.10 s
subtrees : $T(int)\to L(T(int))$	$8\binom{n}{2}+23n+3$ $4n^2+19n+3$	$O(n^2)$	0.06 s
eratos : $L(int)\to L(int)$	$16\binom{n}{2}+12n+3$ $8n^2+4n+3$	$O(n^2)$	0.04 s
splitandsort : $L(int,int)\to L(L(int),int)$	$42\binom{n}{2}+58n+9$ $21n^2+37n+9$	$O(n^2)$	0.64 s

Table 7.2: Computed Evaluation-Step Bounds.

Function / Type	Computed Heap-Space Bound / Simplified Computed Bound	Asymptotic Behavior	Run Time
quicksortD : $L(int)\to L(int)$	$2n$ $2n$	$O(n)$	0.07 s
insertionsortD : $L(int)\to L(int)$	$2n$ $2n$	$O(n)$	0.04 s
mergesortD : $L(int)\to L(int)$	0 0	0	0.05 s
pairs : $L(int)\to L(int,int)$	$6\binom{n}{2}$ $3n^2-3n$	$O(n^2)$	0.05 s
triples : $L(int)\to L(int,int,int)$	$14\binom{n}{3}$ $2.3n^3-n^2+24n+3$	$O(n^3)$	0.36 s
quadruples : $L(int)\to L(int,int,int,int)$	$24\binom{n}{4}$ $n^4-6n^3+11n^2-6n$	$O(n^4)$	1.83 s
isortlist : $L(L(int))\to L(L(int))$	$2\binom{n}{2}+2n$ n^2+n	$O(n^2)$	0.11 s
nubD : $L(L(int))\to L(L(int))$	$2n$ $2n$	$O(n)$	0.14 s
transpose : $L(L(int))\to L(L(int))$	$\sum_{1\le i\le n}8m_i$ $8nm$	$O(nm)$	0.98 s
matrixmultT : $(L(L(int)),L(L(int)))\to L(L(int))$	$(\sum_{1\le i\le x}y_i)(8+2n)+2n$ $2xyn+8xy+2n$	$O(nx)$	0.56 s
matrixmultAcc : $(L(L(int)),L(L(int)))\to L(L(int))$	$\sum_{1\le i\le x}2ny_i+2n$ $2xyn+2n$	$O(nx)$	0.37 s
dyad : $(L(int),L(int))\to L(L(int))$	$2nx+2n$ $2nx+2n$	$O(nx)$	0.03 s
lcs : $(L(int),L(int))\to int$	$2nx+2x+4n+2$ $2nx+2x+4n+2$	$O(nx)$	0.14 s
subtrees : $T(int)\to L(T(int))$	$2\binom{n}{2}+5n$ n^2+4n	$O(n^2)$	0.05 s
eratos : $L(int)\to L(int)$	$2n$ $2n$	$O(n)$	0.04 s
splitandsort : $L(int,int)\to L(L(int),int)$	$7\binom{n}{2}+10n$ $3.5n^2+6.5n$	$O(n^2)$	0.63 s

Table 7.3: Computed Heap-Space Bounds.

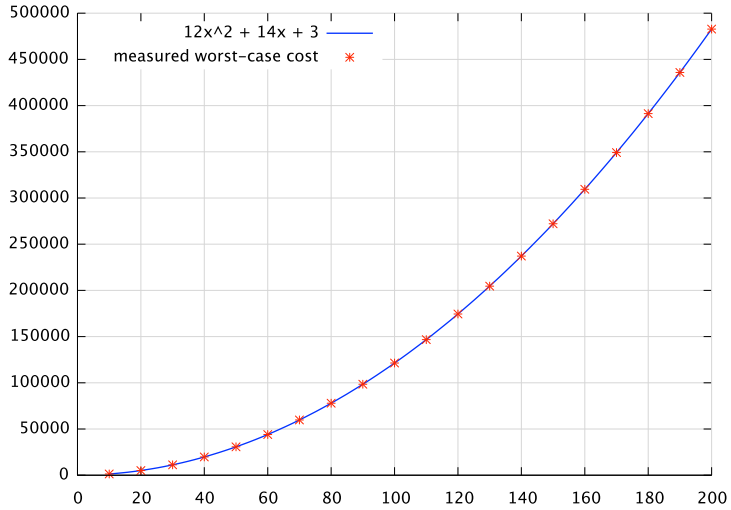

Figure 7.2: The computed evaluation-step bound (line) compared to the actual number of evaluation-steps for reversely sorted list of various sizes (crosses) used by *quicksort*. The x-axis represents the length of the list. The computed bound matches exactly the worst-case costs.

Merge Sort The function *mergesort* is defined in Section 5.5.2. In *mergesortD* we replaced all pattern matches with destructive ones and thus obtain a version of merge sort that deallocates the input list. It does not need any additional heap space. Since the run time of merge sort is $O(n \log n)$, our analysis system cannot represent a tight evaluation step bound. However, it computes a quadratic bound.

Tuples The functions *pairs* and *triples* are described in Section 5.5.1. The function *quadruples* is similar. All heap-space and evaluation-step bounds match exactly the worst-case behavior of the functions. Note the negative factors and fractional numbers in the simplified bounds in contrast to the even factors in the binomial representation.

Duplicates The functions *nub* and *nubD* remove duplicates from a list of lists. The definition of *nub* is given in Section 6.6. In *nubD*, the function *remove* is implemented with a destructive pattern match. Our experiments indicate that both bounds match exactly the actual worst-case behavior.

Matrix Multiplication We implemented two versions of matrix multiplication for matrices that are represented as lists of integers. The function *matrixmultT* transposes the second matrix before the actual multiplication. The function *matrixmultAcc* uses an accumulator to perform the multiplication without transposing the second matrix.

Both evaluation-step bounds are asymptotically tight. Figure 7.3 shows a compar-

ison of the computed bounds with the measured worst-cast evaluation steps. The bound for *matrixmultAcc* is almost tight while the bound for *matrixmultT* is a bit off. The reason is that the analysis cannot assume that all inner lists of a matrix have the same length. As a result, there is a loss of potential when transposing matrices.

The heap-space bounds are not asymptotically tight. This shows a general limitation of the analysis system. Consider for instance the function *matrixmultAcc* and the computed bound $\sum_{1\leq i\leq x} 2ny_i+2n$, where n is the length of the outer list in the first component and y_i is the length of the ith inner list in the second component. A tight bound would be $2ny_1+2n$. Such a bound cannot be expressed in our system.

