Automated Amortised Analysis

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

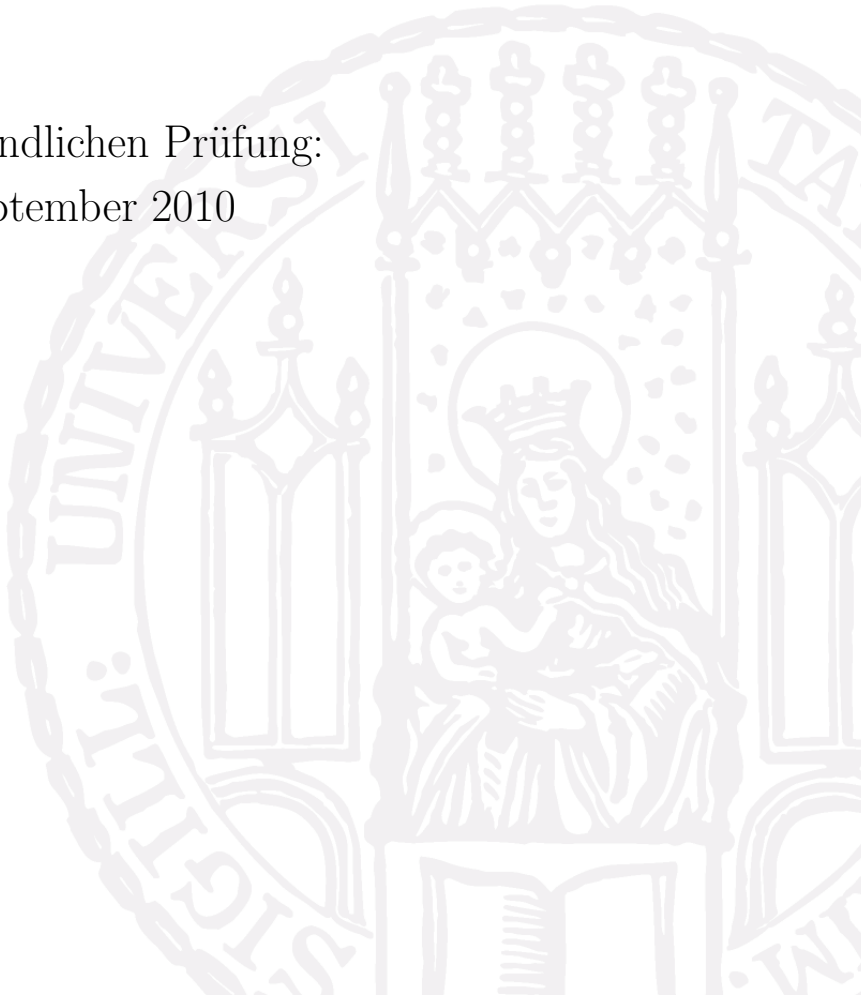
Vorgelegt von
Dipl.-Math. Steffen Jost
am 9. August 2010

Berichterstatter:

Prof. Martin Hofmann
(Ludwig-Maximilians-Universität München, Deutschland)

Prof. Kevin Hammond
(University of St Andrews, Schottland)

Tag der mündlichen Prüfung:
16. September 2010



To my grand-parents, who could not afford to wait for this thesis to be completed.

Deutsche Zusammenfassung

Steffen Jost stellt eine neuartige *statische* Programmanalyse vor, welche *vollautomatisch* Schranken an den Verbrauch quantitativer Ressourcen berechnet. Die Grundidee basiert auf der Technik der Amortisierten Komplexitätsanalyse, deren nichttriviale Automatisierung durch ein erweitertes Typsystem erreicht wird. Das Typsystem berechnet als Nebenprodukt ein lineares Gleichungssystem, dessen Lösungen Koeffizienten für lineare Formeln liefern. Diese Formeln stellen *garantierte* obere Schranken an den Speicher- oder Zeitverbrauch des analysierten Programms dar, in Abhängigkeit von den verschiedenen Eingabegrößen des Programms. Die Relevanz der einzelnen Eingabegrößen auf den Ressourcenverbrauch wird so deutlich beziffert.

Die formale Korrektheit der Analyse wird für eine funktionale Programmiersprache bewiesen. Die strikte Sprache erlaubt: Typen höherer Ordnung, volle Rekursion, verschachtelte Datentypen, explizites Aufschieben der Auswertung und Aliasing. Die formale Beschreibung der Analyse befasst sich primär mit dem Verbrauch von dynamischen Speicherplatz. Für eine Reihe von realistischen Programmbeispielen wird demonstriert, dass die angefertigte *generische* Implementation auch gute Schranken an den Verbrauch von Stapelspeicher und der maximalen Ausführungszeit ermitteln kann. Die Analyse ist sehr effizient implementierbar, und behandelt auch größere Beispielprogramme vollständig in wenigen Sekunden.

English Abstract

Steffen Jost researched a novel *static* program analysis that *automatically* infers formally *guaranteed* upper bounds on the use of compositional quantitative resources. The technique is based on the manual amortised complexity analysis. Inference is achieved through a type system annotated with linear constraints. Any solution to the collected constraints yields the coefficients of a formula, that expresses an upper bound on the resource consumption of a program through the sizes of its various inputs.

The main result is the formal soundness proof of the proposed analysis for a functional language. The strictly evaluated language features higher-order types, full mutual recursion, nested data types, suspension of evaluation, and can deal with aliased data. The presentation focuses on heap space bounds. Extensions allowing the inference of bounds on stack space usage and worst-case execution time are demonstrated for several realistic program examples. These bounds were inferred by the created *generic* implementation of the technique. The implementation is highly efficient, and solves even large examples within seconds.

Acknowledgements

The first person to thank here is my supervisor Martin Hofmann. He kindly invited me to write this thesis about automatic resource analysis at his chair in theoretical computer science in Munich. Martin provided me with essential guidance throughout, especially whenever I believed the problems to be insurmountable.

I thank my current supervisor Kevin Hammond for his offer to work in St Andrews. He provided me with manifold advice, valuable academic freedom and generous support to continue the research on my thesis while I worked for him in bonnie Scotland.

The members of the EmBounded and MRG research projects provided a highly useful environment for researching this thesis. This allowed the analysis to reach an unhoped-for maturity. I especially thank my colleague Hans-Wolfgang Loidl not only for good remarks on my draft, but also for the excellent cooperation in producing and maintaining the implementation and numerous program examples to be analysed.

Apologies to Brian Campbell for not providing him with a final version of this thesis in time, despite his dissertation extending this one. His comments on my dry technical drafts and discussions on our respective works have been beneficial.

My friend and colleague Hugo Simoes deserves a similar apology. Discussing our closely related works has always been interesting and led to significant improvements of this thesis. Especially his observation, that preservation of memory consistency for the higher-order system can be proven independently, came at a crucial point in time.

My thanks extend to Pedro Vasconcelos for good discussions and for explaining his good work on sized types to me; Olha Shkaravska for recognising the connection between Tarjan's work and mine; Jeremy Siek, Ekaterina Komendantskaya and Vladimir Komendantsky for discussions on co-induction; Graham Hutton for suggesting to analyse his synthesised abstract machine; Christoph Herrmann and Robert Rothenberg for proof reading.

I am grateful to my entire family for their ample support. I owe my wife Andrea so much for taking care about really everything so that I could work on this thesis, especially for such a long time. I thank her and our children Melina and Jannis for still loving me despite all the weekends that I spent at my desk instead of with them.

Financial support is acknowledged from the *Graduiertenkolleg Logik in der Informatik* Munich (GKLI); the EU FET-IST projects IST-510255 (EmBounded), IST-2001-33149 (MRG), IST-248828 (Advance); and EPSRC grant EP/F030657/1 (Islay).

Contents

Contents xiii

List of Figures xv

List of Tables xvi

List of Theorems and Definitions xix
Theorems and Definitions of LF xix
Theorems and Definitions of $LF_{\diamond}$ xix
Theorems and Definitions of ARTHUR xx

1 Introduction 1
1.1 Contributions 3
1.2 Thesis Overview 5
1.3 Mathematical Preliminaries 8

2 Intuitive Description of the Proposed Technique 11
2.1 Amortised Analysis 12
2.2 Example: Analysing a Queue 14
2.3 Automatising the Amortised Analysis 16
2.4 Advances over Previous Amortised Analysis Techniques 18

3 Related Research 21
3.1 Affiliated Research 22
3.2 Future Offspring 26
3.3 Distant Relatives 29

4 Basis Language LF 37
4.1 Syntax of LF 38
4.1.1 Let-normal form. 40
4.2 Type System 41
4.3 Operational Semantics 46
4.3.1 Storeless Semantics 51

4.3.2 Memory Consistency 54
4.3.3 Circular Data Structures 56
4.3.4 Observations on Memory Consistency 58
4.4 Introductory Program Examples 63
4.4.1 List Tails 64
4.4.2 List Reversal 66
4.4.3 List Zipping 68
4.4.4 In-Place Insertion Sort 68
4.5 Cost-Aware Semantics 70
4.5.1 Measuring Space Usage 74

5 First-Order Analysis $LF_{\diamond}$ 79
5.1 Cost-Aware Type System $LF_{\diamond}$ 80
5.1.1 Exact Resource Analysis 85
5.2 Potential 86
5.2.1 Generalisation to Arbitrary Recursive Data Types 88
5.2.2 Potential Properties 89
5.3 Soundness of the First-Order Analysis 91
Soundness Theorem LF 91
5.4 Misbehaving Programs and Non-Termination 98
Soundness Theorem for non-terminating LF 100
5.5 Introductory Program Examples Analysed 104
5.5.1 List Tails Analysed 104
5.5.2 List Reversal Analysed 105
5.5.3 List Zipping Analysed 106
5.5.4 In-Place Insertion Sort Analysed 108
5.6 Analysing an Abstract Machine for Arithmetic Expressions 110

6 Higher-Order Analysis Arthur 119
6.1 Defining Arthur 120
6.1.1 Recursive Let Definitions 122
6.1.2 Currying, Uncurrying and Partial Application 123
6.1.3 Additive Pairs 124
6.2 Evaluating Arthur 125
6.3 Typing Arthur 130
6.4 Arthur's Memory Consistency 143
6.5 Arthur's Potential 151
6.6 Soundness of the Higher-Order Analysis 154

7 Implementation **165**
7.1 Interchangeable cost metrics 168
7.2 Ghost-let-normal form 169
7.3 Inference of Annotated Types 170
7.4 Bounds in Natural Language 173
7.5 Interactive Solution Space Exploration 174

8 Examples **177**
8.1 List Folding 179
8.2 Nesting and Function Composition 184
8.3 Applying Twice, Quad and Quad of Quad 188
8.4 Tree Operations 191
8.5 Sum-of-Squares Variations 194
8.6 Revisiting an Abstract Machine for Arithmetic Expressions 197
8.7 Higher-Order Expression Evaluator with Local Definitions 199
8.8 WCET of a Real-Time Controller for an Inverted Pendulum 202

9 Conclusion **207**
9.1 Complexity of the Inference 208
9.2 Limitations 213

Bibliography **217**

Index **227**

List of Figures

2.1 Simulating a queue by using two stacks 15

3.1 Family feature comparison 23
3.2 Comparison of related work 30

4.1 Syntax of the simple language LF 39
4.2 Types of the simple language LF 41
4.3 Free variables of LF terms 45
4.4 List tails 65
4.5 In-place list reversal 67
4.6 List zipping 69
4.7 In-place insertion sort 69

5.1 Types of the cost-aware simple language $\mathrm{LF}_\diamond$ 80
5.2 Sharing relation on $\mathrm{LF}_\diamond$ types 84
5.3 Potential 87
5.4 LP generated for in-place insertion sort 109
5.5 Semantics for a simple language of arithmetic expressions 110
5.6 Abstract machine for evaluating arithmetic expressions 111

6.1 Syntax of the language Arthur 121
6.2 Arthur type grammar 132
6.3 Free resource variables of Arthur types 133
6.4 Ternary subtyping relation for Arthur 139
6.5 Sharing relation for Arthur 140
6.6 Potential of Arthur types 152

8.1 Source code of list-sum 179
8.2 Source code for composite mapping 183
8.3 Twice and Quad 189
8.4 Source code of tree-flattening (`flatten`) 191
8.5 Source code for `repmin` program example 193
8.6 Source code of sum-of-squares (3 variants) 196

8.7 Continuation-passing arithmetic evaluator 198
8.8 Higher-order program evaluator 200

List of Tables

8.1 Analysis predictions and measurements for calculating sums of integer lists of length N by folding 182
8.2 Predictions and measurements for higher-order depth-first tree traversal 192
8.3 Analysis predictions and measurements for higher-order tree replacement 194
8.4 Results for three variants of the sum-of-squares function 195

9.1 Run-time for Analysis and for LP-solving 212

List of Theorems and Definitions

Basis Language LF **38**
4.1 Definition (LF Type Rules) . . . 42
4.2 Definition (LF Sharing) . . . 44
4.3 Definition (Free Variables of LF Terms) . . . 46
4.4 Definition (Program) . . . 46
4.5 Definition (Well-Typed LF Program) . . . 46
4.6 Definition (Store and Values) . . . 47
4.7 Definition (LF Operational Semantics) . . . 49
4.8 Lemma (Heap Stability) . . . 53
4.9 Definition (Consistency of Heap, Stack and Typing Context) . . . 54
4.10 Lemma (Closure of Memory Consistency) . . . 58
4.11 Lemma (Joint Consistency) . . . 58
4.12 Lemma (Consistency of Deallocation) . . . 59
4.13 Lemma (Preservation) . . . 60
4.14 Definition (Sizes of Operational Values) . . . 71
4.15 Definition (Sizes of LF Types) . . . 72
4.16 Lemma (Size Correspondence) . . . 72
4.17 Definition (LF Operational Semantics with Freelist Counter) . . . 74
4.18 Lemma (Additional Memory) . . . 76
4.19 Lemma (Cost Counting Inviolacy) . . . 77
4.20 Lemma (Finite Evaluation Costs) . . . 77

First-Order Analysis $LF_\diamond$ **80**
5.1 Definition ($LF_\diamond$ Type Reduction) . . . 81
5.2 Definition (Sizes of $LF_\diamond$ Types) . . . 81
5.3 Definition ($LF_\diamond$ Type Rules) . . . 82
5.4 Definition ($LF_\diamond$ Sharing) . . . 83
5.5 Definition (Well-Typed $LF_\diamond$ Program) . . . 84
5.6 Lemma (Embedding LF in $LF_\diamond$) . . . 84
5.7 Definition (Potential) . . . 87
5.8 Lemma (Potential Deallocation) . . . 89

5.9 Lemma (Non-Increasing Potential) . . . 90
5.10 Lemma (Shared Potential) . . . 90
1 Theorem (Soundness LF) . . . 91
5.11 Definition (Out of Memory Semantics) . . . 98
2 Theorem (Soundness for Non-Terminating LF) . . . 100

Higher-Order Analysis Arthur **120**
6.1 Definition (Free Variables of Arthur Terms) . . . 122
6.2 Definition (Arthur Values) . . . 125
6.3 Definition (Sizes of Arthur Values) . . . 126
6.4 Definition (Arthur Operational Semantics) . . . 127
6.5 Lemma (Heap Stability) . . . 130
6.6 Definition (Free Resource Variables of Arthur Types) . . . 133
6.7 Definition (Sizes of Arthur Types) . . . 133
6.8 Definition (Arthur Type Rules) . . . 134
6.9 Definition (Subtyping) . . . 138
6.10 Lemma (Transitivity of Subtyping) . . . 138
6.11 Definition (Arthur Sharing) . . . 138
6.12 Lemma (Shared Subtyping) . . . 140
6.13 Lemma (Inversion Lemma) . . . 141
6.14 Definition (Memory Consistency) . . . 144
6.15 Lemma (Consistent Memory Extension) . . . 147
6.16 Lemma (Memory Consistency Preservation) . . . 149
6.17 Lemma (Size Correspondence) . . . 149
6.18 Lemma (Size Determination) . . . 150
6.19 Lemma (Consistent Subtyping) . . . 150
6.20 Definition (Potential) . . . 151
6.21 Lemma (Subtyped Potential) . . . 153
6.22 Lemma (Shared Potential) . . . 154
3 Theorem (Soundness Arthur) . . . 155

1 Introduction

This thesis is about ensuring the correctness of computer systems in an *automated* way. Humanity has reached a point in history where a vast amount of computing power is readily available to the masses. It has become evident that the main challenge now lies in harnessing this vast power. Errors in software design have already caused a lot of damage. While the outstanding failures of expensive endeavours like space rockets[A] or interplanetary probes[B] are highly regrettable, the problem has already gained an entirely new dimension by the fact that small embedded computer systems have become an ubiquitous part of everyday life: approximately four billion of these small systems were shipped worldwide in 2006 alone [Vol07]. These numerous systems must be quickly developed under ever increasing economic pressures, while at the same time it must also be ensured somehow that, for example, an airbag is inflated at exactly the right time. While industry is currently satisfied that airbags are inflated when needed in the majority of cases, we believe that it is desirable that such devices *always* work as expected – at least they should not fail any more due to *automatically* detectable software errors.

We believe that an important step in the solution of the software crisis lies in the use of *automated* formal methods. Formal methods are already employed for some safety-critical projects, but they are expensive because they require highly trained specialists and can take significant time to perform. A widespread use of these techniques can only be achieved if the average programmer is able to quickly employ a proven software analysis tool as a part of the coding process. Once such a highly complex automated tool is developed, it is cheap to use. In addition to increasing safety by allowing the programmer to verify that his program meets its specifications, such automated tools will also reduce the time to market a product by eliminating the need for expensive testing and redesign cycles. This reduction in the design cost of a product makes the use of formal methods appealing even for non-safety-critical applications. More reliable products at a cheaper price is clear win-win situation for both producer and consumer.

It is our thesis that the wide-spread use of such automated program analysis techniques is now practical. We present such a system together with its implementation, in this thesis. The analysis that we present will not guarantee the overall correctness of a program in every respect, for this would be a far too complex a task to achieve in a single piece of work. Instead, we focus on the resource requirements for a given program, namely that our analysis establishes an *upper bound* on a program's overall resource usage.

[A] ESA Ariane 5, 4 June 1996, $\sim 1,000,000,000$\$, [Dow97]
[B] NASA Mars Climate Orbiter, 23 September 1999, $\sim 327,600,000$\$, [NAS99]

Any compositional quantifiable resource may be accounted for, including *worst-case execution time*. However, in this thesis we will mainly focus on *heap space* memory for the clarity of the demonstration. Running out of memory usually brings a software program to a full stop and therefore causes a total loss of the (sub-)system. The amount of available memory is still an important cost factor for small embedded devices. Even for other systems, for which memory has recently become relatively cheap, it is important to know that the finite amount of available memory is not exhausted due to space leaks or other problems. Being able to manage and address an overly-large amount of memory also increases the overall complexity of the system. Hence, a large number of software faults are due to poor memory management on behalf of the programmer [Joy96]. Other scenarios where memory consumption of a program is a concern include cloud computing, smart cards, or software for mobile devices. For example, software offered by download for mobile phones must run on a multitude of devices, each running a different mix of software. However, malfunctioning phones do not generate any revenue, but will cause expensive hotline traffic and device returns instead. Therefore it is desirable to have guaranteed bounds on the memory requirement of a program prior to its runtime, at the design stage.

1.1 Contributions

We present a novel program analysis technique for determining upper-bound functions on the use of quantitative resources. The technique is type-based and applicable to all kinds of typed programming paradigms. We demonstrate the analysis for strictly evaluated functional programs. The important hallmarks of our analysis are as follows.

a) The proposed analysis is *fully automatic*, i.e. no expert knowledge is required to take advantage of the method.

b) The analysis is entirely *static*, i.e. it delivers bounds at compile-time, which are valid for all imaginable runs of the program on any arbitrary input data.

c) The approach is *fully formal*, i.e. the soundness proofs in this thesis guarantee the cost bounds are always observed.

d) The analysed language is *fully featured*, and does not restrict a programmer to unnatural or inexpressive language constructs. In particular, the analysed language allows:

i) *higher-order* types,

ii) *full (mutual) recursion*,

iii) *nested* recursive data types,

iv) *suspension* of evaluation, and

v) *aliasing* of data.

e) The produced bounds are *data-dependent*, which increases precision by providing custom bounds for easily determinable classes of input data.

f) The analysis is *highly efficient* and can thus be performed within seconds, allowing frequent and plentiful use.

g) The inferred bounds hold also for *non-terminating* programs.

h) The analysis is fully *generic*, i.e. it can be used to derive bounds for any compositional quantifiable resource, such as *heap space* usage, *stack space* usage or even *worst-case execution time*.

The key aspect of this thesis is that it deals directly with higher-order functions without requiring source-level transformations that could alter resource usage, unlike other fully automatic systems proposed for performing a guaranteed resource usage analysis.

This thesis also has already contributed to the academic research conducted in the field of program analysis through the following publications:

a) [HJ03] bounds the heap space usage of first-order functional programs;

b) [HJ06] bounds the heap space usage of object-oriented imperative JAVA programs;

c) [JLS+09] demonstrates applicability to worst-case execution time;

d) [JLH+09] bounds arbitrary resources, especially worst-case execution time, and formalises treatment of user-defined recursive data types;

e) [LJ09] treats practical aspects of the technique from a user-perspective; and

f) [JLHH10] extends analysis to both polymorphic and higher-order types.

Furthermore, the research generated by this thesis played an important role in three notable research projects: Mobile Resource Guarantees [MRG05], EmBounded [EmB09], and Islay [Isl11].

1.2 Thesis Overview

In this thesis we describe an automated analysis technique for programs written in a standard functional programming language, that produces formally guaranteed data-dependent bounds on the heap space usage.

What is meant by data-dependent is best explained by an example. Consider some function that inserts a node into a given red-black tree. Red-black trees are a common textbook [Oka98] example for balanced binary search trees, whose nodes are coloured either red or black in order to help to efficiently maintain a rough balancing. How this mechanism works is not important for this example. What is important, is that the worst-case heap space consumption of the node insertion algorithm depends on these colours. The analysis is able to recognise this dependency and is able to bound the heap space usage of the red-black tree insertion function from above by the formula

$$10R + 28B + 20 \text{ heap cell units}$$

where R and B are the respective numbers of red and black nodes in the input tree into which the new element is to be inserted. So we learn that the insertion is generally more expensive for a tree having many black nodes.[C]

However, general input dependent bounds on resource usage are only useful if they easily allow one to distinguish large classes of inputs of roughly the same resource usage. Consider having a black box for a program that can compute the precise execution cost for any particular input. Even if this black box computes very fast, one still needs to examine all inputs one by one in order to determine the worst case or to establish an overview of the general cost behaviour. Since the number of concrete inputs may be large or even infinite, this is generally infeasible. In contrast, we can guarantee that the cost bounds produced by our automated analysis technique are always simple. Namely, the bounds are always *linear* in different input sizes, so we can easily learn the overall cost behaviour for all possible inputs.

An intuitive overview of the proposed technique and its theoretical underpinnings is first given in Chapter 2.

Chapter 3 yields an overview over related work that is also concerned with quantitative resource analysis, both our own, which was mainly based on this thesis, and that

[C] The cost bound for red-black tree insertion was taken from [LJ09]. The bound is with respect to a boxed memory model without deallocation. The analysed code is also given in that paper.

of others. We also discuss future work in this chapter already, in Section 3.2. The reason being that the work of other researchers on some of the possible extensions to this thesis has already begun.

A simplified first-order version of the strict functional programming language to be analysed, which features *full recursion* and *recursive data types*, is defined in Chapter 4. The language is called LF for historic reasons; the name is not an acronym, in particular the L does not stand for "linear". We present a standard monomorphic non-linear type system for the first-order language in Section 4.2, which will form a basis for the construction of the analysis. We will extend this language to include higher-order types in one of the later chapters.

The evaluation of the first-order language is formally described by a standard big-step semantics in Section 4.3. The model deliberately allows freedom for its implementation in various ways, according to the desired target machine. It also makes no assumptions on the allocation mechanism, which is modelled by a simple non-deterministic choice. The operational model thus abstracts away many low-level implementation details, such as memory fragmentation or the reanimation of stale pointers through subsequent allocations. We discuss these issues in Section 4.3.1, but consider them to be orthogonal to our primary concern, the counting heap space usage.

The language is then illustrated by the discussion of four informative program examples in Section 4.4, which show how aliasing and deallocation are handled.

Thus far, Chapter 4 stated fairly conventional basics. This changes with its last Section 4.5, which shows how to augment the previously stated big-step semantics in order to track the allocation and deallocation of heap space, allowing us to *measure* the cost of an execution sequence. This shows how our operational model focuses on the pure counting of heap space usage. We count heap space usage in terms of atomic heap cells, which may represent single bytes or entire blocks of memory. This choice is up to the user, according to the desired target machine, as our model is formulated variable enough.

An analysis for the first-order fragment of the language is then detailed and proven sound in Chapter 5. The analysis is formulated as a type system in Section 5.1, referred to as $\mathrm{LF}_\diamond$, which uses annotated types and imposes a set of arithmetic side conditions on these annotations. Removing these side conditions leaves just the standard monomorphic type system of LF, which had thus been introduced a basis for the construction of the analysis in Section 4.2 before. The only peculiarity of

the type system is the use of explicit structural rules, in order to track the possible introduction of aliasing later in the annotated type system that makes up the analysis. Note that the presence of the structural contraction rule does imply that the type system is *not* linear.

What happens to be linear are the arithmetic side conditions imposed by our analysis. The generated linear programming problem (LP) can thus be efficiently solved by a standard LP-solver. The numbers returned by the solver can then be easily interpreted to produce a data-dependent bound on the heap space usage of the analysed program, as discussed in Section 5.2.

The main result of the chapter is then a proof that the result of the analysis is always sound, stated in Section 5.3. We prove that for each program evaluation that has unlimited memory resources available, there exists an evaluation leading to the same result value that only uses at most as many heap cells as predicted by our analysis. We discuss the inferred results of the analysis for the four introductory program examples in Section 5.5 and then demonstrate its inference mechanism step-by-step using a fifth program example in Section 5.6.

Notice that the aforementioned theorem only makes claims about terminating programs, since non-terminating computations do not have a result value. However, it is also important to know whether non-terminating programs evaluate with a finite memory, such as an airbag controller that continuously processes sensor data. We therefore prove in Section 5.4 that even non-terminating programs cannot exceed the heap space bounds that have been inferred through our analysis in any arbitrarily large, but finite, amount of time.

Chapter 6 then extends the previously studied language by *higher-order* functions and *delayed evaluation*; followed by the non-trivial proof of soundness in Section 6.6 again. The structure of that chapter is otherwise largely similar to the structure of the preceding chapter, albeit being more dense and omitting an unannotated version of both the type system and the operational semantics. The higher-order language and type system are referred to by the name ARTHUR.

We then discuss the implementation of our technique and issues related to the implementation, such as analysing for the usage of resources other than heap memory, e.g. worst-case execution time, in Chapter 7. Afterwards we examine some more complicated program examples for various cost metrics, not only heap usage, in Chapter 8 and eventually conclude in Chapter 9.

1.3 Mathematical Preliminaries

We denote the non-negative rational numbers by $\mathbb{Q}^+$. We put $\mathbb{D} = \mathbb{R}_0^+ \cup \{\infty\}$, i.e., the set of non-negative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}_0^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$. If U is a subset of $\mathbb{D}$ we write $\sum U$ for the (possibly infinite) sum over all its elements. Since $\mathbb{D}$ contains no negative numbers, questions of ordering and non-absolute convergence do not play a role; any subset of $\mathbb{D}$ has a sum, perhaps ∞. We write $\sum_{i \in I} x_i$ for $\sum \{x_i \mid i \in I\}$ and use other familiar notations.

The ordinary union of sets is denoted by $\cup$, the disjoint union by $\dot{\cup}$ and the multiset union by $\uplus$, in order to distinguish between them. For a binary relation $\sim$, we may abbreviate the sentence $\forall x.\, x \in A \Rightarrow x \sim z$ by simply writing $A \sim z$. For example, we write $\{a, b\} \notin A$ if neither a nor b are elements of set A.

For sets of linear inequalities, we write $\psi \vDash \phi$ to denote that ψ entails ϕ, i.e. that all solutions to ψ are also solutions to ϕ. A solution or *valuation* $\boldsymbol{v}$ to a set of linear inequalities is a mapping, that maps all variables in the constraint set to non-negative rational values such that all equations in the constraint set are satisfied, written $\boldsymbol{v} \vDash \phi$.

For partial maps, we use the following notations when used to model stacks, heaps, or typing contexts: Let f be a partial map. The domain, co-domain and image of f are denoted by $\mathrm{dom}(f)$, $\mathrm{codom}(f)$ and $\mathrm{img}(f)$ respectively. We specify the domain and co-domain by writing $f : \mathrm{dom}(f) \rightharpoonup \mathrm{codom}(f)$. We may abbreviate $x \in \mathrm{dom}(f)$ by $x \in f$ and sometimes $f(x)$ by writing f_x. Furthermore we denote by $f[x \mapsto y]$ the partial map that sends x to y and acts like f otherwise. In the opposite way, $f \backslash x$ denotes the map which is undefined for x and otherwise acts like f. The restriction of f on $\mathcal{X}$ is written $f \restriction \mathcal{X}$, i.e. the map that acts like f for all $x \in \mathcal{X}$ and that is undefined otherwise. The empty map $\varnothing$ is often omitted, i.e. $[x \mapsto y]$ denotes the singleton map sending x to y. The notation $f \subseteq g$ is a convenient shorthand for $\forall x \in \mathrm{dom}(f).\, x \in \mathrm{dom}(g) \wedge f(x) = g(x)$, i.e. g is an *extension* of f.

Typing contexts, which are also partial maps, are traditionally written in a different notation, e.g. the context assigning variable x the type A is simply written $\{x{:}A\}$. We may even omit the curly braces for smaller contexts. For two contexts Γ and Δ having disjoint domains we simply write Γ, Δ to denote the joint context that maps the variables of Γ and Δ accordingly, e.g. $\{x{:}A, y{:}B, z{:}C\}$ denotes the context that maps the variables x, y, z to the types A, B, C respectively. Γ, Δ is undefined if the intersection of their domains is not empty. Note that the order of writing for

a context is without significance, e.g. $\{x{:}A, y{:}B, z{:}C\} = \{z{:}C, y{:}B, x{:}A\}$. The empty context is written $\{\,\}$.

The *free variables* of a term t are denoted by $\mathrm{FV}(t)$, which will be formally defined for the terms of our programming languages in Definitions 4.3 & 6.1. We define capture-avoiding substitution in the standard way and write $t[r/x]$ to denote the term that is obtained by replacing all free occurrences of variable x within term t with the term r, where all bound variables are suitably renamed to avoid the capture of free variables in r. We generally adopt the Barendregt convention [Bar85] and assume that all bound variables are always suitably fresh. We also assume that the sets of variables (or identifiers) are generally infinite, but countable.

We write $[\,]$ to denote the empty list and $[a, b, c]$ for the list containing the elements a, b and c. The concatenation of two lists is written $l_1 +\!\!+\, l_2$. If h is a suitable element and t is a (possibly empty) list, then $h :: t$ is the list obtained by prepending h to t; while $t +\!\!+ [h]$ denotes the list obtained by attaching element h at the end of list t. The cardinality of a list l is denoted by $|l|$, e.g. $|[a, b, c]| = 3$. We identify each list with its own index map, i.e. if $l = [a, b, c]$ then $l_2 = b$ according to our previously introduced abbreviations. The expression l_5 is undefined in this example. We may sometimes write $\vec{l}$ for a list l to enhance readability.

For readability within program examples, we sometimes allow ourselves to replace free variables which are not of importance by the underscore symbol '_', in reseblance to practical conventions found in popular functional programming languages, such as Haskell and SML. Multiple occurrences of this symbol stand for different unnamed variables as usual, hence they are not connected with each other in any way.

2 Intuitive Description of the Proposed Technique

We explain our basic approach informally by using an analogy to emissions trading in the real world: One might consider heap space to be a valuable resource that must not be wasted recklessly. Like burning fossil fuels for energy, which generates the greenhouse gas CO_2 as a byproduct, we may consider the littering of heap memory with garbage as an unwanted "emission" of a program. As attempted in the real world, we will reign in the amount of undesired "emissions" that are produced by decreeing that each heap space allocation must be justified by spending a valuable "carbon credit". Since these "carbon credits" cannot be generated out of thin air, a program must ask in advance (i.e. through its input) for enough credits to complete its run. Furthermore these credits must also be handed down to subroutines that the main program subcontracts do part of its work, so that their use of heap space is justified just as well. A program may also be rewarded for responsible use of heap space by receiving extra "carbon credits" for each heap space deallocation, i.e. for each "emission" that is undone.

The benefit of such a system is, as in the real world, that the total amount of "emissions" produced is *a priori* bounded by the number of "carbon credits" that have been issued. A program can thus never allocate more heap space than the amount it asked for at its very start. However as opposed to real world bureaucracy, our control system incurs a zero overhead: ensuring that each unit of heap allocation is matched by expending a "carbon credit" is achieved through an augmented typing derivation for the program's source code. Whether the typing of a program is correct or not can be checked readily at compile time. The code itself and its actual execution remains unchanged. Another huge advantage over the real world is that we can and will rigorously prove that each and every emission is actually registered, in other words, that no heap allocation escapes our analysis.

2.1 Amortised Analysis

The mathematical foundation behind the principle informally described above is known as *amortised analysis*, first described by R. E. Tarjan in [Tar85]. Amortised analysis is a general tool to determine the complexity of an algorithm, be it time or space. A good introduction to amortised analysis with respect to functional programming was written by C. Okasaki [Oka98]. We will further remark on Okasaki's work towards the end of this section.

In an amortised analysis, each state that an algorithm may enter during its process is abstracted into a single number, called the *potential*. The *amortised cost* of an

operation I performed by the algorithm is then defined as its *actual cost* (time, space, etc.) plus the *difference in potential* of the state before and after performing the operation.

$$\textit{amortised cost}(I) \;\geq\; \text{actual cost}(I) \;-\; \text{potential before}(I) \;+\; \text{potential after}(I)$$

If the method of assigning a potential to a state is cleverly chosen, then the amortised cost of each individual operation may be zero or at least constant (or can be suitably bounded by a constant), even when their actual cost varies in relation to the state and is thus difficult to determine. This usually means that expensive operations lower the potential accordingly, while others, especially resource-beneficial operations such as deallocations, may increase the potential.

Determining the cost of an entire sequence of operations then becomes very easy: the total cost is the sum of the amortised costs plus the difference in potential of the state at the start of the operation sequence and at the very end. Calculating the sum of all amortised costs can often be made even easier by choosing the potential in such a way that all individual amortised costs are zero. In that case, the sum is simply zero, or in other words, all actual costs are fully amortised by the initial potential.

One usually ensures also that the potential is always non-negative. This is not always necessary, but it has the following two advantages:

a) First, in the presence of operations having a negative actual cost, such as heap space deallocation, one must ensure that the *high watermark* of resource usage is also tracked. For example, an allocation of two bytes which is followed immediately by the deallocation of two bytes, has a net cost of zero, but still requires two freely available bytes to succeed. If the potential is somehow not allowed ever to become negative, then such temporary costs will be revealed by the increased starting potential that is required.

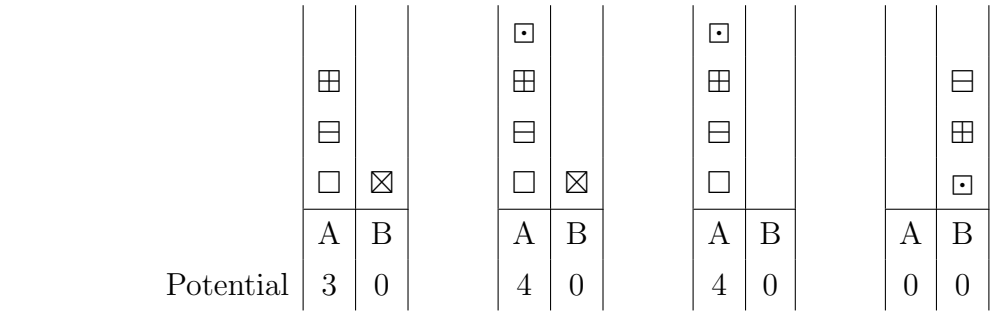
b) Second, even if the final state is completely unknown, the overall worst-case cost of all operations can be bounded from above by the potential of the initial state alone, since the worst-case must assume that the final state has a potential of zero. Of course, this requires all amortised costs to be non-positive as well, since otherwise we would still need to track an upper bound on the actual number of each possible operation, which is unlikely if we do not know the final state.

Note that the latter situation applies to the analogy from the introduction: each carbon credit represents a potential of one and the number of carbon credits in circulation is obviously always non-negative. Furthermore each emission has an amortised cost of zero, since each emission must be paid for exactly by expending a number of carbon credits. According to a.), one might also consider awarding newly generated carbon credits to companies for consuming CO_2, without affecting the overall amount of emissions generated in the observed time period.

If both properties are not required, i.e. the final state is known and all actual costs, such as execution time costs, are non-negative, then a negative potential can be safely allowed. This allows costs also to be expressed in relation to the final state, if its potential happens to be negative, rather than to the initial state alone. Recall that the overall cost is the sum of the amortised costs for each operation performed, plus the difference in potential between the initial state and the final state, both of which can be negative. However, since heap space operations may have negative actual costs, we will ensure that the potential is always non-negative for the remainder of this thesis.

2.2 Example: Analysing a Queue

The following standard example is a simplified version of an example provided in Okasaki's book [Oka98]. Okasaki himself cites R. Hood and R. Melville [HM81] and D. Gries [Gri81] as the source for this example. It demonstrates the usefulness of an amortised analysis by examining an implementation of a queue using two stacks A and B. Enqueueing is performed on stack A, dequeueing is performed on B. If a dequeueing request is received while B is empty, the whole contents of A are first moved to B, thereby reversing their order, prior to dequeueing from B eventually. A sequence of example operations on such a simulated queue is shown in Figure 2.1 on the facing page. The costs shown in the figure are the number of elementary stack operations that are required to perform the requested queue operation, which could correspond to the execution time. It is easy to see that the enqueueing operation `enqu`$(\cdot)$ has a cost of one; while dequeueing with operation `dequ`() has a variable cost, which is often one, but which is sometimes proportional to the size of A. This variable cost is unpleasant, since calculating the cost of a sequence of queue operations requires us to keep track of the state of stack A somehow.



Operation	$\overrightarrow{\texttt{enqu}(\boxdot)}$	$\overrightarrow{\texttt{dequ}() = \boxtimes}$	$\overrightarrow{\texttt{dequ}() = \square}$
Actual Cost	1	1	5
Change of Potential	1	0	-4
Amortised Cost	2	1	1

Figure 2.1: Simulating a queue by using two stacks

However, if we decree that the potential is exactly the size of A, then enqueueing always has an amortised cost of two (one for the actual cost, one for the increase in potential) and dequeueing has always an amortised cost of one, since the cost of moving A over to (the empty stack) B cancels exactly out against the decrease in potential. It is obvious that the potential, which is equal to the size of stack A, is always non-negative. Thus, the actual cost of a sequence of operations is bounded by the initial size of A plus twice the number of enqueues plus the number of dequeues. Therefore we only need to know about the initial state, but we do not have to track the full state for determining the cost of an entire sequence of queue operations.

In this particular example, we can verify this analysis by observing that each element is moved exactly three times: once into A, once from A to B, and once out of B. Handling each element therefore has a cost of at most three units. The amortised analysis recognises this. In effect, the cost of moving an element from A to B is accounted for earlier in time by attributing it to the enqueueing operation rather than the later dequeueing operation, where the cost actually comes into effect. Note that the amortised analysis is precise for the entire sequence. In general, however, the accumulated amortised costs at any point during a sequence present a worst-case bound on the actual costs.

2.3 Automatising the Amortised Analysis

The crucial task in applying the amortised analysis technique is to carefully assign the potential in a useful way. One has to be clever to do this. This restricts the technique to experts and prevents a widespread use, since it is infeasible to try all possible ways of assigning potential. We overcome this problem by choosing the following restriction:

The potential is always chosen to be a linear combination of the number of nodes required to store a program's current data structures in the (heap) memory.

This implies that the upper bound on worst-case cost is also linearly tied to the input data structures, since the bound depends on the potential of the initial state. This is the price we have to pay for an automated amortised analysis at the press of a button.

The benefit is that we can construct a type based analysis, which is only concerned with the entities directly mentioned in the context of each program expression, as opposed to traditional amortised analysis, which always takes into account the entire memory configuration.

In the above queue example, both stacks have clearly the same type. However, the stacks must be treated differently, since each element of A contributes 1 to the overall potential, whereas each element of B contributes a potential of 0. We choose to record this information also within the type, by adding a number annotation to each type constructor: stack A is assigned the type "Stack(Shapes,1)", while stack B is assigned the type "Stack(Shapes,0)". Therefore we can easily compute the potential of a data structure if we know its assigned type.

The total potential is then a sum of the potential of each data structure based on its type, plus an independent amount referred to as *petty potential* that is not tied to any data, i.e. the constant term of the linear potential formula. It is crucial that this sum ranges over all typings within the current scope. Hence, one and the same data structure may contribute more than once to the potential, if any aliasing occurs. We comment more on the treatment of aliasing in the next section.

The purpose of our annotated type system is then to ensure that each operation has an amortised cost of zero. Each type rule will therefore pose certain (linear)

constraints on the potential annotations within the types of the entities that are manipulated by an expression. An amortised cost of zero is achieved by using the petty potential. Its role is similar to petty cash, while the potential assigned to a data structure based on a type is treated more like money sitting in a savings account, that cannot be used directly for making payments. For example, moving a single element from stack B to stack A has an amortised cost of two: one for the actual cost and another one for the increase in potential, which is saved for later. This saving is represented by the "typecast" of an element of "Stack(Shapes,0)" to fit the type of elements of "Stack(Shapes,1)". This cost of two must be paid from the "petty cash" potential, and this is enforced by the constraints of the type rule used for typing this operation.

Automating the entire process is then achieved by the following steps:

a) by constructing a standard typing derivation;

b) by annotating each occurring type, including intermediate ones, with a unique variable ranging over the rational numbers;

c) by constructing an annotated typing derivation, which closely follows the structure of the standard typing derivation, essentially adding constraints;

d) by collecting all constraints from the annotated typing derivation; and

e) by solving the gathered constraints using a standard solver for linear programming problems (LP).

The solution returned by the LP-solver, if it exists, then yields the numbers for the potential annotations for the argument types of a program. These in turn allow us to compute a program's heap space usage instantly through a simple linear arithmetic expression, which depends on the actual sizes of the concrete input for the program, such as the length of a list. The LP is deliberately not restricted to integral solutions, so that that we can infer and express bounds like, e.g. "three heap units required for processing two elements of the input list". We will manually perform the entire process for an informative program example in Section 5.6.

Note that the process can only fail at the first or the last step, i.e. the generation of the constraints always succeeds for a well-typed program. In order to increase the range of programs that admit such an annotated typing derivation, we also allow forfeiting of potential at various points. For example, if one branch of a conditional admits

the type "Stack(Shapes,3)" and the other "Stack(Shapes,1)", then the result will be simply restricted to type "Stack(Shapes,1)". Similarly we allow the dropping of assumptions from the typing context. In terms of the analogy from the introduction this means that we may discard carbon credits at any time without using them. This is harmless in the sense that we still have a valid bound on the total amount of "emissions", but it may be an over-approximation. We note that one might also use over-approximations for the actual costs when dealing with resources other than heap space, for example attributing the worst-case number of clock cycles required to perform an operation as the actual cost for all such operations.

2.4 Advances over Previous Amortised Analysis Techniques

In his book [Oka98], Okasaki observes that amortised analysis generally breaks in the presence of persistent data structures commonly used in functional programming. The problem is that Okasaki assigns the potential to the data structures as they exist in the memory. Thus potential only decreases by deallocation. Therefore, if a persistent data structure is dereferenced more than once, potential is wrongly duplicated each time that data is processed.

His solution to this dilemma is to resort to lazy evaluation, in the hope that data is only processed once, with the heap being updated to the result. This works well for a range of interesting algorithms. However, applying the amortised analysis technique to a lazy functional language gives rise to further interesting problems, which are subject to currently ongoing research [Sim10].

Our thesis focuses entirely on a strict functional language, mainly because it seemed more promising to deliver accurate bounds. We were initially not aware of the problem described by Okasaki (and Amortised Analysis in general) and thus circumvented it unintentionally.

We avoid this problem by assigning the potential to one and the same data structure on a per-reference basis. Therefore each use of a data structure is assigned its own individual and independent potential.

The price for this is that we need an explicit contraction rule in our type system, which governs the multiplication of references. A type system is usually oblivious

to data aliasing and we keep it that way. Aliases are not tracked nor counted, only their introduction is noted. If a reference must be duplicated, then our explicit contraction rule ensures that the existing potential of the referenced data structure is split linearly among the newly created aliased references to that data structure. This guarantees that the overall potential remains unchanged by aliasing. For example, a reference of type "Stack(Shapes,9)" can be exchanged for two references of type "Stack(Shapes,6)" and "Stack(Shapes,3)" which can then be used independently, despite the persistence of the data structure on the heap.

An important and useful consequence of assigning potential on a per-reference basis is that aliased data is then assigned the same amount of potential as unaliased data. This is useful, since in a purely functional language it is impossible for the consumer of data to recognise aliasing within the input, therefore its execution cost should not depend on aliasing. Of course, the producer of data may still reduce its execution costs through aliasing, as we shall see later on.

3 Related Research

In this chapter we provide a survey over the family of research concerned with program analysis. We first start by examining directly related research, which are all concerned with the amortised analysis technique. The second section then lists the expected future members of that family, which are either already in the making or still waiting to be conceived. At last, we consider and compare other attempts at program analysis for estimating the resource usage of programs.

3.1 Affiliated Research

During the very long time that it took to accomplish this thesis, several publications were spawned, which we now discuss within their historical context. In fact, there is little left in this thesis that has not been already published somewhere. However, these publications would have never happened without this thesis. While the research was exclusively conducted for this thesis, we believe it is highly beneficial to the research community to learn about our findings as soon as possible, and to integrate it into large research projects.

The author contributed a significant amount of this thesis towards the success of the publications [HJ03, HJ06, JLS+09, JLH+09, LJ09, JLHH10]. An overview over the different features contained in each of these publications is given in Figure 3.1. Providing a uniform access to this body of work in a common revised notation is also an important feature of this thesis. Although not all available extensions to the automated amortised analysis technique could be included in the formal framework of this thesis, we discuss and explain most features informally, except for the object-oriented imperative language of [HJ06], which we excluded since it does not deal with a purely functional language.

The author also wrote several deliverables for research projects. Both, the participation in research projects and the above mentioned publications, helped the amortised analysis and its implementation to reach a sound maturity.

Mobile Resource Guarantees [MRG05] The research project Mobile Resource Guarantees was primarily concerned with equipping mobile byte code with machine-checkable certificates, that can be independently verified by the receiver of the code. The author provided an implementation of the automatic amortised analysis to determine bounds on heap space usage as a role model for a property to be encoded in these certificates. An overview that discusses the amortised analysis in this context can be found in [LMJB09].

Feature	[Hof00]	[HJ03]	[HJ06]	[JLH+09]	[LJ09]	[JLHH10]	This Thesis
full recursion	+	+	+	+	+	+	+
in-place update	+	+					
compilation to C	+	+					
inference		+		+	+	+	+
aliasing		+	+	+	+	+	+
object-orientation			+				
imperative update			+				
storeless semantics			+	+		+	+
arbitrary recursive data types				+	+	+	
varying resource metrics				+	+	+	
interactive solving					+		
polymorphism						+	
creating circular data						+	
resource parametricity						+	+
higher-order						+	+
non-termination							+
delayed execution							+

Figure 3.1: Family feature comparison

EmBounded [EmB09] The amortised analysis technique as researched by the author played the leading role in the EmBounded project, which aims "to identify, to quantify and to certify resource-bounded code in Hume, a domain-specific high-level programming language for real-time embedded systems" [HM03]. We detail the cooperation with the EmBounded project further below.

Islay [Isl11] The Islay project on "adaptive hardware systems with novel algorithmic design and guaranteed resource bounds" follows a goal similar to the EmBounded project, albeit for software that is executed on reconfigurable and possibly adaptive FPGAs.

The line of research that lead to this thesis was started with "A type system for bounded space and functional in-place update" by Hofmann [Hof00], which describes a linearly typed first-order functional language, called LFPL, that can be compiled into malloc-free C-code. The system included a special type $\diamond$, which represented pointers to free memory regions, making them a first class object inside the functional language. The $\diamond$-type later evolved into our abstract notion of potential, essentially by abstracting it into numbers that count its occurrences for certain pivotal places in a type. The paper already discussed several possible extensions, including polymorphism, higher-order functions and arbitrary recursive data types. Hofmann also observed in [Hof02] that adding higher-order functions to LFPL increases its expressive power from linear space to exponential time.

The work on this thesis was started by the task of constructing a type *inference* for LFPL, which led to the publications [Jos02, HJ03]. The foundations for the resource analysis laid in these works remained largely unchanged: linear inequalities arising from the side conditions of a type derivation are solved by an LP-solver, whose solution yields the factors for a formula that bounds the heap usage of the typed program. Unlike for LFPL, the considered type system included a rule for *contraction*. The paper proved that the generated linear programs always admitted an integral solution, if they were solvable at all. This is unusual, since determining whether or not a linear program admits an integral solution is known to be an NP-hard problem. However, we also showed by an encoding of 3SAT into an LFPL program that obtaining an *optimal* solution remained to be a NP-hard problem.

The following work [HJ06] extended the method to successfully analyse a subset of Java, named RAJA. The object-oriented language included *inheritance*, *up- and downcast* and *imperative updates*. However, a full inference for RAJA is lacking, albeit an automatic type-checking algorithm was recently provided by Hofmann and Rodriguez [HR09]. A combination of the amortised analysis for Java with separation

logic was proposed by Atkey [Atk10], in order to include the treatment of safety aspects in conjunction with imperative update.

Note that it was during the preparation of [HJ06] that our colleague Olha Shkaravska pointed out the connection to Amortised Analysis as proposed by Tarjan [Tar85], of which we had been hitherto unaware. Together with employing a storeless semantics, which allowed the abolishment of the unfortunate "benign sharing" from [HJ03], this led to a much cleaner description of the method.

Extending the technique to deal with a higher-order type system turned out to be difficult. The first description of a workable type system is found in the later deliverables of the EU-funded research project EmBounded [EmB09]. However, the Hume language used in the EmBounded project differs notably from the Arthur language treated in this thesis. Hume uses *under-application* to create function closures, but has no general lambda abstraction nor delayed execution [HM03]. Tracking the size of closures therefore required the technique of *closure-chaining*. There are numerous further technical differences, since Hume is a programming language that is geared towards everyday use, as opposed to the theoretically convenient Arthur. The practical approach required several extensions, especially since the EmBounded project placed a high emphasis on the determination of *worst-case execution time* (WCET), the first results of the our technique being shown in [HBH+07].

The researched extensions published in [JLH+09] notably included the formal treatment of *user-defined recursive data types*, which we informally discuss in Section 5.2.1 and illustrate by an example in Section 5.6. Even more important, the work showed how to generalise the soundness proof in order to have *user-defined resource metrics* as independent plug-ins, briefly discussed in Section 7.1. Metrics for worst-case execution time, heap and stack space usage were included and applied to several program examples, yielding quite precise cost bounds for each. The work thereby showed that our technique is indeed good enough to lift an elaborate machine-level WCET analysis, such as AbsInt GmbH's **aiT** tool [FHL+01], that works on small blocks of C-code, which represent byte code instructions of an intermediate language used in the compilation process, up to the high-level source language. The work thereby confirms earlier experiments with real-time embedded applications reported in [JLS+09].

Integrating our research with the EmBounded project had the advantage of producing a robust implementation of the analysis. Some of the technical aspects of this implementation were examined in [LJ09], such as *interactively* performing incremental calls to the LP-solver in order to enable the user to explore the different cost

bounds a program might simultaneously admit. Another novelty of that work was to demonstrate a cost metric that counts the maximum *number of calls* made to a selectable set of function identifiers.

Brian Campbell published extensions to the technique to infer more precise bounds for stack-space usage [Cam08, Cam09], loosely based upon a very early draft of this thesis. It was shown how to account for *temporary usage* of potential, as well as assigning potential with respect to *depth* and *maxima* of data. These techniques lend themselves much more naturally to the way stack space is used and thereby improve the precision of the cost bounds.

The soundness proof for a *higher-order* version of the amortised analysis technique was then eventually published in [JLHH10]. The considered language is similar to the one in this thesis, but additionally allows *polymorphism* and the creation of *circular* data structures. The work also included polymorphism with respect to resource annotations, named *resource parametricity.* which we consider to be a must-have for a successful analysis of higher-order functions. Comments on the simple technique of using *ghost-let* constructs, which are vital for measuring time and stack usage with a big-step semantics, were also included in this work. Compared with this thesis, only the analysis of *non-terminating* programs and *delayed execution* were missing.

3.2 Future Offspring

Although the amortised analysis technique detailed in this thesis has reached a remarkable maturity, there are still several possibilities for future investigations, which might be rewarded with useful results.

Super-linear bounds. The first way to surmount the limitation to linear cost bounds, was investigated by Shkaravska et al. [Svv07], which allowed the use of resource functions within the constraints for shapely functions only, but the inference appeared to be quite hard.

A more promising approach by Hoffmann and Hofmann [HH10b] lies in representing resource bounds as non-negative linear combinations of binomial coefficients. However, this method appears to be limited to list- and tree-like data structure, although such data structures are indeed the most common ones.

We believe that a breakthrough might be achieved through a combination with sized type techniques [Vas08, VH04, HP99, HPS96]. A combination of both approach would be most useful, since our experience shows that sized type techniques have good chance of succeeding where the amortised analysis has problems and vice versa. In addition, even partial information provided by the sized type analysis can be used to generate super-linear bounds through the amortised analysis. For example, it is possible to recharge the potential of a list if a bound to its length is known. The benefit of such an approach would be that successive application of efficient LP-solving would suffice.

Lazy evaluation. With the advent of many-core processors upon us, it appears that side-effect free functional languages offer a tremendous advantage: first, by eliminating the complexity that side-effects incur in a parallel setting [HM00, MPJS09], and second, by introducing parallelism automatically [HS07]. Hence, the importance of functional languages is likely to rise in the future. The lazily evaluated HASKELL language is among the most popular functional languages of the current times. However, all the languages treated by our technique thus far were assumed to be eagerly evaluated, with the notable exception of Okasaki's work [Oka98], which excludes information. So it would already be highly desirable to analyse a single thread which lazily evaluated on its own.

Of course, worst-case bounds for a strictly evaluated language remain valid if we switch the evaluation strategy to be lazy, but those bounds would be of very poor quality. There are many programs that require an infinite amount of resource when evaluated strictly (due to non-termination), but which only require a small finite amount of resources under lazy evaluation. For example, the simple program `take 5 (repeat 7)`, that computes a list containing the first five elements of an infinite list, would already not terminate under a strict evaluation strategy, but clearly does so when lazily evaluated. It is therefore natural to ask whether we can do better than simply performing a standard strictness analysis to reduce the program beforehand.

Note that the delayed execution included in ARTHUR is insufficient, since linear pairs are not allowed to be shared. The power of lazy evaluation lies within the sharing of a reference, albeit maintaining that only the first access to an alias requires any evaluation, while any subsequent access cheaply looks up the previous result.

However, it seems quite possible to fix this. First experiments are very promising and seem to be quite capable to deal with infinite stream generators. The

technique appears to call for a mixture of our per-reference and Okasaki's per-location potential. Further research by Simões et al. [Sim10] is already on its way, and we expect exciting results in the nearby future.

Negative Potential. A golden rule for amortised analysis is that potential is always non-negative. This is necessary to track a temporary maximum resource usage. However, monotonic resources such as worst-case execution time cannot exhibit temporary highs, and hence possibly allow us to ignore this rule. While a positive potential accounts cost earlier than they actually occur, a negative potential would delay the point in time when costs are accounted for. Costs could then be expressed in terms of the input *and* output sizes. Of course, this would require a strict type system, that ensures that types with negative potential cannot be unlawfully discarded. It would be interesting to learn how this technique could increase the range of programs that can be successfully analysed and whether there is a further impact on the aforementioned analysis of lazily evaluated languages.

Lower bounds. Sized type based analysis systems usually need to track sizes in ranges in order to work, i.e. both maximal and minimal sizes are accounted for. Thus far we have only examined worst-case cost, but it seems straightforward to analyse for best-case costs as well, thereby inferring cost ranges. In addition to help the integration with sized type approaches, minimal costs might also be of interest for lazily evaluated languages.

Ghost-data. Amortised analysis is strong when branches of computation closely correspond to the processed data structures, such as recursion over a list or a tree. However, this is not always the case and the amortised analysis often performs poorly if the recursion does not match the structure of the data. It might be possible to salvage these cases by assigning potential to inferred imaginative data structures, that capture the recursion scheme, thereby turning it explicit. The idea is that the analysis always produces an answer, that may perhaps provide more information about resource usage than a simple "fail".

Numeric potential. The general idea in our amortised analysis is that every constructor carries its own potential annotation. However, numbers were neglected, since one could view the terms $\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots$ as constructors. A difficulty lies in the fact that it is in practice impossible to add annotations for each number contained in a numeric type. As an approximation we experimented with assigning the number 0 a zero potential and the number r exactly r-times the potential of the number 1. This corresponds to viewing `0` and `succ` as the only

constructors of a number value, and is equivalent to treating a number as a list of unit elements of the corresponding length. Of course, non-negative fractional values can clearly be assigned a potential in the same manner, i.e. value times annotation.

We found that the applications studied during the EmBounded project often required this experimental extension of numeric potential in order to deal with loops that are bounded by numeric computations. However, this extension is unsafe, as negative potential might occur (see above) and must be manually checked for, which is tedious. It suffices to check that all number values that carry non-zero potential are non-negative, such that the analogy of treating numbers similarly to unit lists can be upheld, i.e. that the list length is always non-negative. Furthermore, assigning the number 0 a zero potential is rather arbitrary and only works well for loops that count down to zero. Therefore it should be investigated whether this could be generalised, e.g. numeric types given two annotations for basis and step, and whether such a generalisation can be successfully inferred.

3.3 Distant Relatives

In this section we review several works by other authors that are also concerned with *quantitative resource analysis*. The section is largely based on our review of related work presented in [JLHH10], albeit some citations of works that are not directly concerned with quantitative resource analysis were moved to more appropriate sections of this thesis (e.g. work concerned with deallocation safety is discussed in Section 4.3.1).

Note that a direct comparison based on experiments between our amortised analysis approach and the reviewed works is lacking, since many of the approaches discussed in this section are experimental and may be concerned with different programming languages having different operational cost models. Instead, we rely on the evaluations of the authors given in their respective works.

We furthermore concentrate on the comparison of general features provided by each approach, which we summarised in Figure 3.2 on the next page for convenience. Each feature receives a rating, where a "+" symbol signifies that a feature was achieved,

	Analysable Programs	Recursion	Polymorphism	Programming Paradigm	Higher-Order Functions	Nested Data Structures	Automatic Inference	Efficiency	Implementation	Heap Space	Stack Space	Time
Amortised Analysis	○	+	+	Any	+	+	+	+	+	+	+	+
This Thesis	○	+	-	Fun	+	+	+	+	○	+	-	-
[HPS96, HP99]	-	+	+	Fun	+	-	-	-	○	+	+	-
[CK01, CNPQ08]	-	+	-	Imp	-	-	+	-	+	+	+	-
[VH04]	○	+	+	Fun	+	-	+	○	+	+	+	+
[Vas08]	○	+	+	Fun	-	-	+	○	+	+	+	-
[FHL+01]	○	-	-	Imp	-	-	○	○	+	-	-	+
[BCC+03]	○	-	+	Imp	-	-	○	-	+	-	-	-
[Mét88]	-	○	-	Fun	-	-	+	+	-	-	-	+
[Ben01]	○	+	-	Fun	○	-	+	○	○	-	-	+
[GL02]	+	+	+	Fun	+	+	+	-	+	-	-	+
[AAG+07, AGG09]	+	+	○	Imp	-	○	○	○	+	+	-	+
[GMC09]	+	○	-	Imp	-	+	○	○	+	-	-	+
[San90]	+	+	-	Fun	+	-	-	-	-	-	-	+
[CW00]	○	+	+	Fun	+	+	-	+	+	○	○	+
[TEX03]	+	+	-	Fun	+	+	+	+	-	+	-	-
[BFGY08]	○	-	+	Imp	○	○	+	+	+	+	+	-

Figure 3.2: Comparison of related work

while a "−" symbol stands for a feature that was not or only poorly achieved. The "○" symbol stands for considered or partially achieved features. For example, this thesis receives an "○" for its range of analysable programs, since it can only successfully analyse programs having a linear resource consumption, which is a noteworthy restriction. Likewise, its implementation receives an "○" since it uses a somewhat different syntax and lacks the support for linear pairs of Arthur.

It is noteworthy that many approaches to statically determine the runtime cost of a program are type based. Types are useful for resource analyses, because they can relate additional resource relevant information along the dataflow of a program, or they can provide a sound basis for simplifying the analysis problem through varying levels of abstraction.

Sized types. Sized types express bounds on the sizes of data structures. The sizes are attached as annotations to types in a similar fashion as our potential annotations, but the two should not be confused. The difference is that sized types express *absolute* upper bounds on the possible size of data structures (and often a lower bound as well), whereas our annotations are weight factors of a linear resource bound that are *relative* to the size. Sized types are thus a special form of *dependent types*, where the type are parametrised with a value. This allows a more fine grained control over the values that are associated with the type.

A significant problem for sized type approaches are nested data structures, since the size bounds of sub-lists of a list are then required to be uniform, which may lead to unacceptable imprecision. Note that this is not at all a problem for our amortised analysis, as the potential is assigned according to the individual sizes of nested structures.

The groundbreaking work for the sized type analysis technique was written by Hughes, Pareto and Sabry [HPS96], who described a *type checking* algorithm for a simple higher-order, polymorphic, non-strict functional language to determine progress in a reactive system. This work was subsequently developed to describe space usage in Embedded ML [HP99], a strict functional language using regions to control memory usage. However, their work only dealt with linear data structures, like lists, but not trees, and the final analysis was not implemented.

Chin and Khoo [CK01] introduced a type inference algorithm that is capable of computing size information from high-level program source. Chin et al. [CNPQ08] then presented a heap and a stack analysis for a low-level assembly language with explicit

(de-)allocation, which is also restricted to linear bounds. The system can automatically insert deallocation primitives when sufficient alias information is provided. By inferring path-sensitive information and using symbolic evaluation they are able to infer heap and stack space bounds.

Vasconcelos and Hammond developed automatic inferences for a sized type analysis that are capable of deriving cost equations for abstract time- and heap-consumption from unannotated program source expressions based on the inference of sized types for recursive, polymorphic, and higher-order programs [VH04]. Vasconcelos' PhD thesis [Vas08] extended these previous approaches by using abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. By including user-defined sizes, it is possible to infer sizes for algorithms on non-linear data structures, such as binary trees, thereby improving upon the previous works in this line of research. The runtime of the inference is exponential in the program size. The thesis also corrects an error in the soundness proof provided by Chin and Khoo [CK01].

The two following works discussed are not quantitative resource analyses per se, but they employ dependent types and methods that are relevant to this discussion. Danielsson [Dan08] introduced a library of functions that makes the complexity analysis of a number of purely functional data structures and algorithms almost fully formal. He does this by using a dependent type system to encode information about execution time, and then by combining individual costs into an overall cost using an annotated monad. However, the system requires some manual cost annotations. Abel [Abe06] extended higher-order sized types to allow higher-kinded types with embedded function spaces. He used this system to formalise termination checking but did not consider resource consumption in general. One should note that a strict upper bound on resource consumption does not always imply termination. For example, one may easily write a non-terminating loop that has a net heap space usage of zero. Vice versa, one may expect specialised termination checkers, such as the system proposed by Abel, to deliver an answer for programs whose quantitative resource bounds cannot be recognised.

Abstract Interpretation. The basic idea of abstract interpretation is to simplify and omit all calculations that do not directly influence the execution costs. For example, instead of computing a numeric value, it is only computed whether it is positive or negative, thereby speeding up the execution of the program to be examined. While having the attraction of being very general, one major disadvantage

of abstract interpretations is that analysis results usually depend on concrete data input values to perform the abstracted runs of the program. Unlike type-based approaches, abstraction interpretations also often risk non-termination in analysing recursive definitions.