Dyadic Product The function *dyad* is described in section Section 2.2.2. Our experiments indicate that both computed bounds exactly match the worst-cast behavior.

Longest Common Subsequence The function *lcs* computes the length of the longest common sequence of two sequences that are represented as lists of lists. Both computed bounds are asymptotically tight. Our experiments indicate that the constants in the heap-space bound are optimal. Figure 7.4 shows that the evaluation step bound is close to the optimal one.

The function is described in detail in Section 7.3.

Subtrees The function *subtrees* computes a list of all subtrees for a given tree. Both the heap-space and the evaluation-step bound match exactly the measured worst-case behavior.

Sieve of Eratosthenes The sieve of Eratosthenes is a classic algorithm that computes the list of primes that are smaller than a given number. The code of *eratos* can be found in Section 2.2.2. Our experiments show that both bounds match exactly the measured worst-case behavior of the function.

Note that the worst-case behavior emerges when the input of the function is a list of primes rather than a list *[2,3,...,n]* of succeeding natural numbers.

Split and Sort The function *splitandsort* consists of two sub-functions. The input is a list of values and keys. Firstly, the values are split according to their keys. Secondly, the arising lists of values are sorted. Interestingly, the prototype computes asymptotically tight, quadratic bounds for *splitandsort*.

In Section 7.3, I define the function and explain why it may be surprising that an automatic analysis finds a quadratic rather than a cubic bound.

The source code and the experimental validation of all examples are available online[8].

[8] `http://raml.tcs.ifi.lmu.de`

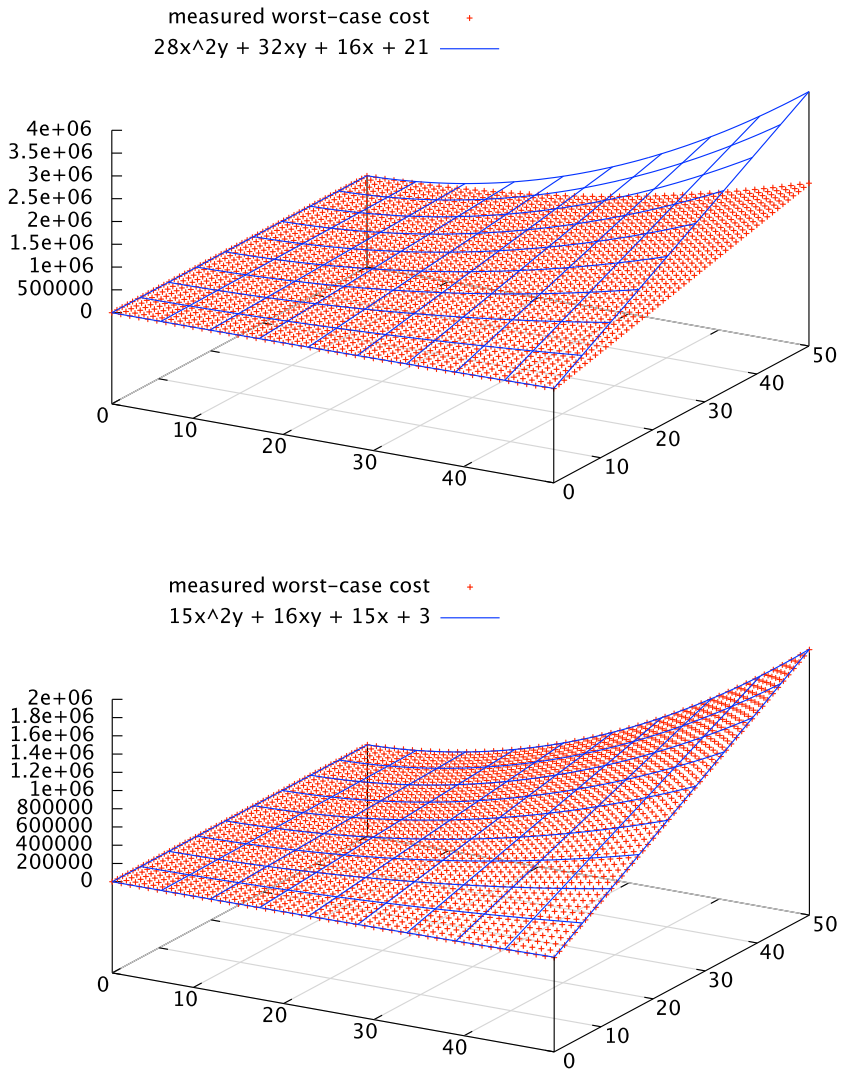

Figure 7.3: The computed evaluation-step bound (lines) compared to the actual worst-case number of evaluation-steps for sample inputs of various sizes (crosses) used by *matrixmultT* (at the top) and *matrixmultAcc* (at the bottom). The x-axis represents the dimension $x \times x$ of the (quadratic) matrix in the first argument. The y-axis represents the second component of the dimension $x \times y$ of the matrix in the second argument. The integers in the matrices do not influence the running times of the functions.

7.3 Case Studies

I present the experimental evaluation of four more involved programs in more detail in this section. To begin with, I demonstrate the compositionality of the analysis by implementing insertion sort for lists of lists. I then show that you can analyze a natural implementation of an algorithm that computes the length of the longest common subsequence of two sequences. The last two examples—split and sort, and breadth-first traversal with matrix multiplication—illustrate the advantages of the amortized method.

7.3.1 Lexicographic Sorting of Lists of Lists

The following RAML code implements the well-known sorting algorithm insertion sort that lexicographically sorts lists of lists. To lexicographically compare two lists one needs linear time in length of the shorter list. Since insertion sort does quadratically many comparisons in the worst case it has a running time of $O(n^2 m)$ if n is the length of the outer list and m is the maximal length of the inner lists.

```
leq (l1,l2) = match l1 with | nil → true
   | (x::xs) → match l2 with | nil → false
        | (y::ys) → (x<y) or ((x == y) and leq (xs,ys));


insert (x,l) = match l with | nil → [x]
               | (y::ys) → if leq(x,y) then x::y::ys
                           else y::insert(x,ys);

isortlist l = match l with | nil → nil
                | (x::xs) → insert (x,isortlist xs);
```

Below is the output of the analysis for the function *isortlist* when instantiated to bound the number of needed evaluation steps. The computation needs less than a second on typical desktop computers.

```
isortlist: L(L(int)) → L(L(int))
Positive annotations of the argument
0     →  3.0         2      →  16.0
1     →  12.0        [1,0]  →  16.0

The number of evaluation steps consumed by isortlist is at most:
         8.0*n^2*m + 8.0*n^2 - 8.0*n*m + 4.0*n + 3.0
where
   n is the length of the input
   m is the length of the elements of the input
```

The more precise bound implicit in the positive annotations of the argument is presented in mathematical notation in Table 7.2 on page 151.