Where they can be applied, impressive results can, however, be obtained even for large commercial applications. For example, AbsInt's **aiT** tool [FHL+01], and Cousot et al.'s ASTREE system [BCC+03] have both been deployed in the design of the software of Airbus Industrie's Airbus A380. Typically, such tools are limited to non-recursive programs and may require significant programmer effort to use effectively. Note that the ASTREE system is concerned with general software verification and does not compute quantitative resource bounds, whereas the **aiT** tool is focused on determining worst-case execution time.

Considering user-defined higher-order programs appears to be quite difficult. The work by Le Métayer [Mét88] can handle predefined higher-order functions with known costs, such as standard `map`/`fold` functions. However, the system is restricted to a single linear data structure and non-mutual recursion.

Benzinger's work [Ben01] on worst-case complexity analysis for programs extracted through the higher-order proof assistant NuPrl similarly supports higher-order functions if the complexity information is provided explicitly, but cannot infer such information by itself.

Gómez and Liu [GL02] have constructed an abstract interpretation for determining time bounds on higher-order programs. Their analysis executes an abstract version of the program that calculates cost parameters, but which otherwise mirrors the normal program execution strategy. Unlike our type-based analysis, the cost of this analysis therefore depends directly on the complexity (or actual values) of the input data and the number of iterations that are performed, does not give a general cost metric for all possible inputs, and will even fail to terminate when applied to non-terminating programs.

The COSTA system by Albert et al. [AAG+07, AGG09] performs a fully automatic resource analysis for an object-oriented bytecode language, and is also largely generic over the resource being analysed. First a set of recurrence relations are generated, which are then solved by Mathematica or a special recurrence solver that is tailored for the use of resource analysis and as such produces better results than general recurrence solvers. Similar to the amortised analysis technique it produces a closed-form

upper bound function over the size of the input. Unlike our system, however, data-dependencies cannot be expressed and the accuracy of the bounds remains unclear, especially in the case of function composition. Furthermore, higher-order functions cannot be treated directly.

A powerful, recent analysis which demonstrated high quality bounds for product code is Gulwani, Mehra and Chilimbi's SPEED system [GMC09]. The system uses a symbolic evaluation approach to calculate non-linear complexity bounds for C/C++ procedures using an abstract interpretation-based invariant generation tool. Numerical loop bounds can be automatically inferred, but loops iterating over data structures require manually produced numerical bounds. In general, their strength lies in dealing with programs performing many numerical computations, while their weakness lies in dealing programs that recurse over inductive data types. Their work is thus complementary to the one presented in this thesis. However, unlike this thesis, they target only first-order programs and they only consider bounds on worst-case execution time. They do, however, consider non-linear bounds and disjunctive combination of cost information.

Further Approaches. The interesting work by Sands [San90] is distinguished by dealing with a higher-order functional language that is lazily evaluated, a topic that we are highly interested in as noted in the previous section. The approach relies on an additional neededness analysis and quickly becomes complex for more elaborate data structures. The work focuses on time analysis and was intended to aid human reasoning. However, the approach appears unsuited for constructing a fully automatic inference.

A simple system that resembles our notion of petty potential, similar to a clock winding down, was proposed by Crary and Weirich [CW00]. The work allows user-defined primitive-recursive cost functions rather than fixing a canonical cost function like we do. The cost functions are intended to limit computational steps, but Crary and Weirich suggest that imposing bounds space usage is also possible, and especially straightforward for stack space. Their system is implemented as a type-checker for a higher-order type system. However, the system only checks given bounds, but is not able to infer them.

Taha, Ellner and Xi [TEX03] present the multi-staged language GeHB. First stage computations are allowed to have arbitrary resource consumption, while second stage computations may not allocate any new heap memory. GeHB programs automatically generate first-order LFPL programs from higher-order specifications, which is

an interesting feature. The system thereby offers more expressivity than LFPL while reusing the mechanics developed for LFPL, but it does not deliver a cost formula for the heap space consumption of the first stage.

Braberman et al. [BFGY08] infer polynomial bounds on the live heap usage for a Java-like language with an automatic region-based garbage collector for memory management. The system appears to efficiently compute bounds on heap and stack space consumption, but cannot cover general recursive methods.

4 Basis Language LF

We define an eager first-order monomorphic functional language, called LF, as a basis for our compile-time program resource usage analysis. The language is fairly standard and *not* linearly typed, despite what the acronym may suggest. The name was chosen in reminiscence of Hofmann's LFPL [Hof00], which inspired the research that eventually led to this thesis.

We first formulate a standard type system and operational semantics as a basis for our work. We then formulate a cost-aware extension to the type-system of LF, which we refer to by the name $\mathrm{LF}_\diamond$, in Chapter 5. The little precious diamond symbol representing resource-awareness is also derived from [Hof00], albeit $\mathrm{LF}_\diamond$ does not posses a specific resource type. The type system for $\mathrm{LF}_\diamond$ allows us to bound the heap space usage of typed terms. We generally distinguish between unannotated and annotated (or cost-aware) by adding the $\diamond$-symbol the name of the various rules.

We will extend $\mathrm{LF}_\diamond$ furthermore in Chapter 6 by higher-order functions, resource parametricity and delayed execution through linear pairs.

4.1 Syntax of LF

The terms of LF are given in Figure 4.1, consisting of terms for the unit and boolean constants, variables and let bindings, function calls, conditionals, additive pairs, sums and lists respectively.

The pattern matches for Sums and Lists must be specified as either read-only or destructive: using the term `match` only reads the heap memory without altering it, whereas using `match!` on an entity causes it to be deallocated immediately after the match. There is no destructive match for pairs since they will be stack allocated in our operational model, unless the pair is ultimately contained in a sum or a list node. Including a deallocation primitive allows programs to exhibit a much more interesting resource usage. It seemed natural to us to pair deallocation with pattern matching in a purely functional language. A general deallocation primitive could have been easily treated by our analysis, since any syntactic construct can be assigned an arbitrary positive or negative cost. This is used to employ the amortised analysis method for stack space usage, as demonstrated in [Cam09, JLH+09], since stack space is affected by almost all operations. More comments about our choices for memory handling follow in Section 4.3.

$e ::=$	`*` | `true` | `false`	Constant
	| x | `let` x `=` e_1 `in` e_2	Variable
	| f`(`x_1`,` $\ldots$ `,`x_n`)`	Function Call
	| `if` x `then` e_t `else` e_f	Conditional
	| `(`x_1`,` x_2`)` | `match` x `with` `(`x_1`,` x_2`)` $\rightarrow e$	Pair
	| `Inl(`x`)` | `Inr(`x`)`	Sum Construction
	| `match` x `with|Inl(`y`)` $\rightarrow e_l$ `|Inr(`z`)` $\rightarrow e_r$	Sum Access
	| `match!` x `with|Inl(`y`)` $\rightarrow e_l$ `|Inr(`z`)` $\rightarrow e_r$	Sum Elimination
	| `Nil` | `Cons(`x_1`,`x_2`)`	List Construction
	| `match` x `with|Nil` $\rightarrow e_1$ `|Cons(`x_1`,`x_2`)` $\rightarrow e_2$	List Access
	| `match!` x `with|Nil` $\rightarrow e_1$ `|Cons(`x_1`,`x_2`)` $\rightarrow e_2$	List Elimination

Figure 4.1: Syntax of the simple language LF

Further base types could have been added without any difficulties. Note that data types such as natural numbers or trees can already be build using pairs, sums and the unit constant as included. The inclusion of primitives for lists is therefore superfluous from the analysis perspective, but allows us to discuss interesting program examples in a more succinct manner. Directly including a fixed size type, such as integers, is analogous to the treatment of booleans. The direct inclusion of user-definable recursive data types is also possible, as also shown in [JLH+09], but requires a cumbersome notation and was hence avoided in this thesis. We will outline the treatment of arbitrary recursive data types in Section 5.2.1, once the basics of the amortised analysis have been explained. Our prototype implementation of the amortised analysis always allowed the use of arbitrary recursive data types, as there would have been no benefit for a non-general implementation. So for the sake of simplicity we restrict ourselves here to booleans, pairs, sums and lists. We may occasionally allow ourselves to use fixed size integers or tree types in some of the program examples without much further explanations.

4.1.1 Let-normal form.

LF was designed to allow maximum simplicity from a proof-theoretic viewpoint while covering all the essentials of a functional programming language. This is the reason why LF only allows terms in a relaxed *let-normal form* (also known as administrative normal form [FSDF93]), i.e. subexpressions are restricted to be variables in most cases. This might require the use of additional `let` expressions, e.g. writing `let` y `=` $g(x)$ `in` $f(y)$ instead of the more succinct program term $f(g(x))$. For another example, defining a pair of boolean constants also requires two `let` expressions, e.g. `let` x `=` **tt** `in let` y `=` **ff** `in (`x`,` y`)`.

The reason for this mild restriction, that is often considered as syntactic sugar, is that it removes boring redundancies from our proof obligations: sequential evaluation of expressions is dealt with precisely once – in the case that deals with the `let` expression. Without the let-normal form the proof steps for the `let` expression alone would have to be repeated in all cases that involve subexpressions, thus obscuring the essential new parts of the proof within each case.

Note that we do not need the full let-normal form for expressions that do not sequentially compose their subexpressions. For example, conditional and pattern match only require a variable for the entity that decides the branch to execute, but not for the subexpressions that define the branches themselves. Since only one of the branches

zero-order types:	$A ::= \mathsf{unit} \mid \mathsf{bool} \mid A \times A \mid A + A \mid \mathsf{list}(A)$
first-order types:	$F ::= (A, \ldots, A) \to A$

Figure 4.2: Types of the simple language LF

is executed, no threading is required, and hence we allow any subexpression in such positions, not just variables.

It is important to note that the transformation to let-normal form is harmless and commonly used in compilers. It has other pleasant side-effects, such as making the evaluation order explicitly visible in the code, thereby eliminating any ambiguity. However, this becomes only important for our amortised analysis technique when dealing with cost metrics that bound stack space or worst-case execution time, for example. Similarly, the *ghost-let* technique, discussed later in Section 7.2, is also unimportant for the heap space metric that is primarily considered in this thesis, so we ignore these for now.

4.2 Type System

The types of LF are shown in Figure 4.2, and consist of the unit type unit, the type bool for booleans, the product type $A \times A$, the sum type $A + A$, the type of lists over entries of type A, and a first-order type for functions.

We now formulate the standard typing rules for LF. The statement

$$\Sigma; \Gamma \vdash e : A$$

means that term e has the type A within the typing context Γ and the program signature Σ. A typing context is a partial mapping from variables to LF zero-order types, as already described in the mathematical preliminaries in Section 1.3. A signature Σ is a partial mapping from function identifiers to LF first-order function types; since the signature is fixed for a given program, for brevity, we usually omit to mention Σ throughout the rules.

Definition 4.1 (LF Type Rules)**.**

$$\frac{}{\{\,\} \vdash * : \mathsf{unit}} \quad (\text{LF}\vdash\textsc{Unit})$$

$$\frac{c \in \{\mathtt{true}, \mathtt{false}\}}{\{\,\} \vdash c : \mathsf{bool}} \quad (\text{LF}\vdash\textsc{Bool})$$

$$\frac{}{x{:}C \vdash x : C} \quad (\text{LF}\vdash\textsc{Var})$$

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2, x{:}A \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : C} \quad (\text{LF}\vdash\textsc{Let})$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_k) \to C}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash f(x_1, \ldots, x_k) : C} \quad (\text{LF}\vdash\textsc{Fun})$$

$$\frac{\Gamma \vdash e_t : C \qquad \Gamma \vdash e_f : C}{\Gamma, x{:}\mathsf{bool} \vdash \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : C} \quad (\text{LF}\vdash\textsc{If})$$

$$\frac{}{x_1{:}A, x_2{:}B \vdash (x_1\,,\, x_2) : A{\times}B} \quad (\text{LF}\vdash\textsc{Pair})$$

$$\frac{\Gamma, x_1{:}A, x_2{:}B \vdash e : C}{\Gamma, x{:}A{\times}B \vdash \mathtt{match}\, x\ \mathtt{with}\ (x_1\,,\, x_2) \to e : C} \quad (\text{LF}\vdash\text{E-}\textsc{Pair})$$

$$\frac{}{x{:}A \vdash \mathtt{Inl}(x) : A{+}B} \quad (\text{LF}\vdash\textsc{Inl})$$

$$\frac{}{x{:}B \vdash \mathtt{Inr}(x) : A{+}B} \quad (\text{LF}\vdash\textsc{Inr})$$

$$\frac{\Gamma, y{:}A \vdash e_l : C \qquad \Gamma, z{:}B \vdash e_r : C}{\Gamma, x{:}A{+}B \vdash \mathtt{match}\ x\ \mathtt{with}\ \mathtt{|Inl}(y) \to e_l\, \mathtt{|Inr}(z) \to e_r : C} \quad (\text{LF}\vdash\text{E-}\textsc{Sum})$$

$$\frac{\Gamma, y{:}A \vdash e_l : C \qquad \Gamma, z{:}B \vdash e_r : C}{\Gamma, x{:}A{+}B \vdash \mathtt{match!}\, x\ \mathtt{with}\ \mathtt{|Inl}(y) \to e_l\, \mathtt{|Inr}(z) \to e_r : C} \quad (\text{LF}\vdash\text{E!-}\textsc{Sum})$$

$$\frac{}{\{\,\} \vdash \mathtt{Nil} : \mathsf{list}(A)} \quad (\text{LF}\vdash\textsc{Nil})$$

$$\frac{}{x_h{:}A, x_t{:}\mathsf{list}(A) \vdash \mathtt{Cons}\,(x_h\,,x_t) : \mathsf{list}(A)} \quad (\mathrm{LF} \vdash \textsc{Cons})$$

$$\frac{\Gamma \vdash e_1 : C \qquad \Gamma, x_h{:}A, x_t{:}\mathsf{list}(A) \vdash e_2 : C}{\Gamma, x{:}\mathsf{list}(A) \vdash \mathtt{match}\ x\ \mathtt{with}\ \mathtt{|Nil} \to e_1\,\mathtt{|Cons}\,(x_h\,,x_t) \to e_2 : C} \quad (\mathrm{LF} \vdash \textsc{E-List})$$

$$\frac{\Gamma \vdash e_1 : C \qquad \Gamma, x_h{:}A, x_t{:}\mathsf{list}(A) \vdash e_2 : C}{\Gamma, x{:}\mathsf{list}(A) \vdash \mathtt{match!}\,x\ \mathtt{with}\ \mathtt{|Nil} \to e_1\,\mathtt{|Cons}\,(x_h\,,x_t) \to e_2 : C} \quad (\mathrm{LF} \vdash \textsc{E!-List})$$

Structural Rules

$$\frac{\Gamma \vdash e : C}{\Gamma, x{:}A \vdash e : C} \quad (\mathrm{LF} \vdash \textsc{Weak})$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash e : C \qquad \curlyvee(A\,|A_1, A_2\,)}{\Gamma, z{:}A \vdash e[z/x, z/y] : C} \quad (\mathrm{LF} \vdash \textsc{Share})$$

We see that LF type system is entirely standard, except for containing explicit structural type rules. Of the usual structural rules, only permutation is implicitly built-in due to our definition of typing contexts in Section 1.3, while weakening and contraction have been made explicit.

Please note that the presence of explicit structural type rules for weakening and contraction do *not* at all mean that the type system is linear. In order to avoid possible misunderstandings, we repeat the terminology here as given by Pierce [Pie04]: Once would obtain an *affine* type system if we removed contraction (LF$\vdash$Share); a *relevant* (or *strict*) type system if we removed weakening (LF$\vdash$Weak); or a *linear* type system if we removed both structural rules. However, we do not intend to remove either rule.

The reason for making weakening and contraction explicit will become apparent later, when we deal with the resource aware type system, since both structural rules are important for tracking resource usage. For an intuition about this, recall from the introduction that we intend to derive bounds on resource usage in relation to the sizes

of the data handled. Therefore, the application of the weakening rule (LF⊢Weak) may introduce over-approximation, since it allows discarding of requested data without looking at it; whereas contraction (LF⊢Share) allows data to be used twice, which must therefore adjust the weight of that data within the resource bound formula accordingly. Contraction is therefore defined by using a subsequent *sharing* relation $\curlyvee$, which is trivial for LF.

Definition 4.2 (LF Sharing)**.** Given the LF types A, B and C, the ternary relation $\curlyvee(A\,|B, C)$ holds if and only if both $A = B$ and $A = C$ hold.

Note once more that explicit sharing (or contraction) does *not* enforce a linear typing discipline. On the contrary, multiple uses of a variable and hence the creation of aliased data is explicitly allowed, but in order to track resource usage accurately we must ensure that our type system is aware of such multiple uses, since our analysis will be encoded within the type system. The sharing relation $\curlyvee$ will be extended to the annotated type system $\mathrm{LF}_\diamond$ of Chapter 5 by Definition 5.4. The sharing relation will then govern the type annotations only, with the prerequisite that the underlying LF types of all three types in relation are always equal, entailing the above definition for LF-types.

For completeness we remark that the third standard structural rule, permutation, is admissible due to our definition of permutable typing contexts in Section 1.3. We decided not to mention permutation explicitly, since the order in which data has been provided has no impact on resource usage. This is a useful fact to notice, since the amortised analysis just counts resources, but does not distinguish them otherwise.

$$\frac{\Gamma, y{:}B, x{:}A, \Delta \vdash e{:}C}{\Gamma, x{:}A, y{:}B, \Delta \vdash e{:}C} \qquad (\mathrm{LF}\vdash\textsc{Permute})$$

We conclude this section with the precise definition of a *well-typed* LF program, which in turn requires us to define the set of free variables contained in an LF term first. The definitions are all standard.

$$\mathrm{FV}(\texttt{*}) = \mathrm{FV}(\texttt{true}) = \mathrm{FV}(\texttt{false}) = \emptyset$$
$$\mathrm{FV}(x) = \{x\}$$
$$\mathrm{FV}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \mathrm{FV}(e_1) \cup \big(\mathrm{FV}(e_2) \setminus \{x\}\big)$$
$$\mathrm{FV}(f(x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\}$$
$$\mathrm{FV}(\texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f) = \{x\} \cup \mathrm{FV}(e_t) \cup \mathrm{FV}(e_f)$$
$$\mathrm{FV}(\texttt{(}x_1\texttt{,}\ x_2\texttt{)}) = \{x_1, x_2\}$$
$$\mathrm{FV}(\texttt{match}\,x\ \texttt{with}\ \texttt{(}x_1\texttt{,}\ x_2\texttt{)} \to e_2) = \{x\} \cup \big(\mathrm{FV}(e_2) \setminus \{x_1, x_2\}\big)$$
$$\mathrm{FV}(\texttt{Inl(}x\texttt{)}) = \mathrm{FV}(\texttt{Inr(}x\texttt{)}) = \{x\}$$
$$\mathrm{FV}(\texttt{match}\ x\ \texttt{with |Inl(}y\texttt{)} \to e_l\texttt{|Inr(}z\texttt{)} \to e_r)$$
$$= \{x\} \cup \big(\mathrm{FV}(e_l) \setminus \{y\}\big) \cup \big(\mathrm{FV}(e_r) \setminus \{z\}\big)$$
$$\mathrm{FV}(\texttt{match!}\,x\ \texttt{with |Inl(}y\texttt{)} \to e_l\texttt{|Inr(}z\texttt{)} \to e_r)$$
$$= \{x\} \cup \big(\mathrm{FV}(e_l) \setminus \{y\}\big) \cup \big(\mathrm{FV}(e_r) \setminus \{z\}\big)$$
$$\mathrm{FV}(\texttt{Nil}) = \emptyset$$
$$\mathrm{FV}(\texttt{Cons(}x_1\texttt{,}x_2\texttt{)}) = \{x_1, x_2\}$$
$$\mathrm{FV}(\texttt{match}\ x\ \texttt{with |Nil} \to e_1\texttt{|Cons(}x_1\texttt{,}x_2\texttt{)} \to e_2)$$
$$= \{x\} \cup \mathrm{FV}(e_1) \cup \big(\mathrm{FV}(e_2) \setminus \{x_1, x_2\}\big)$$
$$\mathrm{FV}(\texttt{match!}\,x\ \texttt{with |Nil} \to e_1\texttt{|Cons(}x_1\texttt{,}x_2\texttt{)} \to e_2)$$
$$= \{x\} \cup \mathrm{FV}(e_1) \cup \big(\mathrm{FV}(e_2) \setminus \{x_1, x_2\}\big)$$

Figure 4.3: Free variables of LF terms

Definition 4.3 (Free Variables of LF Terms)**.** The *free variables* of an LF term e are denoted by $\mathrm{FV}(e)$, and are defined in the usual way as shown in Figure 4.3 on the preceding page.

Note that the ordinary `let` expression of LF does not allow recursive definitions, hence $x \in \mathrm{FV}(\texttt{let}\ x\ \texttt{=}\ x\ \texttt{in}\ e)$ holds.

Definition 4.4 (Program)**.** A *program* $\mathcal{P}$ is a partial mapping from function identifiers f to a pair, consisting of an ordered set of variable names $(y_1, \ldots, y_k)$, and an LF term e_f, such that $\mathrm{FV}(e_f) \subseteq \{y_1, \ldots, y_k\}$. Such a pair is usually written $(y_1, \ldots, y_k \to e_f)$, and we say that e_f is *defining body* of f and that $(y_1, \ldots, y_k)$ are names of the arguments for the function.

Definition 4.5 (Well-Typed LF Program)**.** An LF program $\mathcal{P}$ is *well-typed* with respect to a LF type signature Σ, if and only if for all functions $f \in \mathcal{P}$ with $\Sigma(f) = (A_1, \ldots, A_k) \to C$ and $\mathcal{P}(f) = (y_1, \ldots, y_k \to e_f)$, the LF typing judgement $\Sigma; \{y_1{:}A_1, \ldots, y_k{:}A_k\} \vdash e_f : C$ can be derived by the LF type rules.

4.3 Operational Semantics

We shall now give meaning to LF terms by formally defining how they are evaluated. We will use a standard big-step semantics with a store that is split into a heap space and an (abstracted) stack space, so that the heap space bounds given by our analysis may be measured against this operational model.

Recall that LF allows the programmer to exert a limited control over memory management by explicitly specifying whether a pattern match should be destructive or read-only. This is of course dangerous, since the evaluation becomes stuck if a subsequent pattern match tries to access the data deleted by an earlier destructive match. However, we will not concern ourselves here with the safety of such deallocation primitives; nor do we want to encourage the introduction of side-effect laden constructs in functional programming languages. The reason for introducing a deallocation primitive is to demonstrate that our resource analysis can handle both the *spending* and the *regaining* of resources equally well. This would hardly be possible without an explicit deallocation primitive: The standard memory management method for most functional languages is to use a background garbage collection, which is a highly dynamic mechanism, whereas our analysis is purely static. Therefore, we would need

a static approximation of *a* garbage collector, which is already a tough problem on its own and outwith the scope of this thesis.

The safe insertion of deallocation primitives is an interesting problem which has been studied on its own, for example see the *usage aspects* by Aspinall and Hofmann [AH02] or the DEEL type system by Konečný [Kon03]. However, the problem is rather orthogonal to this thesis: our analysis not only yields reliable bounds if all deallocations are safe (Theorem 1), but we will also prove that these bounds are respected up to the point where the evaluation halts in error when trying to access previously deallocated data (Theorem 2). So the predicted bound is upheld in any case. The only possibility that could lead to a violation of the predicted bounds lies in the random reanimation of a dangling pointer due to a fresh allocation at the previously freed location with data of a matching type, that is then erroneously accessed through the old reference. However, such a scenario is already prevented by the previously mentioned techniques, since a safe deallocation entails that all possibly dangling pointers are not dereferenced anymore. Within our thesis we avoid this problem by the methods discussed in Section 4.3.1.

The exemplary inclusion of an unsafe deallocation primitive can also make sense in different settings for our general amortised analysis principle. For example, the basic principle remains the same when applying the method to stack space usage [Cam09, JLH+09], but static deallocation points for stack space are readily available.

It is also conceivable that some programmers would indeed relish the explicit control over deallocation, even if safety is not automatically guaranteed. For example in embedded systems design it is often desired to have precise control over resource handling. Last but not least, a programmer is not required to use the explicit deallocation primitives at all, at the expense of increased heap cost.

Definition 4.6 (Store and Values)**.** Let Loc be an arbitrary countable set of *locations*, which shall model memory addresses. We use ℓ to range over elements of Loc. Our store consists of two finite partial mappings: the stack (or environment) and the heap. A *stack* $\mathcal{S} : \mathsf{Var} \rightharpoonup \mathsf{Val}$ is a finite partial mapping from variables to values, and a *heap* $\mathcal{H} : \mathsf{Loc} \rightharpoonup \mathsf{Val}$ is a finite partial mapping from locations to values.

The *values* Val, ranged over by v and w, are defined by the following grammar:

$$v \quad ::= \quad () \mid \mathbf{tt} \mid \mathbf{ff} \mid (v_1, v_2) \mid \ell \mid \mathbf{Null} \mid \mathbf{Bad}$$

Hence a value is either a constant of a basic type (unit, true and false), a pair of consecutive values (v_1, v_2), a general pointer to a location $\ell \in \mathsf{Loc}$, a constant denoting the empty list, or the special constant **Bad**.

There is no particular constant location, especially $\{\mathbf{Null}, \mathbf{Bad}\} \notin \mathsf{Loc}$. The value **Bad** is intended as a special marker for deallocated values. The evaluation will simply get stuck as soon as a location pointing to **Bad** is dereferenced, since no operational rule will be applicable. We will comment further on its purpose after stating the operational rules. Note for now that the value **Bad** will usually not occur within tuple values nor the image of a stack. Any configuration where **Bad** occurs outside the image of the heap will be considered inconsistent.

We may occasionally refer to a value in the image of a stack as a stack value, and similarly to a value in the image of a heap as heap value. However, we do not impose such a distinction between values on their own. Notice that the stack may contain pointers into the heap (i.e. locations), but there are no pointers from the heap into the stack. The stack is extended with additional variable bindings whenever we enter a new scope, inside subterms in the premises of the evaluation rules. When we evaluate a function body we use a new stack which only mentions the actual arguments provided in the function call, thereby preventing access beyond the current stack frame.

The assumption that Loc is infinite avoids irrelevant technical problems, since we want to analyse programs that may have an arbitrarily large memory consumption. The sole purpose of the operational semantics defined in this section is to give meaning to LF terms, regardless of the amount of memory required for evaluation. The augmented operational semantics defined later in Section 4.5 will then enable us to express limited memory resources, so that we can then compare evaluation under restricted and unrestricted heap space availabilities.

Evaluation of an expression term e takes place with respect to a given stack and heap, and yields as its result a value and a possibly updated heap. Thus we have a relation of the form

$$\mathcal{P}; \mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$$

expressing that, within the context of the LF program $\mathcal{P}$, the evaluation of term e under initial stack $\mathcal{S}$ and heap $\mathcal{H}$ succeeds in a finite number of steps and results in value v and post-heap $\mathcal{H}'$. Since the program $\mathcal{P}$ does not change during the evaluation, we usually omit to explicitly state it within the premises of each rule. We shall continue our discussion about memory management after first stating the rules of our operational semantics.

Definition 4.7 (LF Operational Semantics)**.** The operational semantics is given with respect to a fixed program $\mathcal{P}$.

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash * \downarrow \texttt{()}, \mathcal{H}} \quad (\text{LF}\downarrow\textsc{Unit})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash \texttt{true} \downarrow \mathbf{tt}, \mathcal{H}} \quad (\text{LF}\downarrow\textsc{True})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash \texttt{false} \downarrow \mathbf{ff}, \mathcal{H}} \quad (\text{LF}\downarrow\textsc{False})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash x \downarrow \mathcal{S}(x), \mathcal{H}} \quad (\text{LF}\downarrow\textsc{Var})$$

$$\frac{\mathcal{S}, \mathcal{H} \vdash e_1 \downarrow v_0, \mathcal{H}_0 \qquad \mathcal{S}[x \mapsto v_0], \mathcal{H}_0 \vdash e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{let}\ \ x\ \texttt{=}\ e_1\ \ \texttt{in}\ \ e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow\textsc{Let})$$

$$\frac{\begin{array}{c}\mathcal{S}(x_i) = v_i \quad (\text{for } i = 1, \ldots, k) \\ \mathcal{P}(f) = (y_1, \ldots, y_k \to e_f) \qquad [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k], \mathcal{H} \vdash e_f \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash f(x_1, \ldots, x_k) \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow\textsc{Fun})$$

$$\frac{\mathcal{S}(x) = \mathbf{tt} \qquad \mathcal{S}, \mathcal{H} \vdash e_t \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow\textsc{If-T})$$

$$\frac{\mathcal{S}(x) = \mathbf{ff} \qquad \mathcal{S}, \mathcal{H} \vdash e_f \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow\textsc{If-F})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash \texttt{(}x_1\texttt{,}\ x_2\texttt{)} \downarrow \big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big), \mathcal{H}} \quad (\text{LF}\downarrow\textsc{Pair})$$

$$\frac{\mathcal{S}(x) = (v_1, v_2) \qquad \mathcal{S}[x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \vdash e \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match}\, x\ \texttt{with}\ \texttt{(}x_1\texttt{,}\ x_2\texttt{)} \to e \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow\textsc{E-Pair})$$

$$\frac{w = \big(\mathbf{tt}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash \texttt{Inl(}x\texttt{)} \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\text{LF}\downarrow\textsc{Inl})$$

$$\frac{w = \big(\mathbf{ff}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash \texttt{Inr(}x\texttt{)} \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\text{LF}\downarrow\textsc{Inr})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \qquad \mathcal{S}[y \mapsto w'], \mathcal{H} \vdash e_l \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l \texttt{|Inr(}z\texttt{)} \to e_r \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E-Inl})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \qquad \mathcal{S}[y \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash e_l \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match!}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l \texttt{|Inr(}z\texttt{)} \to e_r \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E!-Inl})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \qquad \mathcal{S}[z \mapsto w'], \mathcal{H} \vdash e_r \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l \texttt{|Inr(}z\texttt{)} \to e_r \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E-Inr})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \qquad \mathcal{S}[z \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash e_r \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match!}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l \texttt{|Inr(}z\texttt{)} \to e_r \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E!-Inr})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash \texttt{Nil} \downarrow \mathbf{Null}, \mathcal{H}}\ (\text{LF}\downarrow\text{Nil})$$

$$\frac{w = \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash \texttt{Cons(}x_h\texttt{,}x_t\texttt{)} \downarrow \ell, \mathcal{H}[\ell \mapsto w]}\ (\text{LF}\downarrow\text{Cons})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E-Nil})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match!}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E!-Nil})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \qquad \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H} \vdash e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E-Cons})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \qquad \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash \texttt{match!}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'}\ (\text{LF}\downarrow\text{E!-Cons})$$

4.3.1 Storeless Semantics

While our operational semantics may seem fairly standard at first glance, we have essentially adopted what is known in literature as *storeless semantics* [RBR+05, BIL03, Deu94, Jon81], which abstracts away from two important aspects of memory management: fragmentation and the accidental "reanimation" of stale pointers through recycling of previously issued locations. These issues present interesting and difficult problems on their own, but they are independent from our problem of bounding resource usage. Hence, our suggested strategy is to rely on existing approaches to deal with them independently. The remainder of this subsection is devoted to explain the two issues and approaches to solve them in more detail.

Fragmentation. We see that allocation of heap space (rules LF $\downarrow$INL, LF $\downarrow$INR and LF $\downarrow$CONS) expects a currently unused location $\ell \notin \mathrm{dom}(\mathcal{H})$ and stores the new value at this fresh location ℓ. The choice is non-deterministic. This is in accordance to common implementations, where a primitive is called that returns a pointer to a currently unused block of memory of the appropriate size. For example, the machine could keep a simple list of pointers to free memory blocks, a *freelist*, from which pointers are taken at allocation and to which pointers are returned upon deallocation. Such a freelist-based model was explicitly used in [HJ03]. Notice that since $\mathcal{H}$ is a finite partial mapping and since Loc is assumed to be infinite, there is always an unused location available. As mentioned before, we do not wish the evaluation to become stuck due to limited memory resources here, since the operational semantics are only used to measure memory usage.