We manually identified inputs for which the worst-case behavior of *isortlist* emerges (namely reversely sorted lists with similar inner lists). Then we measured the needed

evaluation steps and compared the results to our computed bound. Our experiments show that the computed bound exactly matches the actual worst-case behavior.

7.3.2 Longest Common Subsequence

An example of dynamic programming that can be found in many textbooks is the computation of (the length of) the longest common subsequence (LCS) of two given lists (sequences). If the sequences $a_1,\ldots,a_n$ and $b_1,\ldots,b_m$ are given then an $n \times m$ matrix (here a list of lists) A is successively filled such that $A(i,j)$ contains the length of the LCS of $a_1,\ldots,a_i$ and $b_1,\ldots,b_j$. The following recursion is used in the computation.

$$A(i,j)=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ A(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j \\ \max(A(i,j-1),A(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j \end{cases}$$

The run time of the algorithm is thus $O(nm)$. Below is the RAML implementation of the algorithm.

```
lcs(l1,l2) =
    let m = lcstable(l1,l2) in
    match m with | nil → 0
      | (l1::_) → match l1 with | nil → 0
                    | (len::_) → len;

lcstable (l1,l2) =
    match l1 with | nil → [firstline l2]
      | (x::xs) → let m = lcstable (xs,l2) in
                  match m with | nil → nil
                    | (l::ls) → (newline (x,l,l2))::l::ls;

newline (y,lastline,l) =
   match l with | nil → nil
   | (x::xs) → match lastline with | nil → nil
        | (belowVal::lastline') →
            let nl = newline(y,lastline',xs) in
            let rightVal = right nl in
            let diagVal = right lastline' in
            let elem = if x == y then diagVal+1
                       else max(belowVal,rightVal)
            in elem::nl;

firstline(l) = match l with | nil → nil
                 | (x::xs) → 0::firstline xs;

right l = match l with | nil → 0 | (x::xs) → x;
```

The analysis of the program takes less than a second on a usual desktop computer and produces the following output for the function *lcs*.

```
lcs: (L(int),L(int)) → int
Positive annotations of the argument
(0,0) → 19.0       (1,0) → 21.0
(0,1) → 6.0        (1,1) → 39.0

The number of evaluation steps consumed by lcs is at
most:     39.0*m*n + 6.0*m + 21.0*n + 19.0
where
   n is the length of the first component of the input
   m is the length of the second component of the input
```

Figure 7.4 shows that the computed bound is close to the measured number of evaluation steps needed. In the case of *lcs*, the run time exclusively depends on the lengths of the input lists.

7.3.3 Split and Sort

Multivariate resource polynomials take into account the individual sizes of all inner data structures. In contrast to the approximation of, say, the lengths of inner lists by their maximal lengths, this approach leads to tight bounds when composing functions.

The function *splitAndSort* demonstrates this advantage.

```
splitAndSort : L(int,int) → L(L(int),int)

splitAndSort l = sortAll (split l);
```

An input to the function is a list such as ℓ = *[(1,0),(2,1),(3,0),(4,0),(5,1)]* that contains integer pairs of the form *(value,key)*. The list is processed in two steps. At first, the function *split* partitions the values according to their keys. For instance we have *split*(ℓ) = *[([2,5],1),([1,3,4],0)]*. In the second step—implemented by *sortAll*—the inner lists are sorted with quick sort.

The function *split* is implemented as follows.

```
split : L(int,int) → L(L(int),int)

split l = match l with | nil → nil
          | (x::xs) → insert( x, split xs);

insert : ((int,int),L(L(int),int)) → L(L(int),int)

insert (x,l) = let (valX,keyX) = x in
   match l with | nil → [([valX],keyX)]
     | (l1::ls) → let (vals1,key1) = l1 in
                  if key1 == keyX then (valX::vals1,key1)::ls
                     else (vals1,key1)::insert(x,ls);
```

The prototype computes the tight quadratic bound $9n^2 + 9n + 3$ on the number of evaluation steps *split* needs for inputs of length n.

The second part of *splitAndSort* is implemented by the function *sortAll*. It uses the sorting algorithm quick sort to sort all the inner lists of its input. The function can be implemented as follows.

```
sortAll : L(L(int),int) → L(L(int),int)

sortAll l = match l with | nil → nil
            | (x::xs) → let (vals,key) = x in
                        (quicksort vals,key)::sortAll(xs);

quicksort : L(int) → L(int)

quicksort l = match l with | nil → nil
              | (z::zs) → let (xs,ys) = splitqs (z,zs) in
                          append(quicksort xs, z::(quicksort ys));

splitqs : (int,L(int)) → (L(int),L(int))

splitqs(pivot,l) = match l with | nil → (nil,nil)
     | (x::xs) → let (ls,rs) = splitqs (pivot,xs) in
                 if x > pivot then (ls,x::rs) else (x::ls,rs);

append : (L(int),L(int)) → L(int)

append(l,ys) = match l with | nil → ys
                | (x::xs) → x::append(xs,ys);
```

The simplified computed evaluation-step bound for *sortAll* is $12nm^2 + 14nm + 14n + 3$ where n is the length of the outer list and m is the maximal length of the inner lists.

Now consider the composed function *splitAndSort* again and assume we would like to derive a bound for the function using the simplified bounds for *sortAll* and *split*. This would lead to a cubic bound for *splitAndSort* rather than a tight quadratic bound. The reason is that—in the evaluation of *splitAndSort*(ℓ)— both n and m can only be bounded by $|\ell|$ the bound $12nm^2 + 14nm + 14n + 3$ for *sortAll*.

In contrast, the use of the multivariate resource polynomials enables the inference of a quadratic bound for *splitAndSort*. For one thing, the actual computed bound $\sum_{1\leq i\leq n}\left(24\binom{m_i}{2} + 26m_i\right) + 14n + 3$ for *sortAll* incorporates the individual lengths m_i of the inner lists. For another thing, the type annotation for the function *split* passes potential from the argument of the function to the inner lists of the result without losses.

As a result, the prototype computes the asymptotically tight, quadratic bound $21n^2 + 37n + 9$ for the function *splitAndSort*. The constant factors are however not tight. The reason is that the worst-case behavior of the function *split* emerges if all values in the input have different keys but the worst-case of *sortAll* emerges if all values in the input have the same key. The analysis cannot infer that the worst-case behaviors are mutually exclusive but assumes that they can occur for the same input.

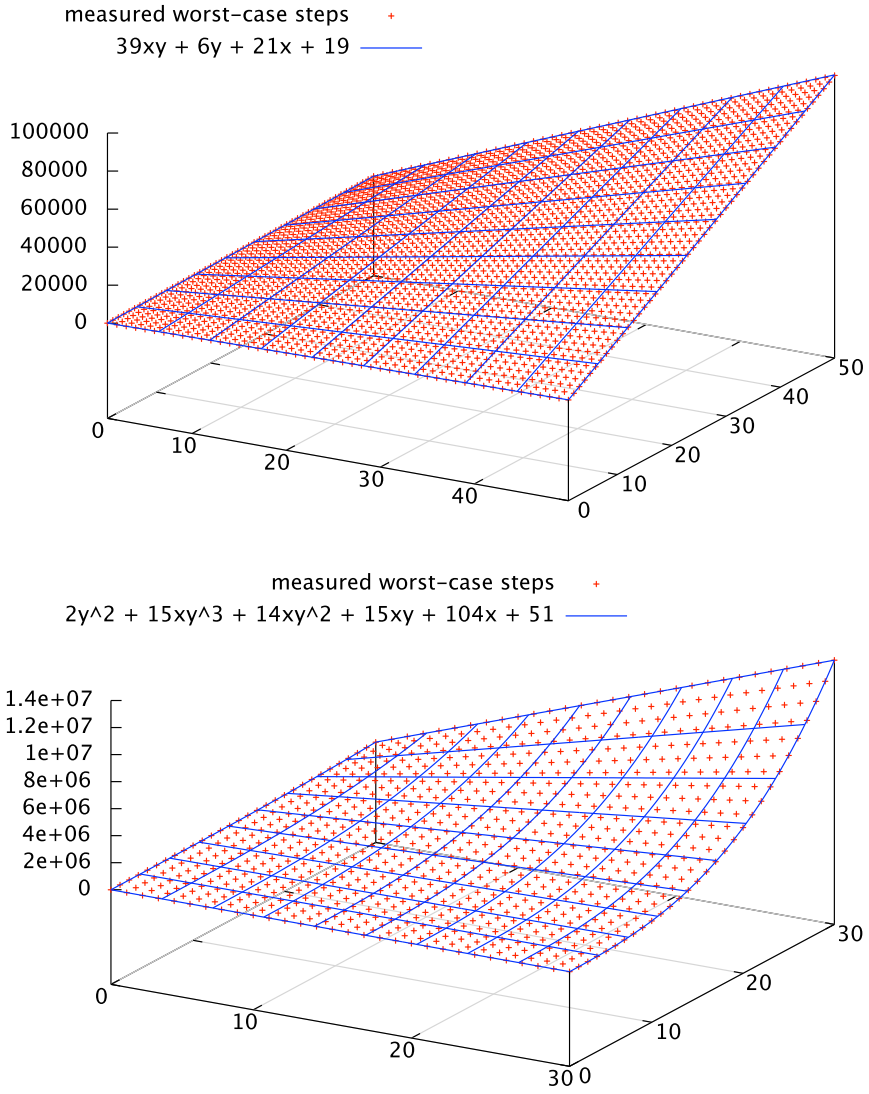

Figure 7.4: The computed evaluation-step bound (lines) compared to the actual worst-case number of evaluation-steps for sample inputs of various sizes (crosses) used by *lcs* (at the top) and *bftMult* (at the bottom). In the first plot, the x-axis represents the length of the first list and the y-axis represents the length of the second list in the arguments of *lcs*. In the second plot, x denotes the number of nodes in the tree and $y \times y$ is the dimension of the matrices in the input of *bftMult*. In both cases, the computed bounds are close to the optimal ones.

7.3.4 Breadth-First Traversal with Matrix Multiplication

A classic example that motivates amortized analysis is a functional queue. A *queue* is a first-in-first-out data structure with the operations *enqueue* and *dequeue*. The operation *enqueue*(a) adds a new element a to the queue. The operation *dequeue*() removes the oldest element from the queue. A queue is often implemented with two lists L_{in} and L_{out} that function as stacks. To enqueue a new element in the queue, you simply attach it to the beginning of L_{in}. To dequeue an element from the queue, you detach the first element from L_{out}. If L_{out} is empty then you transfer the elements from L_{in} to L_{out}; thereby reversing the order of the elements.

Later in this example we shall store trees of matrices (lists of lists of integers) in our queue. So the two lists of queue have type $L(T(L(L(int))))$ in the following RAML implementation.

```
dequeue :  (L(T(L(L(int)))),L(T(L(L(int)))))
        →  (L(T(L(L(int)))),(L(T(L(L(int)))),L(T(L(L(int))))))

dequeue (outq,inq) =
   match outq with
     | nil  → match reverse inq with | nil  → ([],([],[]))
                | t::ts  → ([t],(ts,[]))
     | t::ts  → ([t],(ts,inq));

enqueue :  (T(L(L(int))),(L(T(L(L(int)))),L(T(L(L(int))))))
        →  (L(T(L(L(int)))),L(T(L(L(int)))))

enqueue (t,queue) = let (outq,inq) = queue in
                    (outq,t::inq);

appendreverse : (L(T(L(L(int)))),L(T(L(L(int))))) → L(T(L(L(int))))

appendreverse (toreverse,sofar) =
   match toreverse with
     | nil  → sofar
     | (a::as)  → appendreverse(as,a::sofar);

reverse: L(T(L(L(int)))) → L(T(L(L(int))))

reverse xs = appendreverse(xs,[]);
```

The prototype implementation infers precise linear bounds for the above functions. The evaluation-step bound for *reverse* is for instance $8n+7$ where n is the length of the input list.