However, the important abstraction here is that we ignore possible runtime issues of memory fragmentation, depending of course on the actual method of memory management chosen. All unused locations are treated as equal and may serve as the starting address for storing a value of any finite size. So, for example, if there are exactly three heap cells left, then we can always allocate a continuous block for a value of size three, although in reality, this might require defragmentation. We deliberately want to devise a general, quantitative compile-time memory analysis without confining ourselves to any specific method of runtime memory management.

In order to deal with fragmentation in the freelist memory model there are several known possibilities that interact smoothly with the resource counting we require in the abstract semantics: allocating all data nodes within blocks of the same size;

allocating data within linked lists of equally sized blocks; maintaining several independent freelists for data nodes of various sizes (a slight change to the typing rules is then required to prevent trading data nodes of different sizes against each other), and, finally, compacting garbage collection. In the last case, we would simply run a compacting garbage collection as soon as the freelist does no longer contain a contiguous portion of the required size. All of these strategies would incur some overhead, but at most twice the predicted amount is already sufficient, for example by relying on a two-space compacting garbage collector.

Reanimation of Stale Pointers. In connection with deallocation arises another problem with the allocation mechanism. Deallocation allows the reuse of memory blocks. Since we allow aliased data structures in LF, there may still be other pointers to a deallocated memory block, which then become "stale". Such stale pointers are not a problem unless they are dereferenced. We would expect a runtime error whenever a stale pointer is dereferenced, since obviously the use of an earlier deallocation primitive had not been safe.

However, the crucial property here is that any subsequent allocation does not by chance write data of a matching type to a reused location still pointed to by a stale pointer. In this case the error would go unnoticed even if runtime type checking would be used. Thus it is likely that the program will continue to evaluate using the wrong data of the correct type. Note that this property must also be guaranteed by any conservative garbage collector, usually achieved through conservative reference counting, since any memory management method whose allocation primitive may return memory addresses which may still be accessed through preexisting old pointers is clearly quite risky. While this problem is also orthogonal to bounding memory usage, we cannot defer the solution to this problem like we did for the defragmentation issue, since continuing the evaluation with wrong data might lead to a different memory usage, which we cannot tolerate. We cannot even hope to predict memory usage once such a random error has occurred.

A solution to counter this problem of randomly recycled locations would be to statically reject all programs that might somehow access stale pointers at all, for example using the *alias types* proposed by Walker and Morrisett [WM01], or the *bunched implication logic* as practised by Ishtiaq and O'Hearn [IO01]. Ensuring that dangling pointers are never created in the first place is another possibility, an approach pursued by Peña, Segura and Montenegro [MPS08, PSM06]. A combination of amortised techniques and *separation logic* might also deal with such issues, as the work by Atkey [Atk10] shows.

However, these interesting and sophisticated techniques are not the focus of this thesis. Hence we adopt a much simpler alternative solution, namely to employ indirect pointers (symbolic handles), as used by earlier implementations of the Sun JVM for its compacting garbage collector. Allocations then return a pointer to be used by the program, which itself points to another pointer, eventually leading to the allocated memory block. While the first pointer may occur in several places through aliasing, the second remains unique, since it is hidden from the program itself. Therefore a memory block may be safely reused upon deallocation, which exchange the second pointer by a null pointer. The pointers visible to the program itself are unique numbers that are never reused. They simply become useless, since they just continue to lead to a null pointer after deallocation. The address range thus appears to the program to be much bigger than the actual memory available, but any attempts to dereference such a pointer are guaranteed to lead to a runtime error. The drawback of this approach is the increased time that it takes to resolve the indirection when it is dereferenced.

Our operational semantics only simulates the approach of using indirect pointers. Deallocations (rules LF$\downarrow$E-Inl. LF$\downarrow$E-Inr and LF$\downarrow$E-Cons) will not remove a previously used location from the heap, but will simply overwrite it with the special value **Bad**, which is defined to have zero size. This prevents their reuse through subsequent allocations, since the allocation primitive must always return fresh locations $\ell \notin \mathrm{dom}(\mathcal{H})$. In some sense our operational semantics only deals with the symbolic handles, whereas the true pointers are omitted. We do allow "stale pointers", i.e. pointers to **Bad**, to be kept around, but any attempt to access such a stale pointer is guaranteed to causes the evaluation to become stuck, due to the lack of an applicable rule. Of course, this simple approach cannot guarantee that no dangling pointer is dereferenced, unlike the more sophisticated approaches mentioned above. However, since any such attempt causes the abortion of the evaluation, our inferred resource bounds still hold.

We have thus clearly separated the security problem from our analysis problem. Note that the somewhat unwieldy invariant of "benign sharing" used in our earlier work [HJ03] has now been replaced by the more transparent use of a storeless semantics. We find that the abstract operational semantics used here provides an adequate modular interface between the resource analysis and concrete implementation issues that have been and are being researched elsewhere. As a consequence, we can formulate the following convenient lemma:

Lemma 4.8 (Heap Stability)**.** *If* $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ *then for all* $\ell \in \mathrm{dom}(\mathcal{H})$ *we have either* $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ *or* $\mathcal{H}'(\ell) = \mathbf{Bad}$*. Furthermore we also have* $\mathrm{dom}(\mathcal{H}) \subseteq \mathrm{dom}(\mathcal{H}')$.

Proof. An inspection of the operational rules easily reveals that any existing location can only be altered by overwriting with the special constant **Bad**, whereas newly created values are always bound to fresh locations not already contained in the domain of heap $\mathcal{H}$. The statement $\mathrm{dom}(\mathcal{H}) \subseteq \mathrm{dom}(\mathcal{H}')$ is a trivial consequence of the first claim. □

4.3.2 Memory Consistency

We now formalise whether a given stack and heap agree with a certain typing context. This ternary relation is later used as an invariant in the inductive proof of the main theorem. The invariant trivially holds at the very beginning of the program evaluation, when stack, heap and typing context are empty. It expresses that each single evaluation step alters the memory in a sane fashion, provided that the configuration was consistent with a certain typing before that step was taken. Such a notion of consistency between heap, stack and typing context is needed to prevent the application of Theorem 1 to ill-formed hypothetical configurations, which could not have been obtained through the evaluation of a sane (well-typed) program.

Definition 4.9 (Consistency of Heap, Stack and Typing Context)**.** Let $\mathcal{H}$ be a heap, v a value and A an LF type. If the statement $\mathcal{H} \vDash v{:}A$ can be derived by the following rules, we say that the heap $\mathcal{H}$ is consistent with value v being of type A.

$$\frac{}{\mathcal{H} \vDash ():\mathsf{unit}}$$

$$\frac{}{\mathcal{H} \vDash \mathbf{tt}:\mathsf{bool}}$$

$$\frac{}{\mathcal{H} \vDash \mathbf{ff}:\mathsf{bool}}$$

$$\frac{\mathcal{H} \vDash v{:}A \qquad \mathcal{H} \vDash w{:}B}{\mathcal{H} \vDash (v,w){:}A{\times}B}$$

$$\frac{\mathcal{H}(\ell) = (\mathbf{tt}, v) \qquad \mathcal{H}\backslash\ell \vDash v{:}A}{\mathcal{H} \vDash \ell{:}A{+}B}$$

$$\frac{\mathcal{H}(\ell) = (\mathbf{ff}, v) \qquad \mathcal{H}\backslash\ell \vDash v{:}B}{\mathcal{H} \vDash \ell{:}A{+}B}$$

$$\frac{\mathcal{H}(\ell) = \mathbf{Bad}}{\mathcal{H} \vDash \ell{:}A{+}B}$$

$$\frac{}{\mathcal{H} \vDash \mathbf{Null}{:}\mathsf{list}(A)}$$

$$\frac{\mathcal{H}\backslash\ell \vDash \mathcal{H}(\ell){:}A{\times}\mathsf{list}(A)}{\mathcal{H} \vDash \ell{:}\mathsf{list}(A)}$$

$$\frac{\mathcal{H}(\ell) = \mathbf{Bad}}{\mathcal{H} \vDash \ell{:}\mathsf{list}(A)}$$

We extend this definition naturally to stacks and contexts by defining

$$\frac{\forall x \in \mathrm{dom}(\Gamma)\,.\,\mathcal{H} \vDash \mathcal{S}(x){:}\Gamma(x)}{\mathcal{H} \vDash \mathcal{S}{:}\Gamma}$$

Note that $\mathcal{H} \nvDash \mathbf{Bad}{:}A$, i.e. there exists no heap $\mathcal{H}$ and no type A that would be consistent with the value **Bad**. This implies that **Bad** cannot occur within the image of a consistent stack nor inside another value, such as a pair, sum or list node. However, it may well be that the location stored inside list node representing the tail of the list points to **Bad**.

The type associated with such a deallocated location, i.e. a location that is mapped to **Bad**, does not really matter, since any attempt to use such a location would cause the evaluation to become stuck immediately anyway. However, we require that the value **Bad** is only consistent with types whose values are generally stored in the heap, since only such values can ever be deallocated. This allows us to formulate the size correspondence Lemma 4.16 later in Section 4.5, although this lemma is just a convenience rather than a necessity, as described there. Therefore the value **Bad** is generally treated as consistent with any arbitrary type in our other works [JLH+09, JLHH10].

Note that memory consistency does not exclude degenerated configurations where a single deallocated location is associated with different types at once, although such a memory configuration cannot be generated through the execution of a well-typed program. We do not need to prohibit such hypothetical memory configurations,

since Theorem 1 shows that any well-typed program that would continue from such a strange memory configuration still respects its predicted bounds. Otherwise, the rules for memory consistency are straightforward, apart from a small peculiarity which we discuss next.

4.3.3 Circular Data Structures

The rules for sum and list types in the memory consistency Definition 4.9 on page 54 do not allow the location ℓ to be reused within their premises. It is important to note that $\mathcal{H}\backslash\ell$ is quite different from $\mathcal{H}[\ell \mapsto \mathbf{Bad}]$, since the latter allows ℓ to be consistent with list and sum types (i.e. any boxed type), while the former implies that ℓ is never consistent with any type. Of course, it is still possible that a location of a sum or list type occurs multiple times in the derivation for memory consistency, due to the branching of the derivation tree that occurs for pair types. Hence, the imposed requirement is only that the digraph representing the memory layout must be acyclic. Or in other words, aliased data is allowed as long as there are no infinite cycles, like a circular list. Program Example 4.4.1 will show us how acyclic aliased data can be created and analysed in LF, by constructing lists which share their tails.

The reason for excluding circular data structures in the definition of consistent memory configurations is simplicity: all derivations of memory consistency become finite. This is justified by the fact that circular data structures cannot be constructed within LF, due to the absence of both recursive let definitions and imperative update. Since memory consistency is meant to be an invariant for evaluation, we thus record the property that circular data cannot be produced therein.

Circular data structures are *not* a problem for the amortised analysis technique per se. We have already shown how to treat circular data that may either be created through imperative update in [HJ06], or through general recursive let-definition in [JLHH10]. Furthermore, the corresponding definition of memory consistency for the higher-order system in Chapter 6 does allow cyclic data structures. While our higher-order programs cannot create cyclic data structures either, the higher-order analysis requires a coinductive definition for memory consistency regardless, so there would be no gain in regarding memory configurations containing cyclic data as inconsistent.

We could hence equally allow cyclic data in LF at the expense of using a coinductive definition for memory consistency. Since allowing cyclic data is independent from the higher-order extension, let us briefly discuss the implications of dealing with cyclic

data at this point, as if it were allowed for LF, instead of deferring this discussion to Chapter 6 or later. Of course, we only have proof that the following observations hold for the higher-order system Arthur, but since LF is practically a subset of Arthur, we strongly believe them to be true anyway.

It is conceivable that an LF program receives circular data through its input. Recursion over a cyclic data is possibly infinite, so the resource consumption may suddenly become infinite as well. However, the program itself has not changed, and thus the result of the analysis still applies regardless of the input supplied.

Recall from Section 2.3 that our analysis will determine a rational number for each input, and that it guarantees that the resource usage during the execution of the program is smaller than the sum of all potentials, where the potential of a certain input is its inferred rational number multiplied by the number of corresponding data nodes that a reachable from the root. So a data node that can be reached in two different ways counts twice, while a node that is reachable in three different ways counts thrice and so on. This is necessary, since a functional program cannot distinguish between aliased data and duplicated data, so it matters how often a certain node can be reached through recursion.

Therefore it follows that the potential associated with cyclic data is necessarily either zero or infinite. In other words, the analysis can tell us that is safe to supply circular data for arguments whose annotated types indicate a potential of zero. Since the potential is the number of reachable nodes (infinite) times the annotation (zero), the overall potential for that data structure remains zero, thus the size of that input plays no role in the inferred resource bound and neither whether or not it is cyclic. Consequently, any well-typed program in systems that allow the creation of cyclic data will always enforce zero potential for potentially cyclic data that is newly generated. Note though, that [HJ06] relaxes this limitation somewhat by allowing potential to be ascribed to an initial acyclic fragment of a cyclic data structure.

On the other hand, supplying cyclic data in a position with a non-zero annotation causes the upper cost bound to become infinite. However, we can still obtain some useful information about *liveness* or *progress properties* of the program. For example, the analysis ensures that a certain amount of that circular input (or an input stream) must be processed by the program before it allocates more memory or requires more processor clock cycles. If each input node is associated with a potential of $\frac{1}{3}$, then the program can use one unit of a resource for every three input elements processed, whereas a potential of 5 would allow the program to spend 5 for each processed input element. So despite the overall potential being infinite in both cases, the presented analysis still delivers useful information.

4.3.4 Observations on Memory Consistency

We now record some simple facts about memory consistency. Note that the decoration * used with variables below has no specific meaning and just serves to distinguish between separate entities, similarly to using primed letters or adding a subscript numeric index. Since the latter two decorations will be used extensively in the proof of our main theorem, we introduce the decoration * here which is primarily intended to be used for the entities obtained through the application of lemmas, in order to reduce confusion.

The first lemma simply states that removing elements from the context is just as harmless as is adding arbitrary data to unused memory locations with respect to memory consistency.

Lemma 4.10 (Closure of Memory Consistency)**.** *For all $\mathcal{H}, \mathcal{H}^\star, \mathcal{S}, \mathcal{S}^\star, \Delta$ and Γ such that $\mathcal{H} \subseteq \mathcal{H}^\star$ and $\mathcal{S} \subseteq \mathcal{S}^\star$ and $\Delta \subseteq \Gamma$, i.e. $\mathcal{H}^\star, \mathcal{S}^\star$ and Γ are extensions of $\mathcal{H}, \mathcal{S}$ and Δ respectively. If $\mathcal{H} \vDash \mathcal{S}{:}\Gamma$ then also $\mathcal{H}^\star \vDash \mathcal{S}^\star{:}\Delta$ holds.*

Proof. The proof is trivial for all three extensions. Memory consistency is defined by quantifying over the domain of the context only, requiring the statement to be true for each member of the context's domain individually. Therefore deleting elements from the domain of the context cannot invalidate the statement. For the same reason we can also extend the stack, since it plays no further role other than serving a value for each member of the context. The stack is not used otherwise, so adding extra uninspected elements is harmless. For the extension of the heap, we observe that the statement $\mathcal{H} \vDash \mathcal{S}{:}\Gamma$ implies that all reachable locations are already defined in $\mathcal{H}$. Hence the extension to $\mathcal{H}^\star$ only adds locations that are never inspected by the statement $\mathcal{H}^\star \vDash \mathcal{S}^\star{:}\Delta$. Therefore adding these locations holding any value is again harmless. □

Memory configurations that do not overlap can be safely merged together.

Lemma 4.11 (Joint Consistency)**.** *Fix any heap $\mathcal{H}$, stacks $\mathcal{S}, \mathcal{S}^\star$ and contexts Δ, Γ such that $\mathrm{dom}(\mathcal{S}) \cap \mathrm{dom}(\mathcal{S}^\star) = \emptyset$ and $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Gamma) = \emptyset$, i.e. both stacks and the contexts are disjoint. If $\mathcal{H} \vDash \mathcal{S}{:}\Gamma$ and $\mathcal{H} \vDash \mathcal{S}^\star{:}\Delta$ then $\mathcal{H} \vDash \mathcal{S}, \mathcal{S}^\star{:}\Gamma, \Delta$*

Proof. Trivial, since the latter is by definition just the conjunction of the previous two statements. Recall that joining the partial maps by comma was only defined if the domains are disjoint. □

The "deallocation" of any location from a heap, i.e. replacing its value with the special tag **Bad**, is harmless for the memory consistency statement.

Lemma 4.12 (Consistency of Deallocation)**.** *If* $\mathcal{H} \vDash \mathcal{S}{:}\Gamma$ *then for all* $\ell \in \mathrm{dom}(\mathcal{H})$ *holds* $\mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vDash \mathcal{S}{:}\Gamma$

Proof. Fix any $x \in \mathrm{dom}(\Gamma)$. It suffices to show that $\mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vDash \mathcal{S}(x){:}\Gamma(x)$ follows from $\mathcal{H} \vDash \mathcal{S}(x){:}\Gamma(x)$.

The proof is by induction on the length of the derivation of the memory consistency relation $\vDash$. The cases for unit, booleans and empty lists are vacuously true, since the heap $\mathcal{H}$ is not inspected at all. The rule for pairs does not inspect the heap either, and hence follows directly by applying the induction hypothesis for each premise.

The only rules of Definition 4.9 which actually inspect the heap $\mathcal{H}$ are the rules for sums and lists. So in the remaining cases, $\mathcal{S}(x)$ is necessarily a location $\mathcal{k} \in \mathsf{Loc}$ and $\Gamma(x)$ is either a list or a sum type. If $\mathcal{k} \neq \ell$ then $\mathcal{H}(\mathcal{k}) = \mathcal{H}[\ell{\mapsto}\mathbf{Bad}](\mathcal{k})$ and the conclusion follows from applying the induction hypothesis to the corresponding premise of the memory consistency rule for either list or sum types. Otherwise, if $\mathcal{k} = \ell$, we have $\mathcal{H}[\ell{\mapsto}\mathbf{Bad}](\mathcal{k}) = \mathbf{Bad}$ and the conclusion follows by the consistency of deallocated locations with any list or sum type. □

The previous statement is intuitively clear, since any location pointing to the special constant **Bad** is defined to be consistent with any LF type whose values are stored in the heap deliberately. However, recall that the value **Bad** itself is never consistent with any type, since there is simply no rule to derive $\mathcal{H} \vDash \mathbf{Bad}{:}A$ for any type A in any heap $\mathcal{H}$.

Much more interesting is the following statement, that says that the heap obtained after evaluation is still consistent with the context and stack before the evaluation took place. This is a direct consequence of Lemma 4.8, i.e. that the values at heap locations are only ever altered by overwriting them with the constant **Bad**. Further consequences of this are formulated later as Lemma 5.8 and Lemma 5.9, which turn out be be quite useful in conducting the proof of theorem 1.

Lemma 4.13 (Preservation)**.** *Fix a well-typed* LF *program* $\mathcal{P}$ *with signature* Σ*. If*

$$\Gamma \vdash e{:}C \tag{4.1}$$
$$\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}' \tag{4.2}$$
$$\mathcal{H} \vDash \mathcal{S} : \Gamma \tag{4.3}$$

then

$$\mathcal{H}' \vDash v{:}C \tag{4.a}$$
$$\mathcal{H}' \vDash \mathcal{S}{:}\Gamma \tag{4.b}$$

hold as well.

Proof. Note that claim (4.b) follows directly by Lemma 4.8 and Lemma 4.12. Each location $\ell \in \mathrm{dom}(\mathcal{H})$ is either left unaltered or is overwritten by the value **Bad** and hence does not invalidate memory consistency.

However, the first claim (4.a) requires a proof by induction on the lengths of the derivations of (4.2) and (4.1) ordered lexicographically with the derivation of the evaluation taking priority over the typing derivation. This is required since an induction on the length of the typing derivation alone would fail for the case of function application: in order to allow recursive functions, the type rule for application is a terminal rule relying on the type given for the function in the program's signature. However, proving this case requires induction on the statement that the body of the function is well-typed, which is most certainly a type derivation of a longer length (i.e. longer than one step), prohibiting us from using the induction hypothesis. Note in this particular case that the length of the derivation for the evaluation statement does decrease. An induction over the length of the derivation for premise (4.2) alone fails similarly. Consider the last step in the derivation of premise (4.1) being derived by the application of a structural rule, then the length of the derivation for (4.2) remains exactly the same, while the length of the derivation for premise (4.1) does decrease by one step.

Using induction on the lexicographically ordered lengths of the type and evaluation derivations allows us to use the induction hypothesis if either the length of the derivation for premise (4.2) is shortened or if the length of the derivation for premise (4.2) remains unchanged while the length of the typing derivation is reduced. We first treat the cases where the last step in the typing derivation was obtained by application of a structural rule, which are all the cases which leave the length of the derivation for the evaluation unchanged. We then continue to consider the remaining cases based

on the evaluation rule that had been applied last to derive premise (4.2), since the remaining type rules are all syntax directed and thus unambiguously determined by the applied evaluation rule.

LF⊢Weak**:** Let $\Gamma = \Delta, x{:}A$, with x being the variable that is dropped and let $\ell = \mathcal{S}(x)$. By Lemma 4.10 we may directly apply the induction hypothesis using Δ alone to obtain (4.a) straight away.

LF⊢Share**:** Assume that the last step in the derivation for (4.1) was the application of rule LF⊢Share, yielding the expression $e[z/x, z/y]$. Assume further that $z \in \mathrm{FV}(e)$, since otherwise there would be no effect of the application of rule LF⊢Share. Let $\mathcal{S}_{xy} = (\mathcal{S}\backslash z)[x \mapsto \mathcal{S}(z), y \mapsto \mathcal{S}(z)]$. It is obvious that both $\mathcal{S}_{xy}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ and $\mathcal{H} \vDash \mathcal{S}_{xy} : \Gamma$ hold, since both new variables refer to the same values as the old variable before.

We apply the induction hypothesis to derive (4.a) and $\mathcal{H}' \vDash \mathcal{S}_{xy} : \Gamma_0, x : A_1, y : A_2$. By $A = A_1$ and $A = A_2$ from the definition of the sharing relation, we conclude that $\mathcal{H}' \vDash \mathcal{S}_{xy} : \Gamma_0, z : A$ as required.

LF↓Unit**,** LF↓True**,** LF↓False**:** For (4.a) we simply verify that indeed $\mathcal{H} \vDash$ `():unit` by Definition 4.9. We remark that in this case even $\mathcal{H} = \mathcal{H}'$ holds, so the claim for (4.b) also follows trivially.

The cases for the boolean values follow analogously.

LF↓Var**:** Claim (4.a), $\mathcal{H}' \vDash \mathcal{S}(x) : \Gamma(x)$, follows directly from assumption (4.3) since we have $v = \mathcal{S}(x)$ and again $\mathcal{H} = \mathcal{H}'$.

LF↓Let**:** Let $\Gamma = \Gamma_1, \Gamma_2$, hence $\mathcal{H} \vDash \mathcal{S}{:}\Gamma_1$ and $\mathcal{H} \vDash \mathcal{S}{:}\Gamma_2$ by Lemma 4.10. We thus apply the induction hypothesis for $\mathcal{S}, \mathcal{H} \vdash e_1 \downarrow v_0, \mathcal{H}_0$ and obtain $\mathcal{H}_0 \vDash v_0{:}A$.

From the application of Lemma 4.8 and Lemma 4.12 we obtain $\mathcal{H}_0 \vDash \mathcal{S}{:}\Gamma_2$ by Lemma 4.11, thus allowing us to apply the induction hypothesis again for $\mathcal{S}[x{\mapsto}v_0], \mathcal{H}_0 \vdash e_2 \downarrow v, \mathcal{H}'$ to obtain the desired result.

LF↓Fun**:** First note that $\mathcal{H} \vDash [y_1{\mapsto}v_1, \ldots, y_k{\mapsto}v_k]{:}\{y_1{:}A_1, \ldots, y_k{:}A_k\}$ follows from $\mathcal{H} \vDash [x_1{\mapsto}v_1, \ldots, x_k{\mapsto}v_k]{:}\{x_1{:}A_1, \ldots, x_k{:}A_k\}$ since both statements are equal by Definition 4.9, which eliminates the name right away.

By the assumption that program $\mathcal{P}$ is well-typed, see Definition 4.5, we obtain $\{y_1{:}A_1, \ldots, y_k{:}A_k\} \vdash e_f : C$. Hence we may apply the induction hypothesis for $[y_1{\mapsto}v_1, \ldots, y_k{\mapsto}v_k], \mathcal{H} \vdash e_f \downarrow v, \mathcal{H}'$ leading directly to the required result.

LF ↓If-T, LF ↓If-F: Both case follow trivially by application of the induction hypothesis for the corresponding subterm, i.e. e_t in the case of $\mathcal{S}(x) = \mathbf{tt}$, and e_f the case of $\mathcal{S}(x) = \mathbf{ff}$.

LF ↓Pair: By inspection of Definition 4.9 $\mathcal{H} \vDash \big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big){:}\Gamma(x_1){\times}\Gamma(x_2)$ follows from both $\mathcal{H} \vDash \mathcal{S}(x_1){:}\Gamma(x_1)$ and $\mathcal{H} \vDash \mathcal{S}(x_2){:}\Gamma(x_2)$. This suffices, since $\mathcal{H} = \mathcal{H}'$.

LF ↓E-Pair: We have $\Gamma(x) = A{\times}B$, $\mathcal{S}(x) = (v_1, v2)$ and $\mathcal{H} \vDash (v_1, v_2){:}A{\times}B$. By Definition 4.9 this can only be derived if both $\mathcal{H} \vDash v_1{:}A$ and $\mathcal{H} \vDash v_2{:}B$ hold, the premises of the only applicable rule. Using Lemma 4.11 to merge both statements we obtain $\mathcal{H} \vDash \mathcal{S}[x_1 \mapsto v_1, x_2 \mapsto v_2]{:}\{\Gamma, x_1{:}A, x_2{:}B\}$. Therefore we may apply the induction hypothesis to conclude as required.

LF ↓Inl,LF ↓Inr: We only consider the case for $e = \texttt{Inl}(x)$, since the other case then follows by symmetry.

We have $v = \ell$, $\ell \notin \mathrm{dom}(\mathcal{H})$ (and hence $\mathcal{H} \setminus \ell = \mathcal{H}$) and $\mathcal{H}' = \mathcal{H}[\ell \mapsto (\mathbf{tt}, \mathcal{S}(x))]$. The required statement $\mathcal{H}[\ell \mapsto (\mathbf{tt}, \mathcal{S}(x))] \vDash \ell{:}A{+}B$ follows by Definition 4.9 from premise (4.1), i.e. $\mathcal{H} \vDash x{:}A$, since $\Gamma = \{x{:}A\}$.

LF ↓E-Inl, LF ↓E!-Inl, LF ↓E-Inr, LF ↓E!-Inr: We only consider the right case, i.e. $\mathcal{H}\big(\mathcal{S}(x)\big) = (\mathbf{ff}, w')$, with the other case following by symmetry.

Let $\Gamma = \{\Gamma_0, x{:}A{+}B\}$. We have $\{\Gamma_0, z{:}B\} \vdash e_r{:}C$, $\mathcal{H} \vDash \mathcal{S}{:}\{\Gamma_0, x{:}A{+}B\}$, $\mathcal{S}(x) = \ell$ and $\mathcal{H}(\ell) = (\mathbf{ff}, w')$. Therefore necessarily $\mathcal{H} \setminus \ell \vDash w'{:}B$ by the premises of the only applicable rule of Definition 4.9 that allowed the derivation of $\mathcal{H} \vDash \ell{:}A{+}B$. This allows us to deduce $\mathcal{H} \vDash \mathcal{S}[z{\mapsto}w']{:}\{\Gamma_0, z{:}B\}$ by the closure properties of memory consistency, i.e. Lemma 4.10, and Lemma 4.11. Furthermore $\mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vDash \mathcal{S}[z{\mapsto}w']{:}\{\Gamma_0, z{:}B\}$ by using Lemma 4.12. The former allows the application of the induction hypothesis in the case of the read-only match (LF ↓E-Inr), while the latter allows the use of the induction hypothesis for the destructive match (LF ↓E!-Inr), depending on which rule had been used last to derive premise (4.2). Both cases then yield $\mathcal{H}' \vDash v{:}C.$ directly.

LF ↓Nil: The case is trivial, since by Definition 4.9 we have $\varnothing \vDash \mathbf{Null}{:}\mathsf{list}(A)$.

LF ↓Cons: We have $\mathcal{H}' = \mathcal{H}[\ell \mapsto \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big)]$ for some $\ell \notin \mathrm{dom}(\mathcal{H})$. Furthermore our assumptions yield $\mathcal{H} \vDash \mathcal{S}(x_h){:}A$ and $\mathcal{H} \vDash \mathcal{S}(x_t){:}\mathsf{list}(A)$, which allows us to deduce $\mathcal{H} \vDash \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big){:}A{\times}\mathsf{list}(A)$ by the rule for pairs of Definition 4.9. Continuing with the rule for non-empty list of the same definition, we obtain $\mathcal{H}[\ell \mapsto \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big)] \vDash \ell{:}\mathsf{list}(A)$ as required.

LF ↓E-Nil, LF ↓E!-Nil: Follows trivially by induction, since both heap $\mathcal{H}$ and stack $\mathcal{S}$ as well as value v and post-heap $\mathcal{H}'$ remain unchanged in the premise of either evaluation rule.

LF ↓E-Cons, LF ↓E!-Cons: We have $\mathcal{H} \vDash \ell{:}\mathsf{list}(A)$ and $\mathcal{H}(\ell) = (w_h, x_t)$.Therefore necessarily $\mathcal{H} \setminus \ell \vDash \mathcal{H}(\ell){:}A{\times}\mathsf{list}(A)$ by the only possible rule of Definition 4.9 to derive the previous statement. This in turn could only be deduced if both $\mathcal{H} \setminus \ell \vDash w_h{:}A$ and $\mathcal{H} \setminus \ell \vDash w_t{:}\mathsf{list}(A)$ holds. Using Lemma 4.11 to join both statements into the stack, and Lemma 4.10 to add $[\ell \mapsto (w_h, w_t)]$ back to the heap in the read-only case and $[\ell \mapsto \mathbf{Bad}]$ in the destructive case, we can then apply the induction hypothesis to obtain the required result.

□

4.4 Introductory Program Examples

We now discuss some simple program examples in order to illustrate the use of LF:

`tails` computes all tails of a given list, thereby producing aliased data;

`reverse` demonstrates in-place list reversal through destructive matches;

`zip` merges two lists to a list of pairs, exhibiting ambiguous cost bounds; and

`ins_sort` in-place list sorting by insertion, included for traditional purposes.

These LF program examples are simple enough to be easily understood, yet demonstrate several interesting features, such as aliasing, in-place update. We will apply our analysis technique to each example at the end of this chapter, in Section 5.5, and see what we may learn about these programs through the analysis.

For all program examples discussed in this thesis, we adopt a series of simplifying conventions:

- Most program examples may admit several LF type signatures, e.g. the function `reverse` can reverse lists of any type, hence the function admits the type $\mathsf{list}(\mathsf{bool}) \to \mathsf{list}(\mathsf{bool})$ just as well as type $\mathsf{list}(\mathsf{int}) \to \mathsf{list}(\mathsf{int})$. In such cases we allow ourselves to simply write $\mathsf{list}(A) \to \mathsf{list}(A)$ instead, although the shown type system of LF is not polymorphic, unlike the one shown in [JLHH10]. The letter A should here be interpreted as a meta-variable, meaning that it is to be instantiated to any arbitrary LF type first. We believe that this actually helps the reader to understand the crucial parts of the example and avoids distraction.

- For the same reason we may also use meta-variables for values, e.g. we believe that an example for explaining function `reverse` is easier to understand in the form $\texttt{reverse}([a, b, c, d]) = [d, c, b, a]$. rather than the concrete version $\texttt{reverse}([\mathbf{tt}, \mathbf{ff}, \mathbf{tt}, \mathbf{tt}]) = [\mathbf{tt}, \mathbf{tt}, \mathbf{ff}, \mathbf{tt}]$.

- Similarly we might also make use of integers within examples, although integer numbers are not a part of LF. Standard operations on the integers, like $+, -, *, \ldots, =, >, \geq, \ldots$, might also occur and are to be interpreted as normal function calls, despite the infix notation. We could have added the integers to LF, which would have required us to simply expand many definitions in the standard way. However, it is clear that the integers could be encoded within LF as it is, e.g. by using lists of unit elements. So our system would only become bigger and more unwieldy, but not more powerful.