The point of this example is the use of a queue in a breadth-first traversal of a binary tree. Suppose we have given a binary tree of matrices and we want to multiply the matrices in breadth first-order. The matrices are represented as lists of lists of integers and can have different dimensions. However, we assume that the dimensions fit if the

matrices are multiplied in breadth-first order. Before we implement the actual breadth-first traversal, we first implement matrix multiplication as follows. We use accumulation to avoid transposing matrices before the multiplication.

```
matrixMult : (L(L(int)),L(L(int))) → L(L(int))

matrixMult (m1,m2) =
   match m1 with | [] → []
      | (l::ls) → (computeLine(l,m2,[])) :: matrixMult(ls,m2);

computeLine : (L(int),L(L(int)),L(int)) → L(int)

computeLine (line,m,acc) =
   match line with | [] → acc
      | (x::xs) → match m with [] → []
                  | (l::ls) → computeLine(xs,ls,lineMult(x,l,acc));

lineMult : (int,L(int),L(int)) → L(int)

lineMult (n,l1,l2) =
   match l1 with | [] → []
     | (x::xs) → match l2 with | [] → x*n::lineMult(n,xs,[])
                 | (y::ys) → x*n + y :: lineMult(n,xs,ys);
```

The computed evaluation step bound for *matrixMult* is $15mkn + 16nm + 15n + 3$ if the first matrix is of dimension $n \times m$ and the second matrix is of dimension $m \times k$.[9]

Eventually, we implement the breadth-first traversal with matrix multiplication as follows.

```
bftMult : (T(L(L(int))),L(L(int))) → L(L(int))

bftMult (t,acc) = bftMult'(([t],[]),acc);

bftMult' : ((L(T(L(L(int)))),L(T(L(L(int))))),L(L(int))) → L(L(int))

bftMult'(queue,acc) =
  let (elem,queue) = dequeue queue in
  match elem with | nil → acc
    | t::_ → match t with | leaf → bftMult'(queue,acc)
                | node(y,t1,t2) →
                    let queue' = enqueue(t2,enqueue(t1,queue)) in
                    bftMult'(queue',matrixMult(acc,y));
```

If parametrized with the evaluation-step metric, the prototype produces the following output for *bftMult*.

[9] In fact, the bound that is presented to a user is at bit more general because the analysis can not assume that the dimensions of the matrices fit.

```
bftMult: (T(L(L(int))),L(L(int))) → L(L(int))
Positive annotations of the argument
(0,0)    → 51.0      (1,1)     → 15.0
(0,[1])  → 2.0       ([1],1)   → 14.0
(1,0)    → 104.0     ([[1]],1) → 15.0

The number of evaluation steps consumed by bftMult is at most:
   2.0*y*z + 15.0*y*n*m*x + 14.0*y*n*m + 15.0*y*n + 104.0*n + 51.0
where
   n is the size of the first component of the input
   m is the length of the nodes of the first component of the input
   x is the length of the elements of the nodes of the
     first component of the input
   y is the length of the second component of the input
   z is the length of the elements of the second comp. of the input
```

The prototype derives a non-trivial, asymptotically-tight bound on the number of evaluation-steps that are used by *bftMult*. The analysis of the whole program takes about 30 seconds on a usual desktop computer. It is unclear how such a bound can be computed without the use of amortized analysis.

We compared the computed evaluation-step bound with the measured run time of *bftMult* for balanced binary trees with quadratic matrices. Figure 7.4 shows the result of this experiment where x denotes the number of nodes in the tree and $y \times y$ is the dimension of the matrices. The constant factors in the bound almost match the optimal ones.

The White Rabbit put on his spectacles. 'Where shall I begin, please your Majesty?' he asked. 'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

LEWIS CARROLL
Alice's Adventures in Wonderland (1865)

8 Related Research

The static computation of resource bounds for programs has been studied by computer scientists since the 70s. Today, there exist many different techniques for computing bounds.

In this chapter, I compare my work with related research on automatic resource analysis and on verification of resource bounds. Classically, automatic resource analysis is based on recurrence relations. I discuss this long line of work in Section 8.1. Most closely related to the work in this dissertation is the previous work on automatic amortized analysis, which I describe in Section 8.2.

Other important techniques for resource analysis use sized types, or abstract interpretation and invariant generation. I discuss this research in Section 8.3 and 8.4, respectively. Further related work is discussed in Section 8.5.

8.1 Recurrence Relations

The use of recurrence relations (or recurrences) in automatic resource analysis was pioneered by Wegbreit [Weg75] (compare the discussion in Section 1.2). The proposed analysis is performed in two steps: first extract recurrences from the program, then compute closed expressions for the recurrences. Wegbreit implemented his analysis in the METRIC system to analyze LISP programs but notices that it "can only handle simple programs" [Weg75]. The most complicated examples that he provides are a reverse function for lists and a union function for sets represented by lists.

Webreit's method dominated automatic resource analysis for many years. Benzinger [Ben01] notices in 2001:

> "Automated complexity analysis is a perennial yet surprisingly disregarded aspect of static program analysis. The seminal contribution to this area was

> Wegbreit's METRIC system, which even today still represents the state-of-the-art in many aspects."

Ramshaw [Ram79] and Hickey et al. [HC88] address the derivation of recurrences for *average-case* analysis.

Flajolet et al. [FSZ91] describe a *theory of exact analysis in terms of generating functions* for average-case analysis. A fragment of this theory was implemented in an automatic average-case analyses of algorithms for "decomposable" combinatorial structures. Possible applications of Flajolet's method to worst-case analysis were not explored.

The ACE system of Le Métayer [Mét88] analyses FP programs in two phases. A recursive function is first transformed into a recursive function that bounds the complexity of the original function. This function is then transformed into a non-recursive one, using predefined patterns. The ACE system can only derive asymptotic bounds rather than constant factors as it is done in RAML.

Rosendahl [Ros89] implemented an automatic resource analysis for first-order LISP programs. The analysis first converts programs into *step-counting* version which is then converted into a time bound function via abstract interpretation of the step-counting version. The reported results are similar to Wegbreit's results and programs with nested data structures and typical compositional programs can not be handled.

Benzinger [Ben01, Ben04] applied Wegbreit's method in an automatic complexity analysis for higher-order Nuprl terms. He uses Mathematica to solve the generated recurrence equations. Grobauer [Gro01] reported an interesting mechanism to automatically derive cost recurrences from DML programs using dependent types. The computation of closed forms for the recurrences is however not discussed.

Recurrence relations were also proposed to automatically derive resource bounds for logic programs [DL93].

The COSTA Project

In the COSTA project, both the derivation and the solution of recurrences are studied. Albert et al. [AAG+07] introduced a method for automatically inferring recurrence relations from Java bytecode. They rely on abstract interpretation to generate size relations between program variables at different program points.

The COSTA team states that existing computer algebra systems are in most cases not capable of handling recurrences that originate from resource analysis [AAGP08]. As a result, a series of papers [AAGP08, AAGP11, AGM11] studies the derivation of closed forms for so called *cost relations*; recurrences that are produced by automatic resource analysis. They use partial evaluation and apply static analysis techniques such as abstract interpretation to obtain loop invariants and ranking functions. Another work [AAA+09] studies the computation of *asymptotic* bounds for recurrences.

While the COSTA system can compute bounds that contain integers, the amortized method is favorable in the presence of (nested) data structures and function composition.

8.2 Automatic Amortized Analysis

The research on automatic amortized resource analysis is in many respects inspired by Hofmann's work [Hof00b, Hof00a] on LFPL. Hofmann defines the first-order functional programming language LFPL and shows that LFPL programs can be compiled into a fragment of the programming language C without dynamic memory allocation. LFPL features a linear type system with a special type $\diamond$ that is used to make memory cells first class objects in the language. The destruction of data in a pattern match then binds a memory cell to a variable that can be used for data construction.

Hofmann also showed [Hof02] that adding higher-order functions to LFPL leads to a programming language that can exactly define the functions that are computable in polynomial space and an unbounded stack or equivalently (using a result of Cook) in exponential time.

The concept of automatic amortized resource analysis was introduced by Hofmann and Jost. In a seminal paper [HJ03], they use the potential method to analyze the heap-space consumption of first-order functional programs; establishing the idea of attaching potential to data structures, the use of type systems to prove bounds, and the inference of type annotations using linear programming. By contrast with my work, the analysis system uses linear potential annotations and thus derives linear resource bounds only (as described in Chapter 4).

The subsequent work on amortized analysis for functional programs successively broadened the range of this analysis method while the limitation to linear bounds remained. Jost et al. [JLH+09] extended automatic amortized analysis to generic resource metrics and user defined inductive data structures. A particularly interesting aspect of this work is the development of a resource metric for real-world examples: An amortized analysis system was built into a compiler for the language Hume [HDF+06] and was successfully used in concrete embedded systems to compute memory and clock-cycle bounds for 32 MHz Renesas M32C/85U embedded micro-controllers.

Campbell [Cam09] developed an amortized resource analysis that computes bounds on the stack space of functional programs. He uses potential annotations that define functions in the *depth* of data structures. The potential is linear in the depth of tree-like data and is reflected in tree-like typing contexts. Campbell also proposed a restitution of potential that makes the stack-space analysis more precise; this could also be of interest in for other resources.

Jost et al. [JHLH10] extended linear amortized resource analysis to polymorphic and higher-order programs. Higher-order functions are resource-parametrically analyzed without a previous defunctionalization. In this way, function types can express the cost behaviors at different call sites with only one analysis the function's definition.

Automatic amortized resource analysis was successfully applied to object-oriented programs, too. Hofmann and Jost [HJ06] refined potential annotations with so called *views* to deal with object-oriented language features such as inheritance, casts, and imperative updates. Even though Hofmann and Rodriguez [HR09] presented an automatic

type-checking algorithm, type inference for views and potential annotations is still an open problem.

Atkey [Atk10] integrated linear amortized analysis into a program logic for Java-like bytecode using bunched implications and separation logic. The separating conjunction $A * B$ is used for both separating the data on the heap and the potential that is attached to the data. A subset of the logic allows for effective proof search and inference of resource annotations. Interestingly, the resource logic is also used to prove termination in the presence of cyclic data structures.

All the previous works on amortized analysis only describe systems that are restricted to linear bounds—as the original system by Hofmann and Jost [HJ03]. In this dissertation I present the first automatic amortized analyses for super-linear bounds. Parts of my work appeared at several conferences [HH10b, HH10a, HAH11].

8.3 Sized Types

A *sized type* is a type that features size bounds for the inhabiting values. The size information is usually attached to inductive data types via natural numbers. The difference to the potential annotations of amortized analysis is that sized types bound sizes of data while potential annotations define a potential as a function of the data size.

Sized types were introduced by Hughes et al. [HPS96] in the context of functional reactive programming to prove that stream manipulating functions are productive or in other words, that the computation of each stream element terminates.

Hughes and Pareto [HP99] studied the use of sized types to derive space bounds for a functional language with region-based memory management. The type system features both resource and size annotations to express bounds but the annotations have to be provided by the programmer.

Type inference for sized types was first studied by Chin and Khoo [CK01]. They employ an approximation algorithm for the transitive closure of Presburger constraints to infer size relations for recursive functions. The algorithm only computes *linear* relations and does not scale well for nested data structures.

Vasconcelos [Vas08] studies sized types to infer upper bounds on the resource usage of higher-order functional programs. He employs abstract interpretation techniques for automatically inferring linear approximations of the sizes of data structures and the resource usage of recursive functions. In contrast to RAML, this system can only compute linear bounds.

8.4 Abstract Interpretation

Abstract interpretation is a well-established framework for static program analysis. There are several works that employ abstract interpretation to compute symbolic complexity bounds. Unfortunately, none of the described prototype implementations is

publicly available. Hence, I can compare our analysis only to the results that are reported in the respective papers.

WCET Analysis

Worst-case execution time (WCET) analysis is a large research area that traditionally computes time bounds for "a restricted form of programming, which guarantees that programs always terminate and recursion is not allowed or explicitly bounded as are the iteration counts of loops" [WEE+08]. The time bounds are computed for specific hardware architectures and are very precise because the analysis takes into account low-level features like hardware caches and instruction pipelines.

By contrast with traditional WCET analysis, *parametric* WCET analysis uses abstract interpretation to compute symbolic clock-cycle bounds for specific hardware. Lisper [Lis03] proposed the use of a flow analysis with a polyhedral abstraction and the computation of symbolic bounds for the points in a polyhedral. This method was recently implemented [BEL09]. In contrast to my work it can only handle integer programs without dynamic allocation and recursion.

Altmeyer et al. [AHLW08, AAN11] reported a similar approach. They propose a parametric loop analysis that consists of four phases: identifying loop counters, deriving loop invariants, evaluation of loop exits, and finally, construction of loop bounds. The analysis operates directly on executables and can also handle recursion. However, a user has to provide parameters that bound the recursion (or loop iterations) that traverses a data structure. In contrast, our analysis is fully automatic.

The SPEED Project

A successful method to estimate time bounds for C++ procedures with loops and recursion was recently developed by Gulwani et al. [GG08, GMC09] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. An alternative approach that leads to impressive experimental results is to use "a database of known loop iteration lemmas" instead of the counter instrumentation [GJK09].