- In some of the larger program examples we may also ignore the requirement for let-normal form. For example, we may simply write $\texttt{if}\ a > b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ instead of the required $\texttt{let}\ x = a > b\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, which is in general more difficult to read for a human. We will only perform this simplification when the let-normal form is straightforward to derive.

4.4.1 List Tails

Our first example is the function `tails`, which computes a list containing all the tail-lists of the given input list, shown in Figure 4.4. For example the call $\texttt{tails}([a, b, c, d])$

$$
\begin{array}{lcl}
\Sigma(\texttt{tails}) & := & \mathsf{list}(A) \to \mathsf{list}(\mathsf{list}(A)) \\
\mathcal{P}(\texttt{tails}) & := & l \quad \to \quad \texttt{let}\ r = \\
& & \qquad\qquad\quad \texttt{match}\, l\, \texttt{with} \\
& & \qquad\qquad\quad \texttt{|Nil} \to \texttt{Nil} \\
& & \qquad\qquad\quad \texttt{|Cons}(h,t) \to \texttt{tails}(t) \\
& & \qquad\qquad \texttt{in Cons}(l,r)
\end{array}
$$

Figure 4.4: List tails

evaluates to the list $\big[[a,b,c,d],[b,c,d],[c,d],[d],[\,]\big]$, where a, b, c and d are some arbitrary values.

The given code for `tails`, which is in let-normal form as required, first evaluates a subexpression, whose result is then being bound to a local variable r. This subexpression inspects the input list l. If the input is the empty list, then the empty list is also returned for the subexpression; otherwise the function `tails` is recursively called on the tail of the input list. Eventually, a list node is always created, containing the unmodified input list as its head and the value that was bound to r as its tail.

Suppose we have the heap

$$\mathcal{H} = \big[1 \mapsto (a,2), 2 \mapsto (b,3), 3 \mapsto (c,4), 4 \mapsto (d, \mathbf{Null})\big]$$

then using our operational semantics, we can derive

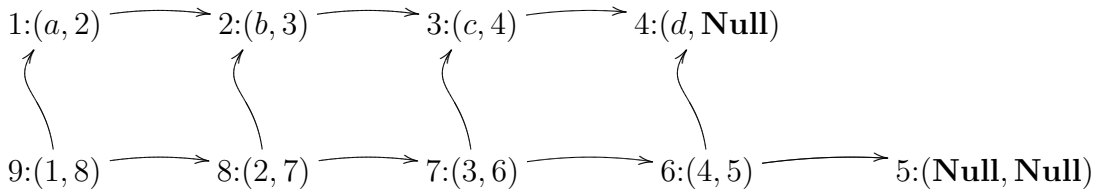
$$
\begin{array}{l}
[l \mapsto 1], \mathcal{H} \vdash \texttt{let}\ r = \texttt{match}\, l\, \texttt{with} \\
\qquad\qquad\qquad\qquad\quad \texttt{|Nil} \to \texttt{Nil} \\
\qquad\qquad\qquad\qquad\quad \texttt{|Cons}(h,t) \to \texttt{tails}(t) \\
\qquad\qquad\quad \texttt{in Cons}(l,r) \downarrow 9, \mathcal{H}'
\end{array}
$$

where

$$\mathcal{H}' = \mathcal{H}\big[5 \mapsto (\mathbf{Null}, \mathbf{Null}), 6 \mapsto (4,5), 7 \mapsto (3,6), 8 \mapsto (2,7), 9 \mapsto (1,8)\big]$$

Note that there is some choice for the locations of the newly allocated list nodes, as the rule for cons allows any currently unused location to be used. For this example we chose the lowest number available at the corresponding point of computation.

However, the more interesting feature of `tails` is that it uses its argument list l twice: once in the match that precedes the recursive call; and another time in the construction of the result. This forces the match to be read-only, since otherwise the result of the computation would be of little use. A second effect is that it also causes the result to contain aliased data, since nodes can be reached through several differing paths. We can see in $\mathcal{H}'$ that the heap value $(b, 3)$ can be accessed from location 9 by two differing paths, namely via $9 \rightsquigarrow 1 \rightsquigarrow 2$ or $9 \rightsquigarrow 8 \rightsquigarrow 2$. Drawing the heap $\mathcal{H}'$ as a diagram makes the aliases easy to see:



Since we are interested in the heap consumption of programs, we will have a brief look at the heap consumption of function `tails` now, before introducing our automatic analysis in the following chapter. We can see that the example call with a list of length four created 5 new list nodes, but is there a general formula to compute the heap space usage of `tails`? Well, one may observe that regardless of the outcome of the case, a call to `tails` will always create one list node. If the input list was empty, this is the only node created. If the input list was non-empty, then `tails` recursively calls itself again through a tail-call. Hence it is easy to see that for an input list of length n, `tails` will allocate $n+1$ new list nodes only. Thanks to aliasing, this is much less than one might expect from the large output of $\big[[a, b, c, d], [b, c, d], [c, d], [d], [\,]\big]$. We will later see that the automatic analysis is able to derive an exact cost bound for this program example.

4.4.2 List Reversal

Our next example is the in-place reversal of a list, shown in Figure 4.5. This function should not allocate any new memory at all, but should deallocate a part of the input before the new space for the output is allocated.

$$
\begin{array}{lcll}
\Sigma(\texttt{reverse}) & := & \mathsf{list}(A) \to \mathsf{list}(A) & \\
\mathcal{P}(\texttt{reverse}) & := & l \quad \to & \texttt{let } n \texttt{ = Nil in rev\_acc}(n, l) \\
\\
\mathcal{P}(\texttt{rev\_acc}) & := & r, l \quad \to & \texttt{match!}\, l \texttt{ with} \\
 & & & \quad \texttt{|Nil} \to r \\
 & & & \quad \texttt{|Cons}(h\texttt{,}t) \to \texttt{let } s \texttt{ = Cons}(h\texttt{,}r) \\
 & & & \qquad\qquad\qquad \texttt{in rev\_acc}(s, t)
\end{array}
$$

Figure 4.5: In-place list reversal

Function `reverse` is defined by a recursive helper function `rev_acc`, that uses an accumulator to gather the output step-by-step:

$$
\begin{aligned}
\texttt{reverse}([a,b,c,d]) &= \texttt{rev\_acc}([\,], [a,b,c,d]) \\
&= \texttt{rev\_acc}([a], [b,c,d]) \\
&= \texttt{rev\_acc}([b,a], [c,d]) \\
&= \texttt{rev\_acc}([c,b,a], [d]) \\
&= \texttt{rev\_acc}([d,c,b,a], [\,]) \qquad\qquad = [d,c,b,a]
\end{aligned}
$$

We see that `rev_acc` has two computational paths, depending on whether its input is the empty list or not. If the input is indeed empty, then nothing is deallocated nor allocated, only the accumulator is returned as it is. If, on the other hand, the input list was non-empty, then the first node is deallocated and the contained element is placed in a newly constructed list node on the top of the accumulator, before recursively calling itself on the tail of the matches list. So no temporary heap space is needed, since each allocation is preceded by a deallocation of a node of the same type.

Note however, that our storeless semantics will produce a result like

$$[x \mapsto 1], \mathcal{H} \vdash \texttt{reverse}(x) \downarrow 8, \mathcal{H}'$$

where

$$\mathcal{H} = \left[\ 1 \mapsto (a, 2), 2 \mapsto (b, 3), 3 \mapsto (c, 4), 4 \mapsto (d, \mathbf{Null})\ \right]$$
$$\mathcal{H}' = \begin{bmatrix} 1 \mapsto \mathbf{Bad},\ 2 \mapsto \mathbf{Bad},\ 3 \mapsto \mathbf{Bad},\ 4 \mapsto \mathbf{Bad}, \\ 5 \mapsto (a, \mathbf{Null}), 6 \mapsto (b, 5), 7 \mapsto (c, 6), 8 \mapsto (d, 7) \end{bmatrix}$$

We see that the former locations are still within the domain of the heap. However, these should not be interpreted to take up memory space, since the locations should be regarded as symbolic handles for real memory address only, as explained in Section 4.3.1. So the locations are just labels for memory addresses, and one memory address might have several labels attached to it, with only the last one being live, and all the previous ones being stale pointers.

4.4.3 List Zipping

The program code for `zip`, shown in let-normal form in Figure 4.6 on the facing page, merges two separate lists into a list of pairs, conserving the order of the input lists. Here are some example computations of `zip`:

$$\begin{array}{llllll} \texttt{zip(} & [a,b,c,d], & [e,f,g,h] & \texttt{)} & = & [(a,e),(b,f),(c,g),(d,h)] \\ \texttt{zip(} & [a,b,c,d], & [e,f] & \texttt{)} & = & [(a,e),(b,f)] \\ \texttt{zip(} & [a], & [e,f] & \texttt{)} & = & [(a,e)] \\ \texttt{zip(} & [\,], & [e,f] & \texttt{)} & = & [\,] \end{array}$$

We can see that the length of the output is equal to the minimum of the lengths of the two input lists. Both matches are read-only and no new aliasing is introduced. Hence the memory usage of function `zip` is also proportional to the minimum of the lengths of the input lists.

4.4.4 In-Place Insertion Sort

Our last simple example for Chapter 4 is the well-known insertion sort algorithm, shown in Figure 4.7 on the next page. The program code assumes the existence of a suitable comparison function $\Sigma(\texttt{leq}) = (A, A) \to \texttt{bool}$.

```
Σ(zip)  :=  (list(A), list(B)) → list(A×B)
P(zip)  :=  l, r  →  match l with
                      |Nil → Nil
                      |Cons(x,l') →
                          match r with
                           |Nil → Nil
                           |Cons(y,r') →
                              let p = (x, y) in
                              let z = zip(l', r') in Cons(p,z)
```

Figure 4.6: List zipping

```
Σ(ins_sort)  :=  list(A) → list(A)
P(ins_sort)  :=  l  →  match! l with
                        |Nil → Nil
                        |Cons(h,t) →
                            let s = ins_sort(t) in insert(h, s)
  Σ(insert)  :=  A, list(A) → list(A)
  P(insert)  :=  x, l  →  match! l with
                          |Nil → let n = Nil in Cons(x,n)
                          |Cons(h,t) →
                              if leq(x, h)
                                 then let t' = Cons(h,t) in Cons(x,t')
                                 else let s = insert(x,t) in Cons(h,a)
```

Figure 4.7: In-place insertion sort

A list of elements is sorted by successively inserting each element of the input list into the empty list:

$$\begin{aligned}
\texttt{sort}\big([c,d,a,b]\big) &= \texttt{insert}\big(c, \texttt{sort}([d,a,b])\big) \\
&= \texttt{insert}\big(c, \texttt{insert}\big(d, \texttt{sort}([a,b])\big)\big) \\
&= \texttt{insert}\big(c, \texttt{insert}\big(d, \texttt{insert}\big(a, \texttt{sort}([b])\big)\big)\big) \\
&= \texttt{insert}\big(c, \texttt{insert}\big(d, \texttt{insert}\big(a, \texttt{insert}\big(b, \texttt{sort}([\,])\big)\big)\big)\big) \\
&= \texttt{insert}\big(c, \texttt{insert}\big(d, \texttt{insert}\big(a, \texttt{insert}(b, [\,])\big)\big)\big) \\
&= \texttt{insert}\big(c, \texttt{insert}\big(d, \texttt{insert}(a,\ [b])\big)\big) \\
&= \texttt{insert}\big(c, \texttt{insert}(d,\ [a,b])\big) \\
&= \texttt{insert}(c,\ [a,b,d]) \\
&= [a,b,c,d]
\end{aligned}$$

Function `sort` sorts its input list *in-place*, i.e. without allocating new heap memory. This is again achieved through step-wise deallocation of the input list and all intermediate lists that are created during the evaluation. Note that no temporary heap space is required either.

An interesting property of this example program is that `sort` deallocates a list node, but never allocates one. On the other hand, `insert` has a computational branch which allocates two list nodes, but only deallocates at most one node. We will soon see how the analysis handles this situation correctly, delivering an exact heap usage bound.

4.5 Cost-Aware Semantics

We conclude this chapter by augmenting our operational semantics with a counter for the size of the remaining heap memory. This cost-aware operational semantics will be the *measure* for the heap space consumption of a program, that we wish to predict by the method shown in the next chapter. Similar to the type systems for LF and $\text{LF}_\diamond$, we distinguish the rules for cost-aware operational semantics from the plain operational semantics in Section 4.3 by adding the symbol $\diamond$ to the name of the rules.

We must first fix a concrete memory model, which we do as general as possible by defining a function $\textsc{Size} : \mathsf{Val} \to \mathbb{N}$. The values chosen for this function should

fit the intended machine model, but are completely arbitrary otherwise. We will always refer to Size instead of using actual numeric values, so all proofs are in fact parametric in Size. For the sake of a simple discussion of our program examples, we *choose* a very simple definition of Size as well.

Definition 4.14 (Sizes of Operational Values)**.** We define $\textsc{Size} : \mathsf{Val} \to \mathbb{N}$ by

$$\begin{aligned}
\textsc{Size}(()) &= 1 \\
\textsc{Size}(\mathbf{tt}) &= 1 \\
\textsc{Size}(\mathbf{ff}) &= 1 \\
\textsc{Size}\big((v_1, v_2)\big) &= \textsc{Size}(v_1) + \textsc{Size}(v_2) \\
\textsc{Size}(\ell) &= 1 \\
\textsc{Size}(\mathbf{Null}) &= 1 \\
\textsc{Size}(\mathbf{Bad}) &= 0
\end{aligned}$$

The idea is that value v occupies $\textsc{Size}(v)$ memory units when stored in the heap. Since sums and lists are both *boxed* in our chosen model, i.e. they are always stored on the heap, a value representing either is a mere pointer ℓ and hence is of course assigned the size of a pointer only. Pairs of values are *unboxed*, which implies that the sizes of nodes generally depend on their contents, since any node storing a pair of values has to store both values, and not just a fixed-size pointer to a pair. Note that the amortised analysis technique can be applied to various memory models, both boxed and unboxed, as discussed and demonstrated in Chapters 7 and 8 respectively.

The size of a data node is determined by its layout as defined in the rule of the operational semantics dealing with the specific constructor of that node, e.g. in rule LF$\downarrow$Cons we see that a cons node of a List is a pair consisting of another value and a pointer. If the value stored in the list is a pair of booleans, then the overall size of that particular list node would be three (one for each boolean and one for the pointer to the tail of the list according to the definition shown above). This can be computed with the definition shown above.

In order to obtain the overall memory occupied by an entity, e.g. an entire list given by a pointer ℓ, one must sum the sizes of all the nodes which form that list. Since we only need to account for the size of the remaining heap space, we do not need a definition to compute the size of a whole actual list stored somewhere in the heap. It is sufficient for us to be able to determine the size of each node of a compound data structure individually.

Our type-based analysis must also know about our choices on the memory model. However, operational values are not accessible within type rules. We therefore define an analogous size function TypSize, which maps LF types to natural numbers, and then prove that the two notions of size agree for consistent memory configurations.

Note that there is an important alternative to this approach, that has been employed in our works [JLH+09, JLHH10]. If the memory model uses tagged records to store a value within the heap, then a single size function can be solely based on these constructor tags. This then allows us to assign different sizes to constructors of one and the same type, and further size related optimisations, such as variably sized constructors, without loosing too much precision. Since both the operational semantics and the types rules can make use of the constructor-tag based size function, we would neither need the following Definition 4.15 nor Lemma 4.16.

Definition 4.15 (Sizes of LF Types)**.** We define $\textsc{TypSize} : \text{LF-type} \to \mathbb{N}$ by

$$\begin{aligned}
\textsc{TypSize}(\mathsf{unit}) &= 1 \\
\textsc{TypSize}(\mathsf{bool}) &= 1 \\
\textsc{TypSize}(A \times B) &= \textsc{TypSize}(A) + \textsc{TypSize}(B) \\
\textsc{TypSize}(A + B) &= 1 \\
\textsc{TypSize}(\mathsf{list}(A)) &= 1
\end{aligned}$$

Note that the assigned size for the type of list and sums represents the size of a pointer to data of that type only, due to the boxing of values for these types as explained further above.

The actual number of heap units occupied by data node depends on its contents, there the function TypSize will appear in sums over the contents of a constructor in the annotated type rules of the next chapter. So the overall size of a cons node of a list of type $\mathsf{list}(A)$ will be the sum of the size of the stored value and the size of the pointer to the tail, i.e. $\textsc{TypSize}(A) + \textsc{TypSize}(\mathsf{list}(A))$. For a cons node of $\mathsf{list}(\mathsf{bool} \times \mathsf{bool})$ holding a pair of booleans, we have $\textsc{TypSize}(\mathsf{bool} \times \mathsf{bool}) + \textsc{TypSize}(\mathsf{list}(\mathsf{bool} \times \mathsf{bool})) = \textsc{TypSize}(\mathsf{bool}) + \textsc{TypSize}(\mathsf{bool}) + 1 = 3$, the same value that we obtained in our discussion of the operational size Size.

We now state formally that our choices for Size and TypSize, as given in Definitions 4.14 and 4.15 above, agree indeed.

Lemma 4.16 (Size Correspondence)**.** *If* $\mathcal{H} \vDash v : A$ *then* $\textsc{Size}(v) = \textsc{TypSize}(A)$.

Proof. By induction on the length of the derivation of $\mathcal{H} \vDash v : A$. We proceed by case distinction on the last rule applied to establish memory consistency.

$\mathcal{H} \vDash ():\mathsf{unit}$ **:** We have $\textsc{Size}(()) = 1 = \textsc{TypSize}(\mathsf{unit})$.

$\mathcal{H} \vDash \mathbf{tt}:\mathsf{bool}$**,** $\mathcal{H} \vDash \mathbf{tt}:\mathsf{bool}$ **:** It is clear that $\textsc{Size}(\mathbf{tt}) = \textsc{Size}(\mathbf{ff}) = 1 = \textsc{TypSize}(\mathsf{bool})$.

$\mathcal{H} \vDash (v, w):A \times B$ **:** Observe that $\textsc{Size}\big((v, w)\big) = \textsc{Size}(v) + \textsc{Size}(w) = \textsc{TypSize}(A) + \textsc{TypSize}(B) = \textsc{TypSize}\big(A \times B\big)$ holds, where the middle equality is justified by applying the induction hypothesis twice. This is the only case that requires the use of the induction hypothesis.

$\mathcal{H} \vDash \ell:A+B$ **:** This statement can be obtained from three different memory consistency rules. However, we have $\textsc{Size}(\ell) = 1 = \textsc{TypSize}(A+B)$ by definition, regardless of whether $\mathcal{H}(\ell)$ is equal to either $(\mathbf{tt}, v), (\mathbf{ff}, w)$ or $\mathbf{Bad}$.

$\mathcal{H} \vDash \mathbf{Null}:\mathsf{list}(A)$ **:** By definition $\textsc{Size}(\mathbf{Null}) = 1 = \textsc{TypSize}\big(\mathsf{list}(A)\big)$.

$\mathcal{H} \vDash \ell:\mathsf{list}(A)$ **:** This statement can be obtained from two different memory consistency rules only. We clearly have $\textsc{Size}(\ell) = 1 = \textsc{TypSize}\big(\mathsf{list}(A)\big)$ by definition, regardless of whether $\mathcal{H}(\ell) = (v, w)$ or $\mathcal{H}(\ell) = \mathbf{Bad}$ holds.

□

Please note that strict equality is in fact required if we want to safely account for deallocation of data objects as well as their allocation. Otherwise, if equality could not be achieved for a given memory model, one would need two different type-based size functions for allocation and deallocation, which yield the possible maximal size and the possible minimal size for safely estimating the worst-case cost of allocation and deallocation respectively, since, e.g., for deallocation we can only expect the minimal number of heap cells to be freed.

The loss of the strict equality between Size and TypSize would thus render our analysis less precise by these necessary over-approximations. For example, an unboxed sum type would suffer from this imprecision, if its components had different sizes. However, in practise this is usually avoided, since unboxed data nodes are then often padded to gain other advantages, such as gaining a speed up in memory access by simplified calculations on uniformly spaced addresses.

4.5.1 Measuring Space Usage

The judgement $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \downarrow v, \mathcal{H}'$ means that under the initial stack $\mathcal{S}$ and heap $\mathcal{H}$, the expression e evaluates to value v, and post-heap $\mathcal{H}'$, provided that there are at least $m \in \mathbb{N}$ heap units available before the computation. Furthermore, $m' \in \mathbb{N}$ unused heap units will be available after the computation. In contrast, the statement $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ from the previous unannotated Definition 4.7 just states that e evaluates to v using an unknown, but finite, amount of heap units.

For example, $\mathcal{S}, \mathcal{H} \overset{3}{\underset{1}{\vdash}} e \downarrow v, \mathcal{H}'$ means that three heap cell units are sufficient to allow e to be evaluated, and that exactly one heap unit is unused after the computation. This unused cell might or might not have been used temporarily, so we do not know whether $\mathcal{S}, \mathcal{H} \overset{2}{\underset{0}{\vdash}} e \downarrow v, \mathcal{H}'$ is also derivable or not. Of course, it is the goal of our analysis to determine the minimal number m and m' such that an evaluation is possible. Note that the two values can track both the overall net heap space costs (i.e. total heap allocations minus total deallocations) as well as the minimum number of free heap memory that are sufficient for the computation to succeed (i.e. maximum heap residency). Therefore a temporary high usage does not escape our measurement through the annotated operational semantics.

Definition 4.17 (LF Operational Semantics with Freelist Counter)**.**

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m}{\vdash}} * \downarrow (), \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\textsc{Unit})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m}{\vdash}} \texttt{true} \downarrow \mathbf{tt}, \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\textsc{True})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m}{\vdash}} \texttt{false} \downarrow \mathbf{ff}, \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\textsc{False})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m}{\vdash}} x \downarrow \mathcal{S}(x), \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\textsc{Var})$$

$$\frac{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m_0}{\vdash}} e_1 \downarrow v_0, \mathcal{H}_0 \qquad \mathcal{S}[x \mapsto v_0], \mathcal{H}_0 \overset{m_0}{\underset{m'}{\vdash}} e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\textsc{Let})$$

$$\frac{\begin{array}{c}\mathcal{S}(x_i) = v_i \quad (\text{for } i = 1, \ldots, k) \\ \mathcal{P}(f) = (y_1, \ldots, y_k \to e_f) \qquad [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k], \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e_f \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} f(x_1, \ldots, x_k) \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\textsc{Fun})$$

$$\frac{\mathcal{S}(x) = \mathbf{tt} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} e_t \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{If-T})$$

$$\frac{\mathcal{S}(x) = \mathbf{ff} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} e_f \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{If-F})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m}} (x_1, x_2) \downarrow \big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big), \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\text{Pair})$$

$$\frac{\mathcal{S}(x) = (v_1, v_2) \qquad \mathcal{S}[x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \overset{m}{\vdash_{m'}} e \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{match}\, x\, \texttt{with}\ (x_1, x_2) \to e \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E-Pair})$$

$$\frac{w = \big(\mathbf{tt}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \overset{m' + \textsc{Size}(w)}{\vdash_{m'}} \texttt{Inl}(x) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\text{LF}\downarrow^{\diamond}\text{Inl})$$

$$\frac{w = \big(\mathbf{ff}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \overset{m' + \textsc{Size}(w)}{\vdash_{m'}} \texttt{Inr}(x) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\text{LF}\downarrow^{\diamond}\text{Inr})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H} \overset{m}{\vdash_{m'}} e_l \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \texttt{|Inr}(z) \to e_r \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E-Inl})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(w)}{\vdash_{m'}} e_l \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{match!}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \texttt{|Inr}(z) \to e_r \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E!-Inl})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[z \mapsto w'], \mathcal{H} \overset{m}{\vdash_{m'}} e_r \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \texttt{|Inr}(z) \to e_r \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E-Inr})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[z \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(w)}{\vdash_{m'}} e_r \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m'}} \texttt{match!}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \texttt{|Inr}(z) \to e_r \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E!-Inr})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash_{m}} \texttt{Nil} \downarrow \mathbf{Null}, \mathcal{H}} \quad (\text{LF}\downarrow^{\diamond}\text{Nil})$$

$$\frac{w = \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} \texttt{Cons}(x_h\texttt{,}x_t) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\text{LF}\downarrow^{\diamond}\textsc{Cons})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E-}\textsc{Nil})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E!-}\textsc{Nil})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \\ \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H} \vdash^{m}_{m'} e_2 \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E-}\textsc{Cons})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \\ \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash^{m + \textsc{Size}(w)}_{m'} e_2 \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow v, \mathcal{H}'} \quad (\text{LF}\downarrow^{\diamond}\text{E!-}\textsc{Cons})$$

Note that deallocation only occurs in the operational rules LF $\downarrow^{\diamond}$E-Inl, LF $\downarrow^{\diamond}$E-Inr and LF $\downarrow^{\diamond}$E-Cons, where the freed location ℓ is changed to **Bad** for the ongoing computation. The counter for the remaining available heap space is correspondingly increased by the size of the deallocated value for the subsequent steps of the computation.

We observe that additional memory is harmless.

Lemma 4.18 (Additional Memory). *Fix an* LF$_\diamond$ *program* $\mathcal{P}$ *with signature* Σ. *If* $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'$ *holds, then for any* $k \in \mathbb{N}$ *also* $\mathcal{S}, \mathcal{H} \vdash^{m+k}_{m'+k} e \downarrow v, \mathcal{H}'$ *as well.*

Proof. By induction of the length of the derivation of the evaluation. For each rule we easily verify that it admits an increase of m, and returns a corresponding increase for m', provided that its premises allow the same, which we can assume by the induction hypothesis. □

It is clear that any evaluation having an unbounded amount of heap memory available succeeds, if the same evaluation already succeeds with only a finite amount of memory being available.

Lemma 4.19 (Cost Counting Inviolacy)**.** *Fix an* $\mathrm{LF}_\diamond$ *program* $\mathcal{P}$ *with signature* Σ*. If* $\mathcal{S}, \mathcal{H} \vdash_{m'}^{m} e \downarrow v, \mathcal{H}'$ *then* $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ *as well.*

Proof. The rules for the cost-aware operational semantics always have strengthened preconditions as compared to the standard operational semantics without cost counting. The rules are in fact identical except for the annotations added above and below the turnstile symbol. □

The opposite statement is also intuitively clear: any finite evaluation can only use a finite amount of heap space.

Lemma 4.20 (Finite Evaluation Costs)**.** *Fix an* $\mathrm{LF}_\diamond$ *program* $\mathcal{P}$ *with signature* Σ*.*

If $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ *then there exists* $m, m' \in \mathbb{N}$ *such that* $\mathcal{S}, \mathcal{H} \vdash_{m'}^{m} e \downarrow v, \mathcal{H}'$ *holds as well.*

Proof. By induction on the length of the derivation for $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$. For each rule that performs allocation or deallocation, we readily verify that the only restriction on m is that it was chosen to be sufficiently large. The value m' can simply be computed from the rule applications. The cases for the other rules follow trivially from the induction hypothesis, except for rule LF↓Let, which follows by Lemma 4.18. □

5 First-Order Analysis $\mathbf{LF}_{\diamond}$

$$\begin{array}{ll} \text{zero-order types:} & A ::= \mathsf{unit} \mid \mathsf{bool} \mid A \times A \mid (A, q) + (A, r) \mid \mathsf{list}(A, q) \\ \text{first-order types:} & F ::= (A, \ldots, A) \xrightarrow{q \triangleright q'} A \end{array}$$

$$\text{for } q, q', r \in \mathbb{Q}^+$$

Figure 5.1: Types of the cost-aware simple language $\mathrm{LF}_\diamond$

We will now augment our basic type system for LF by annotations to capture the potential of data. We refer to this refinement of the type rules by the name $\mathrm{LF}_\diamond$, and also add a $\diamond$-symbol to the name of the type rules, in order to distinguish them from their unannotated counterparts in LF.

The intuition for $\mathrm{LF}_\diamond$ is that the type of an entity gives rise to a canonical formula the denotes the potential of all values for that type, as given by Definition 6.20 later. It is the task of the $\mathrm{LF}_\diamond$ type system to ensure that the specific parameters used to specialise the canonical potential are meaningful with respect to the actual evaluation costs as specified by the cost-aware operational semantics given in Definition 4.17. The formulation of the $\mathrm{LF}_\diamond$ type system and proving its soundness are the main contributions of this chapter.

5.1 Cost-Aware Type System $\mathrm{LF}_\diamond$

The types of $\mathrm{LF}_\diamond$ are shown in Figure 5.1. The sum and list types were augmented by annotations to reflect their possible potential as outlined in Section 2.3. Furthermore the function type has received two annotations which will signify the fixed potential required to execute the function (q) and the amount of potential available after the computation of the function (q'), both in addition to the potential carried in the types of the input and output.

We could have assigned potential related annotation to all types, but this would have only added ambiguity without increasing expressivity. This will only become apparent with Definition 5.7, which formally defines the connection between annotated

types and potential, so we will comment in more detail on this directly after that definition. For now, recall from the introduction that the potential associated with a certain value of a specific annotated type varies with the kind and number of nodes that represent that value, as well as the annotation.

In order to relate $\mathrm{LF}_\diamond$ with LF, we define the erasure of type annotations as expected.

Definition 5.1 ($\mathrm{LF}_\diamond$ Type Reduction)**.**

$$\begin{array}{lcl} |\mathsf{unit}| & = & \mathsf{unit} \\ |\mathsf{bool}| & = & \mathsf{bool} \\ |A{\times}B| & = & |A|{\times}|B| \\ |(A,q)+(B,r)| & = & |A|{+}|B| \\ |\mathsf{list}(A,q)\,| & = & \mathsf{list}(|A|) \\ \left|(A_1,\ldots,A_n)\xrightarrow{q \triangleright q'} B\right| & = & |A_1|,\ldots,|A_n| \to |B| \end{array}$$

We naturally extend this definition to type contexts and type signatures by function composition.

We extend Definition 4.15, the function TypSize that determines the size of a type, from unannotated LF types to annotated $\mathrm{LF}_\diamond$ types by simply erasing the annotations. So the size of an $\mathrm{LF}_\diamond$ type is equal to the size of its underlying LF type.

Definition 5.2 (Sizes of $\mathrm{LF}_\diamond$ Types)**.** $\textsc{TypSize} : \mathrm{LF}_\diamond\text{-type} \to \mathbb{N}$ is defined by $\textsc{TypSize}(A) := \textsc{TypSize}\big(|A|\big)$.

The two definitions above are so trivial that we usually ignore them entirely, allowing ourselves to use $\mathrm{LF}_\diamond$ types wherever an LF type would formally be required.

We will now formulate the $\mathrm{LF}_\diamond$ type rules, which allow us to derive a judgement of the form

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

which means that term e has the $\mathrm{LF}_\diamond$ type A within context Γ and signature Σ. The non-negative rationals p and p' tell us something about the resource usage of the evaluation of term e as detailed later by Theorem 1.

As an intuition, the theorem will give us an upper-bound on the amount of heap units required to safely evaluate the term e. This upper-bound is given by a sum of non-negative rationals, more precisely p plus the potential held by the context Γ. The exact number depends on the actual stack and heap in correspondence to Γ, of course, as specified by the definition of potential, explained soon in Section 5.2.