Another recent innovation for non-recursive programs is the combination of disjunctive invariant generation via abstract interpretation with proof rules that employ SMT-solvers [GZ10].

In contrast to our method, these techniques can not fully automatically analyze iterations over data structures. Instead, the user needs to define numerical "quantitative functions". This seems to be less modular for nested data structures where the user needs to specify an "owner predicate" for inner data structures. It is also unclear if quantitative functions can represent complex mixed bounds such as $\sum_{1\leq i<j\leq n}(10m_i + 2m_j) + 16\binom{n}{2} + 12n + 3$ which RAML computes for *isortlist*. Moreover, our method infers tight bounds for functions such as insertion sort that admit a worst-case time usage of

the form $\sum_{1 \le i \le n} i$. In contrast, [GMC09] indicates that a nested loop on $1 \le i \le n$ and $1 \le j \le i$ is over-approximated with the bound n^2.

A methodological difference to techniques based on abstract interpretation is that we infer (using linear programming) an abstract potential function which indirectly yields a resource-bounding function. The potential-based approach may be favorable in the presence of compositions and data scattered over different locations (partitions in quick sort). Additionally, there seem to be no experiments that relate the derived bounds to the actual worst-case behavior and there is no publicly available implementation.

As any type system, our approach is naturally compositional and lends itself to the smooth integration of components whose implementation is not available. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [BHMS04]. On the other hand, our method does not model the interaction of integer arithmetic with resource usage.

8.5 Other Work

There are techniques [BFGY08, CFGV09] that can compute the memory requirements of object oriented programs with region based garbage collection. These systems infer invariants and use external tools that count the number of integer points in the corresponding polytopes to obtain bonds. The described technique can handle loops but not recursive or composed functions.

Taha et al. [TEX03] describe a two-stage programming language in which the first stage can arbitrarily allocate memory and the second stage—that uses Hofmann's LFPL [Hof00b]—can allocate no memory. However, the work reports no method to derive a memory bound for the first stage.

Other related works use type systems to validate resource bounds. Crary and Weirich [CW00] presented a (monomorphic) type system capable of specifying and certifying resource consumption. Danielsson [Dan08] provided a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of purely functional data structures and algorithms. In contrast, my focus is on the inference of bounds.

Chin et al. [CNPQ08] use a Presburger solver to obtain *linear* memory bounds for low-level programs. In contrast, the analysis system I present can compute polynomial bounds.

Polynomial resource bounds were also studied by Shkaravska et al. [SvKvE07] who address the derivation of polynomial size bounds for functions whose exact growth rate is polynomial. Besides this strong restriction, the efficiency of inference remains unclear.

We have seen that amortization is a powerful tool in the algorithmic analysis of data structures. … It seems likely that amortization will find many more uses in the future.

ROBERT ENDRE TARJAN
Amortized Computational Complexity (1985)

9 Conclusion

In this dissertation, I described a novel automatic amortized resource analysis for first-order functional programs. I presented it in the form of type systems for the programming language Resource Aware ML and proved the soundness of the bounds with respect to a big-step operational semantics. In this way, I interlinked two classic areas of theoretical computer science: the analysis of algorithms and the design and implementation of programming languages.

The proposed analysis uses multivariate resource polynomials, which interact well with pattern matching and express a wide range of polynomial relations between different elements of the input. This enables the formulation of simple local type rules that can be easily checked. I developed an efficient type-inference algorithm that relies on linear constraint solving only. The result is the first type-based resource analysis system that automatically computes polynomial bounds.

An experimental evaluation with a prototype implementation showed that programs are analyzed efficiently in practice. I compared the computed bounds with the measured worst-case behavior of programs and found that the constant factors are often close or identical to the optimal ones.

In short, the developed polynomial amortized resource analysis is

- *precise*, since the bounds are resource polynomials,
- *efficient*, because the inference is based on linear programming,
- *reliable*, because of the formal soundness prove with respect to the semantics,
- and *verifiable*, since type derivations are certificates of the bounds.

Nevertheless, the automatic computation of resource bounds is an undecidable problem. As a result, an automatic resource analysis can never achieve the same range and precision of a careful manual analysis.

The method I proposed broadened the range of programs that can be automatically analyzed. However, it can only compute polynomial bounds and the user has to provide a maximal degree in the inference to restrict the search space of the bounds. Additionally, there are still programs with a polynomial resource behavior that can not be analyzed automatically.

Yet it remains true that manual analyses are prone to error and are often infeasible in software development. That is why I envision a twofold approach to resource analysis, which enables software engineers to work with quantitative resource bounds in the same way they work with usual type information. As types, resource bounds should be inferred in most cases. But if the inference fails it should be simple and natural to enrich parts of programs with resource information and to formally reason about soundness in a flexible way.

The findings in this dissertation provide a basis for such a research enterprise. My colleagues and I have already started to investigate the extension of polynomial amortized resource analysis to more advanced language features such as garbage collection, higher-order functions, and user-defined data types. At the same time we are working on the integration of non-polynomial bounds such as $n \log n$ and 2^n.

Techniques such as multivariate resource polynomials and additive shifts might be useful in the development of quantitative program logics that prove the soundness of user-annotated typings. An interactive prove system could rely on our automatic inference methods to ease the use of the logic. Conversely, the user-annotated types could be used in the type inference.

Unlike classic software verification, a quantitative resource analysis of a program cannot prove its correctness. But the correctness of a program can only proved with respect to some specification. The reason why resource analysis appeals to me is the absence of an external specification; its worst-case resource behavior is inextricably linked with every program.

Bibliography

[AAA+09] Elvira Albert, Diego Alonso, Puri Arenas, Samir Genaim, and Germán Puebla. Asymptotic Resource Usage Bounds. In *Programming Languages and Systems - 7th Asian Symposium (APLAS'09)*, pages 294–310, 2009.

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In *Programming Languages and Systems - 16th European Symposium on Programming (ESOP'07)*, pages 157–172, 2007.

[AAGP08] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis - 15th International Symposium (SAS'08)*, pages 221–237, 2008.

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[AAN11] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. Precise and Efficient Parametric Path Analysis. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'11)*, pages 141–150, 2011.

[AB98] Mohamad Akra and Louay Bazzi. On the Solution of Linear Recurrence Equations. *Comput. Optim. Appl.*, 10:195–210, May 1998.

[AGM11] Elvira Albert, Samir Genaim, and Abu Naser Masud. More Precise Yet Widely Applicable Cost Analysis. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference (VMCAI'11)*, pages 38–53, 2011.