Definition 5.3 ($\mathrm{LF}_\diamond$ Type Rules)**.** All number annotations $p, p', q, \ldots$ within the following type rules are restricted to non-negative rational numbers. The signature Σ is again implicitly fixed.

$$\frac{}{\{\,\} \vdash^{0}_{0} * : \mathsf{unit}} \quad (\mathrm{LF}\vdash_\diamond\textsc{Unit})$$

$$\frac{c \in \{\mathtt{true}, \mathtt{false}\}}{\{\,\} \vdash^{0}_{0} c : \mathsf{bool}} \quad (\mathrm{LF}\vdash_\diamond\textsc{Bool})$$

$$\frac{}{x{:}C \vdash^{0}_{0} x : C} \quad (\mathrm{LF}\vdash_\diamond\textsc{Var})$$

$$\frac{\Gamma_1 \vdash^{p}_{p_0} e_1 : A \qquad \Gamma_2, x{:}A \vdash^{p_0}_{p'} e_2 : C}{\Gamma_1, \Gamma_2 \vdash^{p}_{p'} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : C} \quad (\mathrm{LF}\vdash_\diamond\textsc{Let})$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_k) \xrightarrow{p \triangleright p'} C}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash^{p}_{p'} f(x_1, \ldots, x_k) : C} \quad (\mathrm{LF}\vdash_\diamond\textsc{Fun})$$

$$\frac{\Gamma \vdash^{p}_{p'} e_t : C \qquad \Gamma \vdash^{p}_{p'} e_f : C}{\Gamma, x{:}\mathsf{bool} \vdash^{p}_{p'} \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : C} \quad (\mathrm{LF}\vdash_\diamond\textsc{If})$$

$$\frac{}{x_1{:}A, x_2{:}B \vdash^{0}_{0} (x_1, x_2) : A \times B} \quad (\mathrm{LF}\vdash_\diamond\textsc{Pair})$$

$$\frac{\Gamma, x_1{:}A, x_2{:}B \vdash^{p}_{p'} e : C}{\Gamma, x{:}A \times B \vdash^{p}_{p'} \mathtt{match}\, x\ \mathtt{with}\ (x_1, x_2) \to e : C} \quad (\mathrm{LF}\vdash_\diamond\text{E-}\textsc{Pair})$$

$$\frac{}{x{:}A \vdash^{q_l + \textsc{TypSize}(\mathsf{bool} \times A)}_{0} \mathtt{Inl}(x) : (A, q_l) + (B, q_r)} \quad (\mathrm{LF}\vdash_\diamond\textsc{Inl})$$

$$\frac{}{x{:}B \vdash^{q_r + \textsc{TypSize}(\mathsf{bool} \times B)}_{0} \mathtt{Inr}(x) : (A, q_l) + (B, q_r)} \quad (\mathrm{LF}\vdash_\diamond\textsc{Inr})$$

$$\frac{\Gamma, y{:}A \vdash^{p + q_l}_{p'} e_l : C \qquad \Gamma, z{:}B \vdash^{p + q_r}_{p'} e_r : C}{\Gamma, x{:}(A, q_l) + (B, q_r) \vdash^{p}_{p'} \mathtt{match}\ x\ \mathtt{with}\ |\mathtt{Inl}(y) \to e_l\, |\mathtt{Inr}(z) \to e_r : C} \quad (\mathrm{LF}\vdash_\diamond\text{E-}\textsc{Sum})$$

$$\frac{\Gamma, y{:}A \vdash^{\frac{p + q_l + \textsc{TypSize}(\mathsf{bool}\times A)}{p'}} e_l : C \qquad \Gamma, z{:}B \vdash^{\frac{p + q_r + \textsc{TypSize}(\mathsf{bool}\times B)}{p'}} e_r : C}{\Gamma, x{:}(A, q_l)+(B, q_r) \vdash^{\frac{p}{p'}} \texttt{match!}\, x\, \texttt{with}\, \texttt{|Inl(}y\texttt{)} \to e_l \texttt{|Inr(}z\texttt{)} \to e_r : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{E!\text{-}Sum})$$

$$\frac{}{\{\,\} \vdash^{\frac{0}{0}} \texttt{Nil} : \mathsf{list}(A, q)} \quad (\mathrm{LF} \vdash_\diamond \mathrm{Nil})$$

$$\frac{}{x_h{:}A, x_t{:}\mathsf{list}(A, q) \vdash^{\frac{q + \textsc{TypSize}(A\times\mathsf{list}(A,q))}{0}} \texttt{Cons(}x_h\texttt{,}x_t\texttt{)} : \mathsf{list}(A, q)} \quad (\mathrm{LF} \vdash_\diamond \mathrm{Cons})$$

$$\frac{\Gamma \vdash^{\frac{p}{p'}} e_1 : C \qquad \Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q) \vdash^{\frac{p+q}{p'}} e_2 : C}{\Gamma, x{:}\mathsf{list}(A, q) \vdash^{\frac{p}{p'}} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{E\text{-}List})$$

$$\frac{\Gamma \vdash^{\frac{p}{p'}} e_1 : C \qquad s = \textsc{TypSize}\big(A\times\mathsf{list}(A, q)\big) \qquad \Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q) \vdash^{\frac{p+q+s}{p'}} e_2 : C}{\Gamma, x{:}\mathsf{list}(A, q) \vdash^{\frac{p}{p'}} \texttt{match!}\, x\, \texttt{with}\, \texttt{|Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{E!\text{-}List})$$

Structural Rules

$$\frac{\Gamma \vdash^{\frac{p}{p'}} e : C}{\Gamma, x{:}A \vdash^{\frac{p}{p'}} e : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{Weak})$$

$$\frac{\Gamma \vdash^{\frac{q}{q'}} e : C \qquad p \geq q \qquad p - q \geq p' - q'}{\Gamma \vdash^{\frac{p}{p'}} e : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{Relax})$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{\frac{p}{p'}} e : C \qquad \curlyvee(A\,|A_1, A_2\,)}{\Gamma, z{:}A \vdash^{\frac{p}{p'}} e[z/x, z/y] : C} \quad (\mathrm{LF} \vdash_\diamond \mathrm{Share})$$

Definition 5.4 ($\mathrm{LF}_\diamond$ Sharing)**.** The ternary relation $\curlyvee$ is refined on $\mathrm{LF}_\diamond$ types by the rules shown in Figure 5.2

$$\frac{}{\curlyvee(\mathsf{unit}\,|\mathsf{unit},\mathsf{unit})}$$

$$\frac{}{\curlyvee(\mathsf{bool}\,|\mathsf{bool},\mathsf{bool})}$$

$$\frac{\curlyvee(A\,|A_1,A_2) \qquad \curlyvee(B\,|B_1,B_2)}{\curlyvee(A\times B\,|A_1\times B_1,A_2\times B_2)}$$

$$\frac{\curlyvee(A\,|A_1,A_2) \qquad \curlyvee(B\,|B_1,B_2) \qquad q=q_1+q_2 \qquad r=r_1+r_2}{\curlyvee\big((A,q)+(B,r)\,|(A_1,q_1)+(B_1,r_1),(A_2,q_2)+(B_2,r_2)\big)}$$

$$\frac{\curlyvee(A\,|A_1,A_2) \qquad q=q_1+q_2}{\curlyvee\big(\mathsf{list}(A,q)\,|\mathsf{list}(A_1,q_1),\mathsf{list}(A_2,q_2)\big)}$$

Figure 5.2: Sharing relation on $\mathrm{LF}_\diamond$ types

The meaning of the sharing relation depends once more on the definition of potential, Definition 5.7. Intuitively the statement $\curlyvee(A\,|A_1,A_2)$ expresses that the potential held by A is equal to the sum of the potential held by A_1 and A_2 (with respect to the same value).

We conclude this section by defining a well-typed $\mathrm{LF}_\diamond$ program in the same manner as for LF.

Definition 5.5 (Well-Typed $\mathrm{LF}_\diamond$ Program)**.** A program $\mathcal{P}$ is well-typed with respect to an $\mathrm{LF}_\diamond$ signature Σ if for all $f \in \mathrm{dom}(\mathcal{P})$ with $\Sigma(f) = (A_1,\ldots,A_k)\xrightarrow{p\triangleright p'}C$ and $\mathcal{P}(f) = (y_1,\ldots,y_k \to e_f)$ the typing judgement $[y_1{:}A_1,\ldots,y_k{:}A_k] \vdash^{p}_{p'} e_f{:}C$ can be derived by the $\mathrm{LF}_\diamond$ typing rules.

Lemma 5.6 (Embedding LF in $\mathrm{LF}_\diamond$)**.** *If the program $\mathcal{P}$ is well-typed with respect to* $\mathrm{LF}_\diamond$ *signature Σ, then it is also well-typed with respect to* LF *signature $|\Sigma|$.*

Proof. Every type rule of $\mathrm{LF}_\diamond$ has a strengthened precondition as compared to its counterpart type rule within LF. The only $\mathrm{LF}_\diamond$ type rule that has no counterpart LF type rule is $\mathrm{LF}\vdash_\diamond$Relax, which reduces to an identity within LF.

Therefore, if the statement $\Gamma \vdash^{p}_{p'} e{:}C$ is derivable in $\mathrm{LF}_\diamond$, so is the LF typing judgement $|\Gamma| \vdash e{:}|C|$. □

5.1.1 Exact Resource Analysis

The above type rules for $\mathrm{LF}_\diamond$ contain one entirely new rule, $\mathrm{LF} \vdash_\diamond$Relax, whose role is twofold. On one hand, it allows us to carry over excess resources, i.e. if we can type $\Gamma \vdash^{42}_{7} e{:}C$, then adding more potential (which may not actually be used) certainly causes no harm, hence we should be able to derive $\Gamma \vdash^{43}_{8} e{:}C$ as well.

On the other hand, it also allows us to discard potential. While this is something that we generally do not want, it might be necessary to allow us to type certain programs. Imagine, for example, a conditional, having two branches that will each use a different amount of heap memory. A static analysis such as ours, which cannot exclude the possibility that either branch is executed, must then assume the worst-case for the entire conditional term. The rule $\mathrm{LF} \vdash_\diamond$Relax allows us to discard the superfluous potential in the less expensive branch, either before or after examining that branch.

Note that the rule $\mathrm{LF} \vdash_\diamond$If correspondingly only allows branches having the same resource consumption. Therefore the application of $\mathrm{LF} \vdash_\diamond$Relax will be usually required for one of the branches. The benefit of having an explicit structural rule like $\mathrm{LF} \vdash_\diamond$Relax is of course a leaner and cleaner proof.

We distinguish between *exact bounds*, *minimal bounds* and *exact cost formulas*. An exact bound on resource consumption means that there is at least one evaluation that requires as many resources as the bound allows, i.e. the bound is met. A minimal bound is smaller than any other bound; they are often non-linear. An exact cost formula for the resource consumption implies that the bound is both an upper- and a lower-bound at the same time, i.e. the program always uses as many resources as predicted by the cost formula.

If one would prefer the analysis to yield an exact cost formula on resource consumption rather than an upper bound, one needs to change two things:

a) Replacing the rule $\mathrm{LF} \vdash_\diamond$Relax by the following rule

$$\frac{\Gamma \vdash^{p}_{p'} e{:}C \qquad q \in \mathbb{Q}^+}{\Gamma \vdash^{p+q}_{p'+q} e{:}C} \qquad (\mathrm{LF} \vdash_\diamond \textsc{Excess})$$

in order to prohibit the disposal of unused potential.

b) Removing (or altering) the weakening rule $\mathrm{LF} \vdash_\diamond$Weak. Discarding a boxed type from the context by weakening, i.e. without destroying it, will leak memory resources, except if the pointer is already **Null** or **Bad**. Forgetting a proper pointer by weakening will leak memory resources, unless the potential gained through it is zero and it is pointing to an aliased data structure that will be destroyed in an orderly way or returned later through another handle.

Since the above are all conditions to be checked at runtime, a simpler (statically viable), but still safe alternative would be to allow weakening only for stack-allocated types, i.e. unit, bool and products built from these types.

It is clear that these changes decrease the number of programs that the analysis can be successfully applied to. For example, programs that require a different amount of resources for different inputs, despite those inputs have the same structure and number of nodes, may then not be typable anymore.

We remark that our implementation of the amortised analysis will generally apply the type rule $\mathrm{LF} \vdash_\diamond$Relax at each possible step. This causes many inequalities to appear within the generated linear program. However, if none of the inequalities is strict under a given solution, then we know that our inferred cost formula happens to be exact for all evaluations of the examined program.

However, the converse is not true: a solution having strict constraints may still yield an exact cost formula (as opposed to an upper bound). In such a case the strict inequalities can all be derived by the rule $\mathrm{LF} \vdash_\diamond$Excess shown above. Whether or not this is the case is easy to determine, even for an LP generated using $\mathrm{LF} \vdash_\diamond$Relax. All one has to check is whether the second inequality produced by the applications of rule $\mathrm{LF} \vdash_\diamond$Relax becomes an equality under the given solution.

5.2 Potential

The definition of potential expresses the data dependency of the cost bound. It shows us how to map concrete input values to cost values, as formally expressed by Theorem 1 in the next section.

$$
\begin{array}{lcl}
\Phi_{\mathcal{H}}(() : \mathsf{unit}) & = & 0 \\
\Phi_{\mathcal{H}}(\mathbf{tt} : \mathsf{bool}) & = & 0 \\
\Phi_{\mathcal{H}}(\mathbf{ff} : \mathsf{bool}) & = & 0 \\
\Phi_{\mathcal{H}}\big((v, w) : A \times B\big) & = & \Phi_{\mathcal{H}}(v : A) + \Phi_{\mathcal{H}}(w : B) \\
\Phi_{\mathcal{H}}(\ell : (A, q) + (B, p)) & = & \begin{cases} q + \Phi_{\mathcal{H}}(v : A) & \text{If } \mathcal{H}(\ell) = (\mathbf{tt}, v) \\ p + \Phi_{\mathcal{H}}(v : B) & \text{If } \mathcal{H}(\ell) = (\mathbf{ff}, v) \\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Bad} \end{cases} \\
\Phi_{\mathcal{H}}(\ell : \mathsf{list}(A, q)) & = & \begin{cases} q + \Phi_{\mathcal{H}\setminus\ell}\big((v_h, v_t) : A \times \mathsf{list}(A, q)\big) & \text{If } \mathcal{H}(\ell) = (v_h, v_t) \\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Null} \\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Bad} \end{cases}
\end{array}
$$

Figure 5.3: Potential

Definition 5.7 (Potential)**.** The potential Φ : Heap $\times$ Val $\times$ LF$_\diamond$-type $\to \mathbb{Q}^+$ is a map from a type, a value and a heap to a non-negative rational number. We define the potential of a value v of LF$_\diamond$ type A within heap $\mathcal{H}$ recursively by the equations shown in Figure 5.3.

We extend this definition to Φ : Heap $\times$ Stack $\times$ LF$_\diamond$-type-context $\to \mathbb{Q}^+$ naturally by summation.

$$\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi_{\mathcal{H}}(\mathcal{S}(x) : \Gamma(x))$$

So the overall potential is a sum over all the nodes of a data structure, where each node contributes potential according to its kind and the annotation within its type. Any node may contribute several times to this sum with different annotated types, if the data structure is aliased.

It is very important to note that the potential can be easily computed for huge classes of input data. For example, if the input is of type $\mathsf{list}(\mathsf{list}(\mathsf{bool}, 2), 3)$, then we know instantaneously that the potential for a value of this type is three times the length of

the outer list, plus two times the sum of the lengths of the inner boolean lists. The bound is very easy to understand and encompasses all possible inputs.

For a cost analysis to be of any use, it is paramount that we can easily glean the overall cost behaviour of the analysed code. It is not just enough to make it easier to compute the evaluation cost for a specific input data than just running the program on this input, since then one still has to consider all possible input values separately, which is usually infeasible.

5.2.1 Generalisation to Arbitrary Recursive Data Types

The reason for adding annotations only to sums and lists (and functions) is that the number and kind of nodes for values of these types may vary and are only known as we process these types. That is, the potential only becomes available again for direct use if we have matched against them and know for sure whether there is another node of a certain kind or not.

If we were to add annotations everywhere, we would only introduce ambiguity without increasing expressivity. For example, the overall potential contributed by any value of the hypothetical type $\big((\mathsf{bool},1)\times(\mathsf{bool},2)\,,0\big)$ would always be equivalent to the overall potential contributed by any element of hypothetical type $\big((\mathsf{bool},0)\times(\mathsf{bool},0)\,,3\big)$. Both types will always have the same number of nodes per type, regardless of the underlying operational value. Therefore the potential will be 3 for all values that fit either type above. We see that we can always shift the potential outwards to the next enclosing type that has a varying number of nodes, such as a list of type $\mathsf{list}(\mathsf{bool}\times\mathsf{bool},3)$. If there is no enclosing type having a varying number of nodes, then the function type can carry the annotation. So, for example, the hypothetical type $\big((\mathsf{bool},1)\times(\mathsf{bool},2)\,,3\big)\xrightarrow{4\triangleright 5}(\mathsf{bool},5)$ would equivalent to the type $\big((\mathsf{bool},0)\times(\mathsf{bool},0)\,,0\big)\xrightarrow{10\triangleright 10}(\mathsf{bool},0)$.

We remark that the type system in [Hof00] could not abstract away this ambiguity, since potential was represented by first-class objects, namely pointers of the special type $\diamond$. Clearly, the type $\mathsf{list}(A\times\diamond)$ is different from the type $\mathsf{list}(\diamond\times A)$ then, but both are equivalent to the $\mathrm{LF}_\diamond$ type $\mathsf{list}(A,1)$.

Treating general user-defined recursive data types is straightforward, but requires a clumsy notation, as shown in our work [JLH+09]. Since we discuss an elaborate

example that uses user-defined data types later on in Section 5.6, we outline the principal idea here.

In general, one should attach an individual annotation to *each constructor* of a type. This annotation denotes the potential bound upon construction and released upon destruction of that particular constructor of that annotated type. The idea is that the matching of a constructor decides upon the branch taken in the program, and since each computational branch has its individual cost, each constructor should bring along its individual potential to amortise it.

The two annotations for the sum type naturally arise this way. The single annotation for lists in $\mathrm{LF}_\diamond$ is precisely the annotation of the `Cons()` constructor only, while the annotation of the `Nil` constructor is omitted due to ambiguity. Every well-typed value of a list type has precisely one `Nil` constructor, therefore any potential assigned to the `Nil` constructor can be safely shifted outwards, as described in the above paragraph. However, our implementation rigorously assigns one annotation per constructor, including the `Nil` constructor.

5.2.2 Potential Properties

We continue with three technical observations.

Lemma 5.8 (Potential Deallocation)**.** *For any heap $\mathcal{H}$, stack $\mathcal{S}$ and context Γ, if $\mathcal{H} \vDash \mathcal{S}{:}\Gamma$ holds, then the potential $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma)$ is well-defined and furthermore for all $\ell \in \mathrm{dom}(\mathcal{H})$ holds $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) \geq \Phi_{\mathcal{H}[\ell \mapsto \mathbf{Bad}]}(\mathcal{S} : \Gamma)$*

Proof. By the definition of potential for stacks and contexts it suffices to prove $\Phi_{\mathcal{H}}(v : A) \geq \Phi_{\mathcal{H}[\ell \mapsto \mathbf{Bad}]}(v : A)$ for an arbitrary value v and type A such that $\mathcal{H} \vDash v{:}A$. The proof is then by induction on both the complexity of type A and the size of the heap $\mathcal{H}$, with the size of the heap taking priority over the complexity of the type. The claim is vacuously true for most cases, since they do not inspect heap $\mathcal{H}$. Note though that the complexity of the type decreases in the case of pairs and sums. The only non-trivial case is the potential of a non-empty list. Here, the complexity of the type increases, but the domain of the heap is diminished, allowing us to apply the induction hypothesis thanks to Lemma 4.12.

Furthermore by inspection of Definition 5.7 we know that the potential is always a sum of non-negative rationals and thus can only diminish by overwriting a location with the value **Bad**, which will always contribute a potential of zero to the sum for any $\mathrm{LF}_\diamond$ type. □

One should not be misled by this lemma. The overall potential *may increase* during evaluation (e.g. through deallocation). However, this potential must then be associated with the result of the computation; while the potential of the input can only be consumed. The following lemma also makes use of this fact, by conveniently ignoring the potential that is contributed by the result of the evaluation.

Lemma 5.9 (Non-Increasing Potential)**.** *Fix a well-typed* $\mathrm{LF}_\diamond$ *program* $\mathcal{P}$ *with signature* Σ*. If* $\Gamma \vdash^{p}_{p'} e{:}C$ *and* $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ *and* $\mathcal{H} \vDash \mathcal{S} : \Gamma$ *holds, then* $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) \geq \Phi_{\mathcal{H}'}(\mathcal{S} : \Gamma)$ *is true as well.*

Proof. Since both context Γ and stack $\mathcal{S}$ remain are unchanged, the sums for $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma)$ and $\Phi_{\mathcal{H}'}(\mathcal{S} : \Gamma)$ range over the same locations with the same type. By Lemma 4.8, the only changes between $\mathcal{H}$ and $\mathcal{H}'$ are either added locations or changing the value of a location to **Bad**. In the first, the added locations cannot affect the sum, since they are not ranged over. The latter can only lower the sum, according to Lemma 5.8. □

Note that the previous two lemmas only hold due to the functional setting and the absence of heap updates via pointers (as frequently found in imperative code). While these two lemmas greatly simplify our burden of proof, they are not strictly necessary, as our successful treatment of imperative code, including heap update, shows [HJ06].

The last lemma of this section states that sharing preserves potential, i.e. that the potential of the original entity is always equal to the sum of its shared parts.

Lemma 5.10 (Shared Potential)**.** *If* $\mathcal{H} \vDash v : A$ *and* $\curlyvee(A \,|A_1, A_2)$ *holds, then* $\Phi_{\mathcal{H}}(v : A) = \Phi_{\mathcal{H}}(v : A_1) + \Phi_{\mathcal{H}}(v : A_2)$.

Proof. The proof is by induction on the length of the derivation for $\curlyvee(A \,|A_1, A_2)$.

The claim is trivial for the base types unit and bool, since they never carry any potential by Definition 5.7.

For pair types, the claim follows straightforwardly from induction, since the potential of a product type is defined to be equal to the sum of the potential of its parts.

For sum types, we have $\curlyvee\big((B, q) + (C, r) \,|(B_1, q_1) + (C_1, r_1)\,, (B_2, q_2) + (C_2, r_2)\big)$ and both $q = q_1 + q_2$ and $r = r_1 + r_2$. Without loss of generality, we assume $\mathcal{H}(v) = (\mathbf{tt}, w)$, since the other case follows then by symmetry. Thus we have

$$\begin{aligned}\Phi_{\mathcal{H}}(v : (B, q) + (C, r)) &= q + \Phi_{\mathcal{H}}(w : B)\\ &= (q_1 + q_2) + \Phi_{\mathcal{H}}(w : B)\\ &= (q_1 + \Phi_{\mathcal{H}}(w : B_1)) + (q_2 + \Phi_{\mathcal{H}}(w : B_2))\end{aligned}$$

where the last equality follows from $\Phi_{\mathcal{H}}(w : B) = \Phi_{\mathcal{H}}(w : B_1) + \Phi_{\mathcal{H}}(w : B_2)$ which we obtained by using the induction hypothesis for B.

For the last case of list types we have $\curlyvee(\mathsf{list}(B, q)\,|\mathsf{list}(B_1, q_1), \mathsf{list}(B_2, q_2))$ and $q = q_1 + q_2$. The premise of memory consistency implies that v points to a chain of cons nodes, which is either terminated by **Null** or **Bad**. Without loss of generality, we assume that this list has length n and that the values in the cons nodes are $v_1, \ldots, v_n$. Whether it is terminated by **Null** or **Bad** does not matter, since both values will always have a potential of zero. Thus, we observe

$$\begin{aligned}
\Phi_{\mathcal{H}}(v : \mathsf{list}(B, q)) &= \sum_{i=1}^{n} \big(q + \Phi_{\mathcal{H}}(v_i : B)\big) \\
&= nq + \sum_{i=1}^{n} \Phi_{\mathcal{H}}(v_i : B) \\
&= n(q_1 + q_2) + \sum_{i=1}^{n} \Phi_{\mathcal{H}}(v_i : B_1) \; + \sum_{i=1}^{n} \Phi_{\mathcal{H}}(v_i : B_2) \\
&= \sum_{i=1}^{n} \big(q_1 + \Phi_{\mathcal{H}}(v_i : B_1)\big) \; + \sum_{i=1}^{n} \big(q_2 + \Phi_{\mathcal{H}}(v_i : B_2)\big)
\end{aligned}$$

where the penultimate inequality requires the application of the induction hypothesis n-times. □

5.3 Soundness of the First-Order Analysis

We will now formulate and prove the main theorem of this chapter of the thesis. As opposed to Lemma 5.8, the following theorem captures the overall change of potential for the entire heap during the evaluation. This theorem thus allows us to statically determine the overall heap consumption of a program during its entire (finite) evaluation. We will later on prove in Theorem 2 that even non-terminating programs will obey our predicted bounds.

Theorem 1 (Soundness LF)**.** *Fix a well-typed* $\mathrm{LF}_\diamond$ *program* $\mathcal{P}$ *with signature* Σ*. If*

$$\Gamma \vdash^{p}_{p'} e{:}C \tag{1.A}$$

$$\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}' \tag{1.B}$$

$$\mathcal{H} \vDash \mathcal{S} : \Gamma \tag{1.C}$$

then for all $r \in \mathbb{Q}^+$ *and for all* $m \in \mathbb{N}$ *such that* $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + r$ *there exists* $m' \in \mathbb{N}$ *satisfying* $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$ *and* $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'$*.*

Proof. The proof is by induction on the lengths of the derivations of (1.B) and (1.A) ordered lexicographically with the derivation of the evaluation taking priority over the typing derivation. The reason is identical to the one stated in the proof of Lemma 4.13: Induction over the length of the typing derivation alone would not succeed, since the case for function application required us to use the induction hypothesis on the defining body of the function, whose type is obtained by the signature. The condition that the program $\mathcal{P}$ is well-typed allows us to assume an $\mathrm{LF}_\diamond$ typing derivation for the function body, but the length of this typing derivation is of course arbitrary. However, the evaluation of the entire program does progress in this case, so we can use induction on its length.

Similarly, induction over the length of the evaluation alone would fail, since the cases for all structural rules requires the application of the induction hypothesis without any change to the evaluation statement (1.B). In these cases, only the length of the typing derivation was shortened by one step.

We first treat the cases where the last step in the typing derivation was obtained by application of a structural rule, which are all the cases which leave the length of the derivation for the evaluation unchanged, but reduce the size of the type derivation by one step. The other cases use the induction hypothesis only for evaluations with strictly decreased sizes.

For completeness, the proof is written here in full. However, not all cases are interesting. We therefore advise the reader to focus attention on the cases $\mathrm{LF}\downarrow$Let, $\mathrm{LF}\downarrow$Fun, $\mathrm{LF}\downarrow$Cons, $\mathrm{LF}\downarrow$E-Cons, $\mathrm{LF}\vdash_\diamond$Relax and $\mathrm{LF}\vdash_\diamond$Share.

Since the projection of $\mathrm{LF}_\diamond$ to LF is trivial, we will not always explicitly mention the applications of Lemma 4.19 and Lemma 5.6 for brevity and readability. These two Lemmas simply state that the deletion of cost annotations preserve the derivations for the operational semantics and the typing respectively.

$\mathrm{LF}\vdash_\diamond$Weak: Suppose that the last applied rule in the typing derivation was the rule $\mathrm{LF}\vdash_\diamond$Weak. Let $\Gamma = \Delta, x{:}A$. We apply the induction hypothesis for $\Delta \vdash^{p}_{p'} e{:}C$, with both (1.B) and (1.C) remaining unchanged. Thus we have for all $r \in \mathbb{Q}^+$ and $m \in \mathbb{N}$ with $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Delta) + r$, that there exists $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$ with $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'$. The desired result is already subsumed therein, as a further restriction to the forall-quantified m: $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) = \Phi_{\mathcal{H}}(\mathcal{S} : \Delta) + \Phi_{\mathcal{H}}(\mathcal{S}(x) : A) \geq \Phi_{\mathcal{H}}(\mathcal{S} : \Delta)$ since the potential is always non-negative by its definition (Definition 5.7).

LF $\vdash_\diamond$RELAX: This case illustrates the use of r within the theorem, which is a necessary tool in the let-rule case.

Fix an arbitrary $r \in \mathbb{Q}^+$. In order to apply the induction hypothesis we then choose $r' = \big((p-q)+r\big)$, with $r' \in \mathbb{Q}^+$ as required by $p \geq q$ following from the premise of rule LF $\vdash_\diamond$RELAX. Now applying the induction hypothesis for $\Gamma \vdash^{q}_{q'} e{:}C$, with both (1.B) and (1.C) remaining unchanged, yields for all $m \in \mathbb{N}$ with $m \geq q + \Phi_{\mathcal{H}}(\mathcal{S} : \Delta) + (p - q + r) = p + \Phi_{\mathcal{H}}(\mathcal{S} : \Delta) + r$ there exists

$$\begin{aligned} m' &\geq q' + \Phi_{\mathcal{H}'}(v : C) + (p - q + r) \\ &\geq q' + \Phi_{\mathcal{H}'}(v : C) + (p' - q' + r) \end{aligned}$$

(using the premise $p - q \geq p' - q'$ of rule LF $\vdash_\diamond$RELAX). Hence we have

$$m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$$

allowing for $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'$ as required.

LF $\vdash_\diamond$SHARE: Thanks to Lemma 5.10, we know that sharing does not affect the overall potential. Hence this case follows trivially by induction.

LF $\downarrow$UNIT: By premise (1.A) we have $\{\,\} \vdash^{0}_{0} *{:}\mathsf{unit}$, hence $p = 0$ and $\Gamma = \{\,\}$ in terms of the formulation of the theorem. From premise (1.B) we have $\mathcal{S}, \mathcal{H} \vdash * \downarrow \mathbf{Null}, \mathcal{H}$ and thus $\mathcal{H} = \mathcal{H}'$ as well.

Fix $r \in \mathbb{Q}^+$ arbitrarily and fix any $m \in \mathbb{N}$ such that $m \geq 0 + 0 + r$, since $\Phi_{\mathcal{H}}(\mathcal{S} : \{\,\}) = 0$ by the definition of potential (Definition 5.7 on page 87). Thus we choose $m' = m$, which satisfies both $m' \geq 0 + \Phi_{\mathcal{H}}(\mathbf{Null} : \mathsf{unit}) + r = r$ and the additional requirement of rule LF $\downarrow^\diamond$UNIT to derive $\mathcal{S}, \mathcal{H} \vdash^{m}_{m} * \downarrow \mathbf{Null}, \mathcal{H}$ as needed.

LF $\downarrow$TRUE & LF $\downarrow$FALSE: These two cases are analogous to the proof of the previous case for LF $\downarrow$UNIT, since all concerned rules are structurally identical and $\Phi_{\mathcal{H}}(\mathbf{Null} : \mathsf{unit}) = \Phi_{\mathcal{H}}(\mathbf{tt} : \mathsf{bool}) = \Phi_{\mathcal{H}}(\mathbf{ff} : \mathsf{bool}) = 0$ holds regardless of heap $\mathcal{H}$ by Definition 5.7.

LF $\downarrow$VAR: The term e is a simple variable in this case. Fix $r \in \mathbb{Q}^+$ arbitrarily and fix any $m \in \mathbb{N}$ such that $m \geq 0 + \Phi_{\mathcal{H}}(\mathcal{S} : \{x{:}C\}) + r$. We set $m' = m \geq 0 + \Phi_{\mathcal{H}}(\mathcal{S}(x) : C) + r$. By (1.B) and rule LF $\downarrow^\diamond$VAR we then derive $\mathcal{S}, \mathcal{H} \vdash^{m}_{m} x \downarrow \mathcal{S}(x), \mathcal{H}$ as required.

LF ↓Let: Fix $r \in \mathbb{Q}^+$ arbitrarily. Fix any $m \in \mathbb{N}$ such that

$$\begin{aligned} m &\geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1, \Gamma_2) + r \\ &\geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1) + \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) + r \end{aligned}$$

where the last inequality follows by Lemma 5.9 and (1.B). From (1.C) follows $\mathcal{H} \vDash \mathcal{S} : \Gamma_1$, so we can apply the induction hypothesis choosing $r_0 = \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) + r$ and obtain for all $m \in \mathbb{N}$ such that $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1) + r_0$ that there exists $m_0 \in \mathbb{N}$ satisfying $m_0 \geq p_0 + \Phi_{\mathcal{H}_0}(v_0 : A) + r_0$ and eventually $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m_0}{\vdash}} e_1 \downarrow v_0, \mathcal{H}_0$.

We can apply Lemma 4.13 to obtain $\mathcal{H}_0 \vDash \mathcal{S}[x{\mapsto}v_0]{:}\Gamma_2[x{\mapsto}A]$. Hence we apply the induction hypothesis a second time: Since $m_0 \geq p_0 + \Phi_{\mathcal{H}_0}(v_0 : A) + \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) + r = p_0 + \Phi_{\mathcal{H}_0}(\mathcal{S}[x{\mapsto}v_0] : \Gamma_2[x{\mapsto}A]) + r$ from above, there exists $m' \in \mathbb{N}$ satisfying $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$ and $\mathcal{S}[x{\mapsto}v_0], \mathcal{H} \overset{m_0}{\underset{m'}{\vdash}} e_2 \downarrow v, \mathcal{H}'$. Combining the above by rule LF ↓$^\diamond$Let then yields $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \downarrow v, \mathcal{H}'$ as desired.

LF ↓Fun: By the assumption that $\mathcal{P}$ is a well-typed $\mathrm{LF}_\diamond$ program with signature Σ (Definition 4.5 on page 46), we have $[y_1{:}A_1, \ldots, y_k{:}A_k] \overset{p}{\underset{p'}{\vdash}} e_f{:}C$. Note that $[y_1{:}A_1, \ldots, y_k{:}A_k]$ is just a renamed version of Γ. Furthermore the stack $[y_1{\mapsto}v_1, \ldots, y_k{\mapsto}v_k]$ is clearly a renamed fragment of $\mathcal{S}$; with $\forall z \in \mathrm{dom}(\Gamma)\,.\, z \in \mathrm{dom}\big([y_1{\mapsto}v_1, \ldots, y_k{\mapsto}v_k]\big)$. Hence the desired result follows directly from the induction hypothesis in this case. Note that the induction hypothesis is indeed applicable, since the derivation of (1.B) has in fact been shortened and takes precedence over the derivation of (1.A), which has grown in this case.

LF ↓If-T & LF ↓If-F: We consider the case of rule LF ↓If-T first, the case for rule LF ↓If-F follows then similarly.

Fix $r \in \mathbb{Q}^+$ arbitrarily and $m \in \mathbb{N}$ such that $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}\mathsf{bool}) + r = p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}\mathsf{bool}) + r$. Applying the induction hypothesis for $\Gamma \overset{p}{\underset{p'}{\vdash}} e_t{:}C$ from (1.A) and $\mathcal{S}, \mathcal{H} \vdash e_t \downarrow v, \mathcal{H}'$ from (1.B) yields m' with $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$ and $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e_t \downarrow v, \mathcal{H}'$. Hence by the rule LF ↓$^\diamond$If-T also $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'$ follows as required.

LF ↓Pair: The term e is pair of variables in this case. Fix $r \in \mathbb{Q}^+$ arbitrarily and any $m \in \mathbb{N}$ such that $m \geq 0 + \Phi_{\mathcal{H}}(\mathcal{S} : x_1{:}A_1, x_2{:}A_2) + r$. Rule LF ↓$^\diamond$Pair requires $m' = m$, hence it suffices to show that $0 + \Phi_{\mathcal{H}}(\mathcal{S} : \{x_1{:}A_1, x_2{:}A_2\}) + r \geq 0 + \Phi_{\mathcal{H}}\Big(\big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big) : A_1{\times}A_2\Big) + r$. This follows from the definition of potential (5.7) by $\Phi_{\mathcal{H}}(\mathcal{S} : \{x_1{:}A_1, x_2{:}A_2\}) = \Phi_{\mathcal{H}}\big(\mathcal{S}(x_1) : A_1\big) + \Phi_{\mathcal{H}}\big(\mathcal{S}(x_2) : A_2\big) = \Phi_{\mathcal{H}}\Big(\big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big) : A_1{\times}A_2\Big)$.

LF ↓E-Pair: This is the case for pair elimination. Fix any $r \in \mathbb{Q}^+$ arbitrarily and choose any $m \in \mathbb{N}$ such that $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}A{\times}B\}) + r = p + \Phi_{\mathcal{H}}(\mathcal{S}[x_1{\mapsto}v_1, x_2{\mapsto}v_2] : \{\Gamma, x_1{:}A, x_2{:}B\}) + r$, where the last equality follows from $\mathcal{S}(x) = (v_1, v_2)$ and and $\Phi_{\mathcal{H}}\big(\mathcal{S}(x_1) : A\big) + \Phi_{\mathcal{H}}\big(\mathcal{S}(x_2) : B\big) = \Phi_{\mathcal{H}}\Big(\big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big) : A{\times}B\Big)$ from the definition of potential (Definition 5.7 on page 87).

We apply the induction hypothesis for $\mathcal{S}[x_1{\mapsto}v_1, x_2{\mapsto}v_2], \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \downarrow v, \mathcal{H}'$ and obtain m' with $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$ already completing the proof of this case.

LF ↓Inl & LF ↓Inr: Without loss of generality, we only consider the case of rule LF ↓Inl, since the case for rule LF ↓Inr is almost identical.

Fix $r \in \mathbb{Q}^+$ arbitrarily and any $m \in \mathbb{N}$ such that

$$m \geq \big(q_l + \textsc{TypSize}(\mathsf{bool}{\times}A)\big) + \Phi_{\mathcal{H}}(\mathcal{S}(x) : A) + r$$

By Lemma 4.16 and 1.C we have $\textsc{TypSize}(\mathsf{bool}{\times}A) = \textsc{Size}(\mathbf{tt}, w')$ and thus

$$\begin{aligned} m &\geq q_l + \textsc{Size}(\mathbf{tt}, w') + \Phi_{\mathcal{H}}(\mathcal{S}(x) : A) + r \\ m - \textsc{Size}(\mathbf{tt}, w') &\geq 0 + \Phi_{\mathcal{H}[\ell \mapsto (\mathbf{tt}, w')]}(\ell : (A, q_l) + (B, q_r)) + r \end{aligned}$$

by Definition 5.7 as required, since $m' = m - \textsc{Size}(\mathbf{tt}, w')$ by rule LF ↓Inl.

LF ↓E-Inl, LF ↓E-Inr, LF ↓E!-Inl & LF ↓E!-Inr: We concentrate on the case for LF ↓E!-Inl, with the other cases following similarly.

Fix $r \in \mathbb{Q}^+$ arbitrarily and any $m \in \mathbb{N}$ such that

$$m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}\,(A, q_l) + (B, q_r)) + r$$

By premise (1.B) we have $\mathcal{S}(x) = \ell_0$ and $\mathcal{H}(\ell_0) = w$ and $w = (\mathbf{tt}, w')$, therefore by the definition of potential

$$\begin{aligned} m &\geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + q_l + \Phi_{\mathcal{H}}(w' : A) + r \\ m + \textsc{Size}(w) &\geq p + q_l + \textsc{TypSize}\big(\mathsf{bool}{\times}A\big) + \Phi_{\mathcal{H}}(\mathcal{S}[y{\mapsto}w'] : \Gamma, y{:}A) + r \end{aligned}$$

where the last inequality follows by $\textsc{TypSize}\big(\mathsf{bool}{\times}A\big) = \textsc{Size}\,(w)$ obtained through Lemma 4.16 and premise (1.C). We can thus apply the induction hypothesis for $\mathcal{S}[y{\mapsto}w'], \mathcal{H}[x \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(w)}{\underset{m'}{\vdash}} e_l \downarrow v, \mathcal{H}'$ to obtain

$$m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$$

as required by rule LF ↓$^\diamond$E!-Inl.

LF ↓Nil: We have $\{\,\} \vdash^{0}_{0} \texttt{Nil}{:}\mathsf{list}(A,q)$ from (1.A) and $\mathcal{S},\mathcal{H} \vdash \texttt{Nil} \downarrow \mathbf{Null},\mathcal{H}$ from (1.B). Fix $r \in \mathbb{Q}^+$ arbitrarily and fix any $m \in \mathbb{N}$ such that $m \geq 0 + 0 + r$. By rule LF ↓$^\diamond$Nil we have $\mathcal{S},\mathcal{H} \vdash^{m}_{m} \texttt{Nil} \downarrow \mathbf{Null},\mathcal{H}$ and thus require $m' = m$, which satisfies $m' \geq 0 + \Phi_{\mathcal{H}}(\mathbf{Null} : \mathsf{list}(A,q)) + r = r$ as needed.

LF ↓Cons: Let $s = \textsc{TypSize}\big(A{\times}\mathsf{list}(A,q)\big)$. By the premise (1.A) we have that $x_h{:}A, x_t{:}\mathsf{list}(A,q) \vdash^{q+s}_{0} \texttt{Cons}(x_h,x_t){:}\mathsf{list}(A,q)$ and from (1.B) we know that $\mathcal{S},\mathcal{H} \vdash \texttt{Cons}(x_h,x_t) \downarrow \ell, \mathcal{H}[\ell \mapsto v]$ with $v = \big(\mathcal{S}(x_h),\mathcal{S}(x_t)\big)$ and $\ell \notin \mathrm{dom}(\mathcal{H})$.

Fix $r \in \mathbb{Q}^+$ arbitrarily and fix any $m \in \mathbb{N}$ such that

$$m \geq (q+s) + \Phi_{\mathcal{H}}\big(\mathcal{S} : \{x_h{:}A, x_t{:}\mathsf{list}(A,q)z\}\big) + r$$

We now choose $m' = m - s$, hence we have

$$\begin{aligned} m' &\geq q + \Phi_{\mathcal{H}}\big(\mathcal{S} : \{x_h{:}A, x_t{:}\mathsf{list}(A,q)z\}\big) + r \\ &= q + \Phi_{\mathcal{H}}(\mathcal{S}(x_h) : A) + \Phi_{\mathcal{H}}(\mathcal{S}(x_t) : \mathsf{list}(A,q)) + r \\ &= q + \Phi_{\mathcal{H}}(v : A{\times}\mathsf{list}(A)) + r \\ &= 0 + \Phi_{\mathcal{H}[\ell\mapsto v]}(\ell : \mathsf{list}(A,q)) + r \end{aligned}$$

as required, where the equations follow directly from the definition of potential (5.7).

Now in order to prove that $\mathcal{S},\mathcal{H} \vdash^{m}_{m'} \texttt{Cons}(x_h,x_t) \downarrow \ell, \mathcal{H}[\ell \mapsto v]$ holds by virtue of rule LF ↓$^\diamond$Cons, it remains to remark that $m = m' + s$ and furthermore $s = \textsc{Size}(v)$ by Lemma 4.16.

LF ↓E-Nil & LF ↓E!-Nil: We only treat the case for LF ↓E-Nil without loss of generality, since deallocation has no effect on an empty list, which is represented by a null pointer.

Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ with $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A,q)\}) + r$. From premise (1.A) we have $\Gamma \vdash^{p}_{p'} e_1{:}C$ and from (1.B) we obtain both $\mathcal{S},\mathcal{H} \vdash e_1 \downarrow v, \mathcal{H}'$. and also $\mathcal{S}(x) = \mathbf{Null}$. Hence by Definition 5.7 we deduce that $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + r$ and apply the induction hypothesis for e_1 to obtain $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$. and $\mathcal{S},\mathcal{H} \vdash^{m}_{m'} e_1 \downarrow v, \mathcal{H}'$ as required.

LF ↓E-Cons: Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ with

$$m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A,q)\}) + r$$

From (1.B) we obtain $\mathcal{S}(x) = \ell$ and $\mathcal{H}(\ell) = v$ and $v = (v_h, v_t)$, hence by using Definition 5.7 three times we have

$$\begin{aligned} m &\geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi_{\mathcal{H}}(v : A{\times}\mathsf{list}(A,q)) + r \\ &= p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + q + \Phi_{\mathcal{H}}(v_h : A) + \Phi_{\mathcal{H}}(v_t : \mathsf{list}(A,q)) + r \\ &= p + q + \Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q)\}) + r \end{aligned}$$

since $x_h, x_t \notin \mathrm{dom}(\Gamma)$. We have $\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q) \vdash^{p+q}_{p'} e_2{:}C$ from premise (1.A), and using (1.B) we obtain $\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t], \mathcal{H} \vdash e_2 \downarrow v, \mathcal{H}'$. Applying the induction hypothesis for e_2 now yields $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$. and $\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t], \mathcal{H} \vdash^{m}_{m'} e_2 \downarrow v, \mathcal{H}'$ as required for rule LF$\downarrow^{\diamond}$E-Cons.

LF$\downarrow$E!-Cons: Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ with

$$m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A,q)\}) + r$$

From (1.B) we obtain $\mathcal{S}(x) = \ell$ and $\mathcal{H}(\ell) = v$ and $v = (v_h, v_t)$, hence by using Definition 5.7 three times we have

$$\begin{aligned} m &\geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi_{\mathcal{H}}(v : A{\times}\mathsf{list}(A,q)) + r \\ &= p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + q + \Phi_{\mathcal{H}}(v_h : A) + \Phi_{\mathcal{H}}(v_t : \mathsf{list}(A,q)) + r \\ &= p + q + \Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q)\}) + r \end{aligned}$$

since $x_h, x_t \notin \mathrm{dom}(\Gamma)$. Now by applying Lemma 5.8, we deduce

$$m \geq p + q + \Phi_{\mathcal{H}[\ell\mapsto\mathbf{Bad}]}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q)\}) + r$$

Let $s = \textsc{TypSize}\big(A{\times}\mathsf{list}(A,q)\big)$. In addition we also have $s = \textsc{Size}(v) \geq 0$ by Lemma 4.16. Hence by adding s to both sides we have

$$m{+}s \geq p + q + s + \Phi_{\mathcal{H}[\ell\mapsto\mathbf{Bad}]}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q)\}) + r$$

From premise (1.A) we have furthermore $\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A,q) \vdash^{p+q+s}_{p'} e_2{:}C$ and using (1.B) we obtain $\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t], \mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vdash e_2 \downarrow v, \mathcal{H}'$. Applying the induction hypothesis for e_2 now yields $m' \geq p' + \Phi_{\mathcal{H}'}(v : C) + r$. and $\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t], \mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vdash^{m+s}_{m'} e_2 \downarrow v, \mathcal{H}'$ as required for LF$\downarrow^{\diamond}$E!-Cons.

□

5.4 Misbehaving Programs and Non-Termination

Theorem 1 requires as a premise that $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ holds. This is somewhat unsatisfactory, as it requires that the term e evaluates in a finite number of steps. There are, however, several reasons why this might not hold, e.g. the evaluation of the term $\texttt{match}\, x\, \texttt{with |Nil} \to e_1 \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2$ becomes stuck if $\mathcal{H}\big(\mathcal{S}(x)\big) = \mathbf{Bad}$. Another likely reason might be that the evaluation of e simply does not terminate due to a never-ending recursion.

In both cases, it would be highly desirable to know for sure that a program does not exceed its allowed memory resources, especially in the case when things do go wrong. While a big-step (or natural) operational semantics cannot distinguish between non-termination and becoming stuck, we will use this to our advantage now and devise a big-step semantics which exactly captures only those evaluations that do try to allocate more memory than they are allowed, and which would be otherwise legitimate. This is possible since all programs that exceed the available free memory, even non-terminating ones, must do so already after a *finite* number of steps. This basic approach to reason about non-termination with a big-step semantics is not new; similar techniques have been used before [ABH+07, Cam08].

It is fairly easy to see that the extension of the operational semantics in order to capture memory abuse, as given below, is properly defined to complement the cost operational semantics given in Definition 5.3 on page 82.

Definition 5.11 (Out of Memory Semantics)**.**

$$\frac{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} e_1 \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{let}\ \ x\ \texttt{=}\ e_1\ \ \texttt{in}\ \ e_2 \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \textsc{Let I})$$

$$\frac{\mathcal{S}, \mathcal{H} \overset{m}{\underset{m_0}{\vdash}} e_1 \downarrow v_0, \mathcal{H}_0 \qquad \mathcal{S}[x \mapsto v_0], \mathcal{H}_0 \overset{m_0}{\vdash} e_2 \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{let}\ \ x\ \texttt{=}\ e_1\ \ \texttt{in}\ \ e_2 \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \textsc{Let II})$$

$$\frac{\begin{array}{c}\mathcal{S}(x_i) = v_i \quad (\text{for } i = 1, \ldots, k) \\ \mathcal{P}(f) = (y_1, \ldots, y_k \to e_f) \qquad [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k], \mathcal{H} \overset{m}{\vdash} e_f \downarrow \mathsf{Out\ of\ Memory}\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} f(x_1, \ldots, x_k) \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \textsc{Fun})$$

$$\frac{\mathcal{S}(x) = \mathbf{tt} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash} e_t \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \textsc{If-T})$$

$$\frac{\mathcal{S}(x) = \mathbf{ff} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash} e_f \downarrow \text{Out of Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \textsc{If-F})$$

$$\frac{\mathcal{S}(x) = (v_1, v_2) \qquad \mathcal{S}[x_1 \mapsto v_1, x_2 \mapsto v_2], \mathcal{H} \overset{m}{\vdash} e \downarrow \text{Out of Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match}\, x\, \texttt{with}\ (x_1\texttt{,}\ x_2) \to e \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \textsc{E-Pair})$$

$$\frac{w = \big(\mathbf{tt}, \mathcal{S}(x)\big) \qquad m < \textsc{Size}(w)}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{Inl(}x\texttt{)} \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \textsc{Inl})$$

$$\frac{w = \big(\mathbf{ff}, \mathcal{S}(x)\big) \qquad m < \textsc{Size}(w)}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{Inr(}x\texttt{)} \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \textsc{Inr})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H} \overset{m}{\vdash} e_l \downarrow \text{Out of Memory}\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l\, \texttt{|Inr(}y\texttt{)} \to e_r \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \text{E-}\textsc{Inl})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(w)}{\vdash} e_l \downarrow \text{Out of Memory}\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match!}\, x\, \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l\, \texttt{|Inr(}y\texttt{)} \to e_r \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \text{E!-}\textsc{Inl})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H} \overset{m}{\vdash} e_r \downarrow \text{Out of Memory}\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match}\ x\ \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l\, \texttt{|Inr(}y\texttt{)} \to e_r \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \text{E-}\textsc{Inr})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[y \mapsto w'], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(w)}{\vdash} e_r \downarrow \text{Out of Memory}\end{array}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match!}\, x\, \texttt{with}\ \texttt{|Inl(}y\texttt{)} \to e_l\, \texttt{|Inr(}y\texttt{)} \to e_r \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \text{E!-}\textsc{Inr})$$

$$\frac{v = \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big) \qquad m < \textsc{Size}(v)}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{Cons(}x_h\texttt{,}x_t\texttt{)} \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \textsc{Cons})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash} e_1 \downarrow \text{Out of Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1\, \texttt{|Cons(}x_h\texttt{,}x_t\texttt{)} \to e_2 \downarrow \text{Out of Memory}} \quad (\text{LF} \not\downarrow \text{E-}\textsc{Nil})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \overset{m}{\vdash} e_1 \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \mathtt{match!}\, x\ \mathtt{with}\ \mathtt{|Nil} \to e_1 \mathtt{|Cons}(x_h, x_t) \to e_2 \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \text{E!-Nil})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = v \qquad v = (v_h, v_t) \qquad \mathcal{S}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H} \overset{m}{\vdash} e_2 \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \mathtt{match}\ x\ \mathtt{with}\ \mathtt{|Nil} \to e_1 \mathtt{|Cons}(x_h, x_t) \to e_2 \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \text{E-Cons})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = v \qquad v = (v_h, v_t) \qquad \mathcal{S}[x_h \mapsto v_h, x_t \mapsto v_t], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \overset{m + \textsc{Size}(v)}{\vdash} e_2 \downarrow \mathsf{Out\ of\ Memory}}{\mathcal{S}, \mathcal{H} \overset{m}{\vdash} \mathtt{match!}\, x\ \mathtt{with}\ \mathtt{|Nil} \to e_1 \mathtt{|Cons}(x_h, x_t) \to e_2 \downarrow \mathsf{Out\ of\ Memory}} \quad (\mathrm{LF} \not\downarrow \text{E!-Cons})$$

We can now prove that each finite evaluation which exceeds the available free memory must have been started with a smaller amount of free memory than predicted by our analysis. Consequently, *no* well-typed program will exceed the predicted memory resources within any finite number of steps.

Theorem 2 (Soundness for Non-Terminating LF)**.** *Fix a well-typed* $\mathrm{LF}_\diamond$ *program* $\mathcal{P}$ *with signature* Σ*. If*

$$\Gamma \overset{p}{\underset{p'}{\vdash}} e{:}C \tag{2.A}$$

$$\mathcal{S}, \mathcal{H} \overset{m}{\vdash} e \downarrow \mathsf{Out\ of\ Memory} \tag{2.B}$$

$$\mathcal{H} \vDash \mathcal{S} : \Gamma \tag{2.C}$$

then $m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma)$ *holds.*

Proof. The proof is by induction on the lengths of the derivations of (2.A). and (2.B) ordered lexicographically with the derivation of Out of Memory taking priority over the typing derivation.

The most interesting case is probably the case for rule $\mathrm{LF} \not\downarrow$ Let II, which requires the application of Theorem 1.

LF $\not\downarrow$ Let I: The induction hypothesis directly yields $m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1)$ and thus by the non-negativity of the potential also $m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1, \Gamma_2)$ as claimed in the theorem.

LF $\not\downarrow$ Let II: First, suppose that $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1)$ holds, for otherwise we argue as in the previous case that we obtain the desired result immediately. Therefore we may choose $r = m - \big(p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1)\big) \in \mathbb{Q}^+$ with $m \geq p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1) + r$ (in fact equality holds).

By premise (2.B), we have $\mathcal{S}, \mathcal{H} \vdash e_1 \downarrow v_0, \mathcal{H}_0$ and hence we apply Theorem 1 to obtain $\mathcal{S}, \mathcal{H} \vdash^{m}_{m_0} e_1 \downarrow v_0, \mathcal{H}_0$ with

$$p_0 + \Phi_{\mathcal{H}_0}(v_0 : A) + r \leq m_0$$

From the induction hypothesis for e_2 we then know that

$$m_0 < p_0 + \Phi_{\mathcal{H}_0}(\mathcal{S}[x \mapsto v_0] : \{\Gamma_2, x{:}A\})$$

and therefore, by combining our knowledge about m_0 from above, we have

$$\begin{aligned} p_0 + \Phi_{\mathcal{H}_0}(v_0 : A) + r &< p_0 + \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) + \Phi_{\mathcal{H}_0}(v_0 : A) \\ r &< \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) \\ m - \big(p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1)\big) &< \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) \\ m &< p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1) + \Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) \end{aligned}$$

By Lemma 5.9 we obtain $\Phi_{\mathcal{H}_0}(\mathcal{S} : \Gamma_2) \leq \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_2)$ and thence

$$\begin{aligned} m &< p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1) + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_2) \\ m &< p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma_1, \Gamma_2) \end{aligned}$$

as required.

LF $\not\downarrow$ Fun: Applying the induction hypothesis yields

$$m < p + \Phi_{\mathcal{H}}([y_1 \mapsto v_1, \ldots, y_k \mapsto v_k] : \{y_1{:}A_1, \ldots, y_k{:}A_k\})$$

and thus also, as required, we have

$$m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \{x_1{:}A_1, \ldots, x_k{:}A_k\})$$

since $\mathcal{S}(x_i) = v_i$ and $\mathcal{P}$ was assumed to be a well-typed $\mathrm{LF}_\diamond$ program (Definition 4.5).

$\mathrm{LF} \not\vdash$ If-T & $\mathrm{LF} \not\vdash$ If-F: Since the potential for boolean types is always zero, we have $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}\mathsf{bool}) = \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}\mathsf{bool})$ in either case. The type rule $\mathrm{LF} \vdash_\diamond$If furthermore leaves the petty potential p and p' unchanged, hence the claim follows trivially from the application of the induction hypothesis.

$\mathrm{LF} \not\vdash$ E-Pair: The case for pair elimination follows similarly trivial from the induction hypothesis as the previous case. By the definition of potential, Definition 5.7, we have $\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x{:}A{\times}B) = \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma, x_1{:}A, x_2{:}B)$, while the petty potential p and p' remains unchanged by type rule $\mathrm{LF} \vdash_\diamond$E-Pair.

$\mathrm{LF} \not\vdash$ Inl & $\mathrm{LF} \not\vdash$ Inr: According to rule $\mathrm{LF} \not\vdash$ Inr we have $m < \textsc{Size}\,(\mathbf{ff}, \mathcal{S}(x))$ and $\textsc{Size}\,(\mathbf{ff}, \mathcal{S}(x)) = \textsc{TypSize}\big(\mathsf{bool}{\times}B\big)$ by Lemma 4.16. According to type rule $\mathrm{LF} \vdash_\diamond$Inr we also have $p = q_r + \textsc{TypSize}\big(\mathsf{bool}{\times}B\big)$. Since both q_r and $\Phi_{\mathcal{H}}(\mathcal{S}(x) : B)$ are non-negative, we derive $m < p + \Phi_{\mathcal{H}}(\mathcal{S}(x) : B)$ as desired. The case for $\mathrm{LF} \not\vdash$ Inl follows by symmetry.

$\mathrm{LF} \not\vdash$ E-Inl & $\mathrm{LF} \not\vdash$ E-Inr: Assume without loss of generality that $\mathcal{S}(x) = \ell$ and $\mathcal{H}(\ell) = w$ and $w = (\mathbf{ff}, w')$ holds according to rule $\mathrm{LF} \not\vdash$ E-Inr.

By the definition of potential (Definition 5.7 on page 87) we have $p{+}\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma){+}\Phi_{\mathcal{H}}(\mathcal{S}(x) : (A, q_l) + (B, q_r)) = p + q_l + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi_{\mathcal{H}}(\mathcal{S}(x) : A)$, which fits the required premise of type rule $\mathrm{LF} \vdash_\diamond$E-Sum. Thus we obtain the required result directly from the application of the induction hypothesis.

$\mathrm{LF} \not\vdash$ E!-Inl & $\mathrm{LF} \not\vdash$ E!-Inr: Assume again without loss of generality that $\mathcal{S}(x) = \ell$ and $\mathcal{H}(\ell) = w$ and $w = (\mathbf{tt}, w')$ holds according to rule $\mathrm{LF} \not\vdash$ E!-Inl.

By the definition of potential (Definition 5.7 on page 87) we have $p{+}\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma){+}\Phi_{\mathcal{H}}(\mathcal{S}(x) : A{+}q_l B q_r) = p + q_l + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi_{\mathcal{H}}(\mathcal{S}(x) : A)$. Lemma 6.17 then allows us to deduce $\textsc{Size}\,(w) = \textsc{TypSize}\big(\mathsf{bool}{\times}A\big)$, so we can again derive the required result directly from the induction hypothesis.

$\mathrm{LF} \not\vdash$ Cons: Let $s = \textsc{TypSize}\big(A{\times}\mathsf{list}(A, q)\big)$. From the premise (2.C) we have $\mathcal{H} \vDash S(x_h){:}A$ and $\mathcal{H} \vDash S(x_t){:}\mathsf{list}(A, q)$ and hence by Lemma 4.16 we also have $s = \textsc{Size}\big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big)$ and therefore by (2.B) necessarily that $m < s$.

By premise (2.A) we have $p = q + s$ and thus the desired result follows by the non-negativity of both q and $\Phi_{\mathcal{H}}(\mathcal{S} : \{x_h{:}A, x_t{:}\mathsf{list}(A, q)\})$.

LF $\not\downarrow$ E-Nil & LF $\not\downarrow$ E!-Nil: We treat both cases simultaneously, since the only difference in the rules is that `match` is replaced by `match!`, which has no other effect in the case of Nil.

Applying the induction hypothesis yields $m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \Gamma)$. By premise (2.B) we know that $\mathcal{S}(x) = \mathbf{Null}$, and thus by the definition of potential again we have $\Phi_{\mathcal{H}}(\mathcal{S} : \{x{:}\mathsf{list}(A, q)\}) = 0$ and therefore we deduce that the required result $m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A, q)\})$ holds.

LF $\not\downarrow$ E-Cons: The proof is an almost identical subset of the steps taken to prove the case of LF $\not\downarrow$ E!-Cons below. From (2.C) we have $\mathcal{H} \vDash \mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\}$ by unfolding the definition of $\vDash$ once. Furthermore by Lemma 4.12 we have $\mathcal{H} \vDash \mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\}$. Thus we apply the induction hypothesis and obtain

$$m < p + q + \Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\})$$

also $\Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{x_h{:}A, x_t{:}\mathsf{list}(A, q)\}) = q + \Phi_{\mathcal{H}}(\mathcal{S} : \{x{:}\mathsf{list}(A, q)\})$ by Definition 5.7 since $\mathcal{H}\big(\mathcal{S}(x)\big) = (v_h, v_t)$ and hence

$$m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A, q)\})$$

as required.

LF $\not\downarrow$ E!-Cons: Let $s = \textsc{TypSize}\big(A{\times}\mathsf{list}(A, q)\big)$. By premise (2.C) and Lemma 4.16, we also have $s = \textsc{Size}\big(\mathcal{S}(x)\big)$.

Again from (2.C) we have $\mathcal{H} \vDash \mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\}$ by unfolding the definition of $\vDash$ once. Furthermore by Lemma 4.12 we have $\mathcal{H}[\ell{\mapsto}\mathbf{Bad}] \vDash \mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\}$. Thus we apply the induction hypothesis and obtain

$$m + s < p + q + s + \Phi_{\mathcal{H}[\ell\mapsto\mathbf{Bad}]}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\})$$
$$m < p + q + \Phi_{\mathcal{H}[\ell\mapsto\mathbf{Bad}]}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\})$$

thus by Lemma 5.8

$$m < p + q + \Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{\Gamma, x_h{:}A, x_t{:}\mathsf{list}(A, q)\})$$

also $\Phi_{\mathcal{H}}(\mathcal{S}[x_h{\mapsto}v_h, x_t{\mapsto}v_t] : \{x_h{:}A, x_t{:}\mathsf{list}(A, q)\}) = q + \Phi_{\mathcal{H}}(\mathcal{S} : \{x{:}\mathsf{list}(A, q)\})$ by Definition 5.7 since $\mathcal{H}\big(\mathcal{S}(x)\big) = (v_h, v_t)$ and hence

$$m < p + \Phi_{\mathcal{H}}(\mathcal{S} : \{\Gamma, x{:}\mathsf{list}(A, q)\})$$

as required.

□

5.5 Introductory Program Examples Analysed

We will now revisit the program examples from Section 4.4 and examine them with our automated amortised analysis technique and discuss the obtained result. We will focus on the meaning of the obtained results. Since the examples are sufficiently small, it should be easy for the reader to independently verify the findings of the analysis and recognise them as correct.

We show how the analysis is actually performed by examining a fifth example, an evaluator for arithmetic expressions, which also demonstrates mutual recursion. For this last example, we will perform each single step of the analysis technique in great detail, showing how the linear program is generated and what it actually looks like. That example also includes user-defined recursive data types as discussed in Section 5.2.1. We will discuss the effects of different choices for the memory model and their outcome, as well as seeing the effect of switching read-only matches to destructive ones.

5.5.1 List Tails Analysed

Recall that the function `tails`, whose program code and type was previously shown in Figure 4.4 on page 65, computes a list containing all the tail-lists of a given input list. For example, the call `tails(`$[a, b, c, d, e]$`)` with an argument of length five, evaluates to the result list $\big[[a, b, c, d, e], [b, c, d, e], [c, d, e], [d, e], [e], [\,]\big]$.

It is important to note that the output of `tails` is aliased. While a consumer of the output above would see 15 inner cons nodes and 6 outer cons nodes, only 11 out of these 21 nodes exist in the memory. The function `tails` only allocates the 6 outer cons nodes during its computation.

Applying the amortised analysis yields the following $\mathrm{LF}_\diamond$ type for `tails`:

$$\texttt{tails} : \mathsf{list}(A, 2) \xrightarrow{2 \triangleright 0} \mathsf{list}(\mathsf{list}(A, 0), 0)$$

so according to the Definition 6.20 of potential, a call requires $2n + 2$ heap cell units, where n is the length of the input list. Since a cons node occupies two heap units for storage, we see that the worst-case bound is exact for the computation example that we considered. There is at least one computation that meets the predicted bound.

However, we can also inspect the generated LP to learn more about the precision of the inferred cost bound. The inferred cost formula only over-approximates if at least one of the constraints becomes a strict inequality under the chosen solution. In this case, the solution achieves equality for all constraints, hence we know that the cost formula is always exact by the observations made in Section 5.1.1. The function `tails` will always allocate $2n+2$ for an input of length n, no more, no less, regardless of what that input actually is.

For the generalised type $\texttt{tails} : \mathsf{list}(A, a) \xrightarrow{x \triangleright y} \mathsf{list}(\mathsf{list}(A, b), c)$ we obtain the following generated linear program, after some unimportant simplifications to improve readability:

$$x \geq \textsc{TypSize}\big(A{\times}\mathsf{list}(A)\big) + c + y \qquad a \geq \textsc{TypSize}\big(A{\times}\mathsf{list}(A)\big) + c \qquad b = 0$$

So we see that the outer cons nodes of the result may carry potential, while the inner cons nodes can never carry any potential. This is as expected, since the assignment of potential ignores aliasing, hence any potential held by the inner cons nodes of the output would be non-linear in the input.