[AHLW08] Sebastian Altmeyer, Christian Humbert, Björn Lisper, and Reinhard Wilhelm. Parametric Timing Analysis for Complex Architectures. In *4th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 367–376, 2008.

[Atk10] Robert Atkey. Amortised Resource Analysis with Separation Logic. In *Programming Languages and Systems - 19th European Symposium on Programming (ESOP'10)*, pages 85–103, 2010.

[BEL09] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An Efficient Algorithm for Parametric WCET Calculation. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 13–21, 2009.

[Ben01] Ralph Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *J. Funct. Program.*, 11(1):3–31, 2001.

[Ben04] Ralph Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[BFGY08] Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *7th International Symposium on Memory Management (ISMM'08)*, pages 141–150, 2008.

[BHMS04] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic Certification of Heap Consumption. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference (LPAR'04)*, pages 347–362, 2004.

[BPZZ05] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR*, abs/cs/0512056, 2005.

[BT97] Dimitris Bertsimas and John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[Cam09] Brian Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *Programming Languages and Systems - 18th European Symposium on Programming (ESOP'09)*, pages 190–204, 2009.

[CC92] Patrick Cousot and Radhia Cousot. Inductive Definitions, Semantics and Abstract Interpretations. In *19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 83–94, 1992.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages 153–162, 2003.

[CFGV09] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic Polynomial Maximization Over Convex Sets and Its Application to Memory Requirement Estimation. *IEEE Trans. VLSI Syst.*, 17(8):983–996, 2009.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating Sized Types. *High.-Ord. and Symb. Comp.*, 14(2-3):261–300, 2001.

[CNPQ08] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *7th International Symposium on Memory Management (ISMM'08)*, pages 151–160, 2008.

[Coh82] Jacques Cohen. Computer-Assisted Microanalysis of Programs. *Commun. ACM*, 25(10):724–733, 1982.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188, 1987.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CW00] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 184–198, 2000.

[Dan08] Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 133–144, 2008.

[DL93] Saumya K. Debray and Nai-Wei Lin. Cost Analysis of Logic Programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, 1993.

[EP08] P. Etingof and I. Pak. An algebraic extension of the MacMahon Master Theorem. *Proceedings of the American Mathematical Society*, 136(7):2279–2288, 2008.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[FSZ91] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic Average-Case Analysis of Algorithms. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.

[GG08] Bhargav S. Gulavani and Sumit Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Computer Aided Verification, 20nd International Conference (CAV '08)*, pages 370–384, 2008.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conference on Programming Language Design and Implementation (PLDI'09)*, pages 375–385, 2009.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1994.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139, 2009.

[Gro01] Bernd Grobauer. Cost Recurrences for DML Programs. In *6th International Conference on Functional Programming (ICFP'01)*, pages 253–264, 2001.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded Linear Logic. *Theoret. Comput. Sci.*, 97(1):1–66, 1992.

[GZ10] Sumit Gulwani and Florian Zuleger. The Reachability-Bound Problem. In *Conference on Programming Language Design and Implementation (PLDI'10)*, pages 292–304, 2010.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 357–370, 2011.

[HC88] Timothy J. Hickey and Jacques Cohen. Automating Program Analysis. *J. ACM*, 35(1):185–220, 1988.

[HDF+06] K. Hammond, R. Dyckhoff, C. Ferdinand, R. Heckmann, M. Hofmann, H.-W. Loidl, G. Michaelson, J. Sérot, and A. Wallace. The EmBounded Project: Automatic Prediction of Resource Bounds for Embedded Systems. In *7th Symposium on Trends in Functional Programming (TFP'06)*, 2006.

[HH10a] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Programming Languages and Systems - 8th Asian Symposium (APLAS'10)*, pages 172–187, 2010.

[HH10b] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems - 19th European Symposium on Programming (ESOP'10)*, pages 287–306, 2010.

[HJ03] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *Programming Languages and Systems - 15th European Symposium on Programming (ESOP'06)*, pages 22–37, 2006.

[HM03] Kevin Hammond and Greg Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *International Conference on Generative Programming and Component Engineering (GPCE'03)*, pages 37–56. LNCS 2830, 2003.

[Hof00a] Martin Hofmann. A type system for bounded space and functional in-place update. *Nord. J. Comput.*, 7(4):258–289, 2000.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In *Programming Languages and Systems - 9th European Symposium on Programming (ESOP'00)*, pages 165–179, 2000.

[Hof02] Martin Hofmann. The strength of non-size increasing computation. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 260–269, 2002.

[HP99] John Hughes and Lars Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *4th International Conference on Functional Programming (ICFP'99)*, pages 70–81, 1999.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, 1996.

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *18th Conference on Computer Science Logic (CSL'09)*. LNCS, 2009.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 223–236, 2010.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th International Symposium on Formal Methods (FM'09)*, pages 354–369, 2009.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[Ler06] Xavier Leroy. Coinductive Big-Step Operational Semantics. In *Programming Languages and Systems - 15th European Symposium on Programming (ESOP'06)*, pages 54–68, 2006.

[Lis03] Björn Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET'03)*, pages 99–102, 2003.

[Mét88] Daniel Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pou06] Marc Pouzet. *Lucid Synchrone, Version 3. Tutorial and Reference Manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.

[Ram79] Lyle Harold Ramshaw. *Formalizing the Analysis of Algorithms*. PhD thesis, Stanford University, Stanford, CA, USA, 1979. AAI8001994.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, 1972.

[Ros89] Mads Rosendahl. Automatic Complexity Analysis. In *Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156, 1989.

[Rou01] Salvador Roura. Improved Master Theorems for Divide-And-Conquer Recurrences. *J. ACM*, 48(2):170–205, 2001.

[SvKvE07] Olha Shkaravska, Ron van Kesteren, and Marko C. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications, 7th International Conference (TLCA'07)*, pages 351–365, 2007.

[Tar85] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[TEX03] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Embedded Software, Third International Conference (EMSOFT'03)*, pages 340–355, 2003.

[Vas08] Pedro Vasconcelos. *Space Cost Analysis Using Sized Types.* PhD thesis, School of Computer Science, University of St Andrews, 2008.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[Weg75] Ben Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.

[ZZ89] Paul Zimmermann and Wolf Zimmermann. The Automatic Complexity Analysis of Divide-And-Conquer Algorithms. Research Report RR-1149, INRIA, 1989. Projet EURECA.