Note that our implementation does *not* simplify the generated LP before handing it to the LP-solver. We found that a transformation of the generated LP often introduced numeric instabilities, which then significantly increased the time to solve the LP. Since the generated LPs are sparse and also quite small, efficiency is not a concern. For all the program examples in this chapter, the generated LP could be solved almost instantaneously by our solver LP-solve [BEN] on our contemporary laptop (2.53GHz Intel Core 2 Duo, with 6MB cache and 4GB memory). It is also likely that the result from our earlier work [HJ03], that any solvable generated LP also admits an (usually non-optimal) integral solution, applies to $\mathrm{LF}_\diamond$ as well.

5.5.2 List Reversal Analysed

Recall that function `reverse` and its auxiliary recursive helper `rev_acc`, whose code was shown in Figure 4.5 on page 67, reverses a list: $\texttt{reverse}([a, b, c, d]) = \texttt{rev\_acc}([b, a], [c, d]) = [d, c, b, a]$.

We claimed that the reversal was in-place, i.e. by stepwise deallocation of the input list and reassembly of the reversed list in the space that became free through the deallocations, no additional heap memory is required for the computation. A call to function `reverse` should not allocate any new memory at all.

Our analysis infers the types

$$\texttt{reverse} = \mathsf{list}(A,0) \xrightarrow{0 \triangleright 0} \mathsf{list}(A,0)$$
$$\texttt{rev\_acc} = (\mathsf{list}(A,0), \mathsf{list}(A,0)) \xrightarrow{0 \triangleright 0} \mathsf{list}(A,0)$$

which confirms that neither function will allocate any heap space.

A worst-case bound of 0 is necessarily an exact bound. However, it is not necessarily an exact cost formula, since a program might deallocate more memory than it allocates. However, again all inequalities become equalities under the inferred solution.

This distinction can be neatly illustrated if we ask about the correctness of the program. One desired property of list-reversal is, that the length of the result is the same as the length of the input. The worst-case bound of 0 only tells us that the result is certainly not bigger than the input. The exact cost formula tells us that the length of output and input are the same, provided that there is no application of $\mathrm{LF} \vdash \textsc{Weak}$ in the unannotated LF type derivation, which in turn excludes the possibility that allocated memory is reachable from the output.

We remark that the amortised analysis can also successfully produce a bound for the non-destructive list-reversal. In this case the following types are inferred

$$\texttt{reverse} = \mathsf{list}(A,x) \xrightarrow{0 \triangleright 0} \mathsf{list}(A,0)$$
$$\texttt{rev\_acc} = (\mathsf{list}(A,x), \mathsf{list}(A,0)) \xrightarrow{0 \triangleright 0} \mathsf{list}(A,0)$$

where x is the size of a cons node of type A. Thus, the maximum heap space cost of executing non-destructive list-reversal is identical to the size of the spine of input list to be reversed, which is clearly an exact bound.

5.5.3 List Zipping Analysed

We now analyse the zipping of two lists, with the code as shown in Figure 4.6 on page 69. We recall the example $\texttt{zip}([a,b,c,d],[e,f,g]) = [(a,e),(b,f),(c,g)]$, and that the length of the output is equal to the minimum of the lengths of the two input lists.

For the general annotated type

$$\texttt{zip} : \mathsf{list}(A,a), \mathsf{list}(B,b) \xrightarrow{x \triangleright y} \mathsf{list}(A \times B, c)$$

we automatically infer the linear program

$$x \geq y \qquad x + a \geq y \qquad a + b \geq \textsc{TypSize}\big((A{\times}B) \times \mathsf{list}(A{\times}B)\big) + c$$

For readability, we set A and B to the boolean type bool, so that the last inequality becomes $a + b \geq 3 + c$. We also ignore solutions that have potential assigned to the result, which are uninteresting for the purpose of this example, and therefore add the constraints $c = 0$ and $y = 0$. The resulting linear program still admits several solutions:

$$\begin{aligned}
&\texttt{zip} : \mathsf{list}(A, 3), \mathsf{list}(B, 0) \xrightarrow{0 \triangleright 0} \mathsf{list}(A{\times}B, 0)\\
&\texttt{zip} : \mathsf{list}(A, 0), \mathsf{list}(B, 3) \xrightarrow{0 \triangleright 0} \mathsf{list}(A{\times}B, 0)\\
&\texttt{zip} : \mathsf{list}(A, 1.5), \mathsf{list}(B, 1.5) \xrightarrow{0 \triangleright 0} \mathsf{list}(A{\times}B, 0)
\end{aligned}$$

Let n denote the list length of the first argument and m the length of the second, then we can derive the following cost bounds from these types, respectively:

$$3n \qquad\qquad 3m \qquad\qquad 1.5(n + m)$$

Neither of these upper-bounds is minimal, since the minimal upper-bound is clearly $3{\cdot}\min(n, m)$ as observed earlier in Section 4.4.3. However, the three delivered bounds are the best we could have hoped for our technique to produce, since we sacrificed the ability to infer non-linear bounds in order to achieve automation.

We see that the result of our amortised analysis is really the generated linear program, that describes several bounds on the memory consumption of the analysed program. However, multi-dimensional linear programs are not very easy to understand for humans. The implementation of our analysis thus employs a simple heuristic that simply picks one $\mathrm{LF}_\diamond$ type for a program. The user can ask for another solution, if the presented solution is unsatisfactory. More specifically, if the user has external knowledge, for example knowing that the second argument of `zip` will always be the smaller one, the user can ask for an attempt to specifically increase or decrease a certain annotation.

The heuristic employed by our implementation prefers small values on the left-hand side of arrows and, to a lesser extend, larger values on the right-hand side of an arrow. The nesting depth of an annotation has also an effect on the importance to minimise or maximise it. Overall, solutions with fewer strict inequalities are preferred as well, which is achieved by turning all inequalities to equalities with a fresh slack variables, that is penalised in the objective function generated by the heuristic.

It is furthermore interesting to note that the solution leading to the second $\mathrm{LF}_\diamond$ type for `zip` shown above achieves equality for all constraints in the generated linear program. However, we do not obtain an exact cost formula from this solution, since the type derivation for the program requires an application of type rule $\mathrm{LF} \vdash$WEAK in its second computational branch, the one leading to the second inequality.

5.5.4 In-Place Insertion Sort Analysed

The in-place insertion sort algorithm, as implemented in Figure 4.7 on page 69, has been subjected to the amortised analysis technique before [Hof00, HJ03, JLHH10], and we mainly include it here for completeness.

For the generalised annotated types

$$\begin{aligned} \texttt{ins\_sort} &= \mathsf{list}(A,a) \xrightarrow{x \triangleright u} \mathsf{list}(A,b) \\ \texttt{insert} &= A, \mathsf{list}(A,c) \xrightarrow{y \triangleright v} \mathsf{list}(A,d) \\ \texttt{leq} &= (A,A) \xrightarrow{z \triangleright w} \mathsf{bool} \end{aligned}$$

we obtain the linear program shown in Figure 5.4, which includes a constant $\mathscr{C}_{\leq \mathsf{A}}$ to denote the heap space costs incurred by comparing two values of type A. Assuming that this comparison has no heap space costs, i.e. $\mathscr{C}_{\leq \mathsf{A}} = 0$, the LP admits a solution yielding the following $\mathrm{LF}_\diamond$ types

$$\begin{aligned} \texttt{ins\_sort} &= \mathsf{list}(A,0) \xrightarrow{0 \triangleright 0} \mathsf{list}(A,0) \\ \texttt{insert} &= A, \mathsf{list}(A,0) \xrightarrow{2 \triangleright 0} \mathsf{list}(A,0) \\ \texttt{leq} &= (A,A) \xrightarrow{0 \triangleright 0} \mathsf{bool} \end{aligned}$$

We learn that function `ins_sort` has indeed a worst-case heap usage bound of 0. Hence it must sort the input list in-place. The annotation 2 for `insert` shows that the function may allocate an additional list node. In fact, it always does so, by inserting the first argument into the list received as its second argument. However, function `ins_sort` is able to pay this fixed cost, since it always deallocates a list node before calling `insert`.

We remark that the LP becomes infeasible for $\mathscr{C}_{\leq \mathsf{A}}$ being larger than zero, since the number of calls to `leq`, i.e. the number of comparisons, required to sort a list, is non-linear in its length. Note that the function `insert` remains still analysable on its own it this case, since inserting an element into a sorted list of length n requires at most n comparisons.

$$x \geq u \tag{5.1}$$
$$x + a \geq x \tag{5.2}$$
$$x + a \geq x - u + y \tag{5.3}$$
$$x + a \geq x - u + y - v \tag{5.4}$$
$$b \geq c \tag{5.5}$$
$$d \geq b \tag{5.6}$$
$$y \geq \textsc{TypSize}\big(A{\times}\mathsf{list}(A)\big) + d + v \tag{5.7}$$
$$y + c \geq z \tag{5.8}$$
$$y + c \geq z - w + 2 \cdot \textsc{TypSize}\big(A{\times}\mathsf{list}(A)\big) + 2d + v \tag{5.9}$$
$$y + c \geq z - w + y \tag{5.10}$$
$$y + c \geq z - w + y - v + \textsc{TypSize}\big(A{\times}\mathsf{list}(A)\big) + d + v \tag{5.11}$$
$$z \geq \mathscr{C}_{\leq \mathsf{A}} + w \tag{5.12}$$

Analysing function `ins_sort` generates an LP with 11 inequalities over 10 variables. The constant $\mathscr{C}_{\leq \mathsf{A}}$ stands for the heap cost incurred by the comparison for type A.

Figure 5.4: LP generated for in-place insertion sort

$$
\begin{array}{lcl}
\text{data } \mathsf{expr} & = & \mathsf{Val\ int} \mid \mathsf{Plus\ expr\ expr} \\
\texttt{eval} & : & \mathsf{expr} \to \mathsf{int} \\
\texttt{eval}(\mathsf{Val}\ n) & = & n \\
\texttt{eval}(\mathsf{Plus}\ n\ m) & = & \texttt{eval}\ n + \texttt{eval}\ m
\end{array}
$$

Figure 5.5: Semantics for a simple language of arithmetic expressions

5.6 Analysing an Abstract Machine for Arithmetic Expressions

We will now analyse the program code for an abstract machine that evaluates arithmetic expressions, described by Hutton and Wright in [HW06]. This program example is not only interesting because of employing mutual recursion, but also because it was not specifically designed to study space usage. Hutton and Wright *calculated* the abstract machine from the specification of a simple language of arithmetic expressions, whose semantics is shown in Figure 5.5. This exemplary language is very simple, just consisting of integer values and addition operations. It would be easy to add further operations, but we would not gain much more insight, since all numeric operations would have identical heap space costs anyway.

Hutton and Wright first make the evaluation order explicit by rewriting the semantics in *continuation-passing style* [Rey72]. In the second step, they perform the well-known *defunctionalisation* technique, also described by Reynolds [Rey72]. At last, they refactor the obtained program by a simple renaming of identifiers in order to ease the understanding of the calculated code. The program, transformed into an extended LF syntax, is shown in Figure 5.6.

For brevity, we extended LF by adding two new built-in data types. The type expr essentially represents binary trees with unlabelled nodes and integer labels on its leaves, or in other words, an arithmetic formula over integers and the plus operator. The type cont is a list, whose nodes either carry the aforementioned binary tree expression or a simple integer, representing a partially evaluated expression. We must also fix the allocation costs for each constructor, in order to complete our

$$\mathrm{data}\ \mathsf{expr} = \mathsf{Val}(\mathsf{int}) \mid \mathsf{Plus}(\mathsf{expr}, \mathsf{expr})$$
$$\mathrm{data}\ \mathsf{cont} = \mathsf{Stop} \mid \mathsf{Eval}(\mathsf{expr}, \mathsf{cont}) \mid \mathsf{Add}(\mathsf{int}, \mathsf{cont})$$

$$\begin{array}{lcl}
\Sigma(\mathtt{eval}) & := & \big(\mathsf{expr}, \mathsf{cont}\big) \to \mathsf{int} \\
\mathcal{P}(\mathtt{eval}) & := & e, c \quad \to \quad \mathtt{match}\ e\ \mathtt{with} \\
 & & \qquad \mid \mathsf{Val}(n) \to \mathtt{exec}(c, n) \\
 & & \qquad \mid \mathsf{Plus}(n, m) \to \\
 & & \qquad\quad \mathtt{let}\ d = \mathsf{Eval}(m, c)\ \mathtt{in}\ \mathtt{eval}(n, d) \\
\Sigma(\mathtt{exec}) & := & \big(\mathsf{cont}, \mathsf{int}\big) \to \mathsf{int} \\
\mathcal{P}(\mathtt{exec}) & := & c, n \quad \to \quad \mathtt{match}\ c\ \mathtt{with} \\
 & & \qquad \mid \mathsf{Stop} \to n \\
 & & \qquad \mid \mathsf{Eval}(e, d) \to \\
 & & \qquad\quad \mathtt{let}\ f = \mathsf{Add}(n, d)\ \mathtt{in}\ \mathtt{eval}(e, f) \\
 & & \qquad \mid \mathsf{Add}(m, d) \to \\
 & & \qquad\quad \mathtt{let}\ s = n + m\ \mathtt{in}\ \mathtt{exec}(d, s) \\
\Sigma(\mathtt{run}) & := & \mathsf{expr} \to \mathsf{int} \\
\mathcal{P}(\mathtt{run}) & := & e \quad \to \quad \mathtt{let}\ s = \mathsf{Stop}\ \mathtt{in}\ \mathtt{eval}(e, s)
\end{array}$$

Figure 5.6: Abstract machine for evaluating arithmetic expressions

memory model. Since we want to keep some flexibility, we say that the constructors $\mathsf{Val}, \mathsf{Plus}, \mathsf{Stop}, \mathsf{Eval}, \mathsf{Add}$ require the constant amount of $\mathscr{C}_{\mathsf{Val}}, \mathscr{C}_{\mathsf{Plus}}, \mathscr{C}_{\mathsf{Stop}}, \mathscr{C}_{\mathsf{Eval}}, \mathscr{C}_{\mathsf{Add}}$ heap cells to store, respectively, including its arguments (in this case either an integer or a pointer).

We exemplary perform the analysis by hand for this example, following the steps as outlined in Section 2.3. However, performing the analysis exactly as described manually is very tedious and we therefore compact the steps as much as possible, without losing anything essential. We also omit writing down the full $\mathrm{LF}_\diamond$ type derivation tree, but only record the side-conditions arising from it.

First, we verify that the program code is indeed well-typed. This is a standard step, giving us a tree-like derivation for the typing judgement of each of the three first-order functions. Our analysis then follows the structures of these derivations, looking at each rule application precisely once to identify the side-condition. We omit stating the full derivations here, but still follow the overall structure.

Next, we enrich the LF type signature to $\mathrm{LF}_\diamond$ by assigning meta-variables to the type. Before we can do this, we must decide upon the potential assigned to the two new types. Following the discussion in Section 5.2.1, each constructor is assigned its own annotation, denoting the potential bound on construction and released again on matching that particular constructor, regardless of whether the matching is destructive or read-only. Doing this adds some considerable overhead to writing down the type, since we need to write out each constructor and its arguments including their annotations.

For example, writing the simple $\mathrm{LF}_\diamond$ type for boolean lists, $\mathsf{list}(\mathsf{bool}, q)$, in this generic form would result in $\mathsf{list}\{\texttt{Nil} \mid \texttt{Cons(bool,\#,}q\texttt{)}\}$ where the symbol # stand for a self-reference, i.e. the innermost enclosing type with the exactly the same annotations, in this case the whole list type itself. So we see that lists have two constructors: the `Nil`-constructor, having no arguments and no annotation[A]; and the `Cons(, )`-constructor, having two arguments, one for the boolean value stored in the list node of type bool and one for the pointer to the tail of the list. Lists are a recursive data type, hence the tail of the list has the same type with the same annotation as the overall type, of course.

[A] Recall from Section 5.2.1 that this is an optional optimisation. We could award `Nil`-constructors their own annotations, but it would always turn out to be zero anyway.

Annotating the types expr and cont accordingly yields the unwieldy types

$$\begin{aligned}&\mathsf{expr}\big\{\mathsf{Val}(\mathsf{int}, q_1) \mid \mathsf{Plus}(\#, \#, q_2)\big\}\\&\mathsf{cont}\big\{\mathsf{Stop}(q_1)\\&\qquad \mid \mathsf{Eval}(\mathsf{expr}\big\{\mathsf{Val}(\mathsf{int}, q_2) \mid \mathsf{Plus}(\#_{\mathsf{expr}}, \#_{\mathsf{expr}}, q_3)\big\}, \#_{\mathsf{cont}}, q_4)\\&\qquad \mid \mathsf{Add}(\mathsf{int}, \#_{\mathsf{cont}}, q_5)\ \big\}\end{aligned}$$

In order to avoid too many line breaks, we abbreviate the notation for these annotated types by reducing the constructors to single letters and hiding base types and self-reference. There is no loss of information implied, this is only a short-hand notation that leaves some less important information implicit.

$$\begin{aligned}&\mathsf{expr}\big\{\mathsf{V}{:}q_1 \mid \mathsf{P}{:}q_2\big\}\\&\mathsf{cont}\big\{\mathsf{S}{:}q_1 \mid \mathsf{E}{:}\mathsf{expr}\{\mathsf{V}{:}q_1 \mid \mathsf{P}{:}q_2\}, q_4 \mid \mathsf{A}{:}q_5\big\}\end{aligned}$$

The extension of the definition of potential is canonical, i.e. each occurring node contributes an amount of potential that is equal to its annotation for its type.

We can now enrich the signature of the program with variables for the yet unknown annotations. We refer to these meta-variables by the name *resource variables*, in order to distinguish them from term variables. Each inserted resource variable has a fresh unique name. We use different letters for the names of the resource variables purely to enhance readability.

$$\begin{aligned}\Sigma(\texttt{eval}) &:= \big(\mathsf{expr}\{\mathsf{V}{:}v_1 \mid \mathsf{P}{:}p_1\}, \mathsf{cont}\big\{\mathsf{S}{:}s_1 \mid \mathsf{E}{:}\mathsf{expr}\{\mathsf{V}{:}v_4 \mid \mathsf{P}{:}p_4\}, r_1 \mid \mathsf{A}{:}a_1\big\}\big) \xrightarrow{x_1 \triangleright y_1} \mathsf{int}\\ \Sigma(\texttt{exec}) &:= \big(\mathsf{cont}\big\{\mathsf{S}{:}s_2 \mid \mathsf{E}{:}\mathsf{expr}\{\mathsf{V}{:}v_2 \mid \mathsf{P}{:}p_2\}, r_2 \mid \mathsf{A}{:}a_2\big\}, \mathsf{int}\big) \xrightarrow{x_2 \triangleright y_2} \mathsf{int}\\ \Sigma(\texttt{run}) &:= \big(\mathsf{expr}\{\mathsf{V}{:}v_3 \mid \mathsf{P}{:}p_3\}\big) \xrightarrow{x_3 \triangleright y_3} \mathsf{int}\end{aligned}$$

From the definition of potential we immediately know that the heap space consumption of `run` is bounded by $x_3 + v_3 \cdot \#\mathsf{V} + p_3 \cdot \#\mathsf{P}$, where $\#\mathsf{V}$ and $\#\mathsf{P}$ respectively denote the number of Val- and Plus-nodes in the input argument of `run`. Note that fixing the shape of the cost bound formula in advance to a canonical form is what allows us to automate the analysis, as explained in Section 2.3. All that is left to do is to determine possible values for the resource variables.

In the next step, we need to gather all the constraints on the resource variables imposed by the typing rules of $\mathrm{LF}_\diamond$. Fortunately, it is easy to do this informally, if we consider each possible branch of computation for each function individually. Please recall the analogy to money from Section 2.3 again. Some potential, that is not associated with any data, is always ready to be spent to pay costs. This is like

petty cash in your pocket, to which we refer to as petty potential. Some potential, i.e. the part that is associated with data, is like money sitting in a bank account, that we first have to withdraw before we can spend it on buying more heap memory.

Analysing function "`run`". We start with the most simple function `run` of Figure 5.6. According to its $\mathrm{LF}_\diamond$ type, we have a petty potential of x_3 available to spend at the start of the evaluation of this function.

According to the $\mathrm{LF}_\diamond$ type rules, the `let` -expression does not cost us anything, but just says that the costs add up, i.e. only the petty potential that is left after allocating the constructor Stop can be used to pay the subsequent function call.

The cost for allocating the constructor Stop is determined by the type rule for that constructor, which we did not define. However, the pattern is identical to the type rule $\mathrm{LF} \vdash \textsc{Cons}$ for list constructors. So the cost for allocating the constructor Stop is just the actual cost plus the potential according to the type of the constructor,

Glancing ahead to the call to function `eval`, we see that the potential for the newly created Stop-constructor must be equal to s_2 in order to fit the required argument type. The call to `eval` also costs a petty potential of x_1.

Collecting these observations, we arrive at the constraint

$$x_3 = (\mathscr{C}_{\mathsf{Stop}} + s_2) \ + \ (x_1) \tag{5.13}$$

Further equality constraints arise from matching resource variables in the types of the arguments with those required by the signature of `eval`.

$$v_3 = v_1 \qquad\qquad p_3 = p_1 \tag{5.14}$$

$$y_3 = y_1 \tag{5.15}$$

The last equation follows since `run` performs no more computations after the call to `eval`, hence it returns all the petty potential that it receives back from `eval`.

Analysing function "`eval`". We start with a petty potential of x_1. The pattern match then increases the petty potential by either v_1 or p_1, depending on the branch taken.

Let us consider the branch for Val first, which is a simple function call, that produces the following equations by matching annotations, while the last one follows by the rule LF⊢Fun as outlined in the analysis of function `rub`.

$$x_1 + v_1 = x_2 \tag{5.16}$$
$$s_1 = s_2 \qquad r_1 = r_2 \qquad a_1 = a_2 \tag{5.17}$$
$$v_4 = v_2 \qquad p_4 = p_2 \tag{5.18}$$
$$y_1 = y_2 \tag{5.19}$$

In the branch for Plus, we have a petty potential of $x_1 + p_1$ available, some of which is used to allocate an Eval-node with potential r_1, while the rest is handed on to a recursive call.

$$x_1 + p_1 = (\mathscr{C}_{\mathsf{Eval}} + r_1) \ + \ (x_1) \tag{5.20}$$
$$v_1 = v_4 \qquad p_1 = p_4 \tag{5.21}$$

The last two equations follow by matching the $\mathrm{LF}_\diamond$ type of m with the annotated type required by the constructor Eval, which in turn is determined by its use as an argument for a call to `eval`.

Note that the recursive call is treated like any other, there is nothing special about it. However, the recursive nature causes the resource variable x_1 to appear on both sides of equation (5.20). This would allows us to simplify the equation to $p_1 = r_1$.

The call also forces us to match the annotations of the $\mathrm{LF}_\diamond$ types of the arguments with the annotations in the type signature for the called function, but due to recursion, these turn all out be useless identities like $v_1 = v_1$, $p_1 = p_1$, etc.

Analysing function "`exec`" The petty potential we start with is x_2. The first branch, which gains a potential of s_2 by the matching, simply yields one equation.

$$x_2 + s_2 = y_2 \tag{5.22}$$

The second branch gains the potential of r_2, but must pay $\mathscr{C}_{\mathsf{Add}} + a_1$ for creating an Add-node with potential a_1 and then x_1 for calling function `eval`.

$$x_2 + r_2 = (\mathscr{C}_{\mathsf{Add}} + a_1) \ + \ (x_1) \tag{5.23}$$

The third branch receives the potential a_2 from the match, and must pay for the heap space cost of adding two integers, which we define as the unknown constant $\mathscr{C}_+$, and a recursive call.

$$x_2 + a_2 = \mathscr{C}_+ + x_2 \tag{5.24}$$

Note that we omitted some redundant equations, that we have already recorded earlier. Those equations arose from identifying the annotations of supplied argument types with the $\mathrm{LF}_\diamond$ type of a function or a constructor.

We see that, with the exception of the x_i, we can ignore the indices of all the other resource variables occurring in this example, since they have all been identified due to the mutual recursive nature of the program. This leaves us with only six essential equations

$$\begin{aligned} x_3 &= \mathscr{C}_{\mathsf{Stop}} + s + x_1 & \qquad x_1 + v &= x_2 \\ p &= \mathscr{C}_{\mathsf{Eval}} + r & \qquad x_2 + s &= y \\ x_2 + r &= \mathscr{C}_{\mathsf{Add}} + a + x_1 & \qquad a &= \mathscr{C}_+ \end{aligned}$$

Given actual values for the four constants in accordance to our desired memory model, we can feed this linear program to any LP-solver, to obtain a solution, if one exists.

We can also simplify the LP further, to determine the general solution

$$x_3 = \mathscr{C}_{\mathsf{Stop}} \qquad p = \mathscr{C}_{\mathsf{Eval}} + \mathscr{C}_{\mathsf{Add}} + \mathscr{C}_+ \qquad r = \mathscr{C}_{\mathsf{Add}} + \mathscr{C}_{\mathsf{Stop}}$$

with the resource variables x_1, x_2, y, v, s, a all being zero. By inserting this solution into the cost formula determined from `run` in the beginning, we learn that a call to `run` costs *exactly* $\mathscr{C}_{\mathsf{Stop}} + \left(\mathscr{C}_{\mathsf{Eval}} + \mathscr{C}_{\mathsf{Add}} + \mathscr{C}_+\right) \cdot \#\mathsf{P}$ where $\#\mathsf{P}$ is the number of Plus-nodes in the input.

A simple way to interpret this result is that each Plus-node in an input argument to `run` causes the allocation of precisely one Eval and one Add-node, as well as executing one plus-operation on two integers; plus a fixed cost for allocating the initial Stop-node.

Note that this cost bound is exact, since we neither used the type rule $\mathrm{LF} \vdash$Weak nor $\mathrm{LF} \vdash_\diamond$Relax. Whether or not rule $\mathrm{LF} \vdash$Weak is needed is determined by the program code, but there is no obvious criterion for deciding whether the use of $\mathrm{LF} \vdash_\diamond$Relax is necessary. In practice, we simply apply the rule $\mathrm{LF} \vdash_\diamond$Relax at

each possible step. The interested reader can see the result by simply replacing $=$ with $\geq$ in each of the constraints above, since we took care to state each equation the right way around. Any solution with a strict inequality then corresponds to a computing branch that requries an over-approximated bound. The inequations also result in fewer resource variables to be identified, essentially allowing some choice in the solution, simply shifting the point in the program when petty potential is discarded.

We conclude this interesting program example with the observation, that the evaluator can also be run entirely in-place, without performing any fresh heap space allocation, if the read-only matches are replaced by destructive ones. This would exchange the equations (5.16), (5.20), (5.22), (5.23) and (5.24) respectively by the following ones

$$\begin{aligned}
x_1 + v_1 + \mathscr{C}_{\mathsf{Val}} &= x_2 \\
x_1 + p_1 + \mathscr{C}_{\mathsf{Plus}} &= (\mathscr{C}_{\mathsf{Eval}} + r_1) \ + \ (x_1) \\
x_2 + s_2 + \mathscr{C}_{\mathsf{Stop}} &= y_2 \\
x_2 + r_2 + \mathscr{C}_{\mathsf{Eval}} &= (\mathscr{C}_{\mathsf{Add}} + a_1) \ + \ (x_1) \\
x_2 + a_2 + \mathscr{C}_{\mathsf{Add}} &= \mathscr{C}_{+} + x_2
\end{aligned}$$

Whether or not in-place evaluation is possible, is then determined by the memory model. For example, any memory model with

$$\mathscr{C}_{\mathsf{Plus}} \geq \mathscr{C}_{\mathsf{Eval}} \qquad \mathscr{C}_{\mathsf{Eval}} \geq \mathscr{C}_{\mathsf{Add}} \qquad \mathscr{C}_{\mathsf{Add}} \geq \mathscr{C}_{+}$$

would allow in-place update; and so also do all memory models satisfying

$$\mathscr{C}_{\mathsf{Plus}} \geq \mathscr{C}_{\mathsf{Eval}} \qquad \mathscr{C}_{\mathsf{Eval}} \geq \mathscr{C}_{\mathsf{Add}} + \mathscr{C}_{+}$$

In order to find all memory models that allow evaluation of arithmetic expressions within a constant amount of heap space, all one has to do is to simplify the entire LP with the added constraints $p = 0$.

6 Higher-Order Analysis Arthur

In this chapter we extend the amortised analysis technique to deal directly with a higher-order language. One possibility to analyse a higher-order program would be to rely on program transformations, such as defunctionalisation [Rey72], to transform a program using higher-order types it into one that only requires first-order types. However, such a method has several drawbacks over a direct analysis of a higher-order program.

Firstly, transformations generally change the properties of the programs that are being analysed, especially time and space consumption. This is unacceptable in any context where the preservation of costs is important, such as resource-aware applications. Moreover, they may also change which programs *can* be costed (e.g. by making linear programs non-linear), and they can destroy the programmers' intuitions about cost, making it more difficult to rectify unwanted resource behaviour. Failure to understand operational cost is a known obstacle to adopting functional languages, so it is important not to change costs in a seemingly arbitrary way.

Secondly, relying on program transformation destroys *compositionality*, requiring a whole-program analysis. This would make it impossible to analyse programs that make use of closed-source libraries which export higher-order definitions. In contrast, the technique proposed in this chapter can analyse a program that uses such a closed source library, provided that the annotated types for all the exported functions of the library are provided. These annotated types, which represent usage-dependent upper-bound functions on costs, can be automatically generated at the compile time of the library and then packaged with its distribution.[A] We would expect that transforming (or just simplifying) the linear programs contained in the annotated types could meet all requirements for obfuscation, despite the (usually desired) transparency of the generated linear programs with respect to the code that caused their generation.

6.1 Defining Arthur

We now define **A** **R**esource-aware **T**ype system for **H**igher-order heap space **U**sage **R**easoning, called ARTHUR.[B] The syntax of ARTHUR is defined in Figure 6.1, with

[A] Of course, in such a scenario one might also want machine-checkable certificates which guarantee that the shipped annotated types indeed agree to the accompanying code. This can be ensured by methods such as those researched in the EU project Mobile Resource Guarantees [MRG05].

[B] ARTHUR was originally conceived as an analysis tool to accompany the CAMELOT compiler generated by the MRG research project [MRG05]. The output of the CAMELOT compiler was a bytecode language called GRAIL.

$e ::=$	`*` | `true` | `false`	Constant
	| x | `let` x `=` e_1 `in` e_2	Variable
	| `rec` y `=` `fun` $x \to e_1$	Abstraction
	| $x_1 x_2$	Application
	| `if` x `then` e_t `else` e_f	Conditional
	| $(e_1$ `&` $e_2)$ | `fst(`x`)` | `snd(`x`)`	Additive Pair
	| $(x_1$`,` $x_2)$ | `match` x `with` $(x_1$`,` $x_2) \to e_2$	Multiplicative Pair
	| `Inl(`x`)` | `Inr(`x`)`	Sum Construction
	| `match` x `with|Inl(`y`)` $\to e_l$ `|Inr(`z`)` $\to e_r$	Sum Access
	| `match!` x `with|Inl(`y`)` $\to e_l$ `|Inr(`z`)` $\to e_r$	Sum Elimination
	| `Nil` | `Cons(`x_1`,`x_2`)`	List Construction
	| `match` x `with|Nil` $\to e_1$ `|Cons(`x_1`,`x_2`)` $\to e_2$	List Access
	| `match!` x `with|Nil` $\to e_1$ `|Cons(`x_1`,`x_2`)` $\to e_2$	List Elimination

The changes between Arthur and LF are highlighted for convenience.

Figure 6.1: Syntax of the language Arthur

the changes as compared to LF being highlighted for convenience. The obsolete function call from LF is replaced by general application. Possibly recursive functions can be defined through general lambda abstraction. The type rules following in Section 6.3 will generally allow any type for the abstracted variable, thereby allowing higher-order functions.

The second novelty of ARTHUR over LF is the introduction of additive (or linear) pairs, discussed in Section 6.1.3. The term former for additive pairs is necessarily not in let-normal form, since it would defy the purpose of an additive pair to suspend the evaluation of its subexpression. This is not a loss, since the reason for let-normal form was the avoidance of endless repetitions for threading the petty potential within our proofs, which is confined by the let-normal form to the `let` -construct only. However, no threading of petty potential is required, since only one of the subexpressions is ever evaluated, similar to the branches of a conditional.

Everything else remains unchanged, so in the following sections we will concentrate on the novel elements only.

Definition 6.1 (Free Variables of ARTHUR Terms)**.** The definition is almost identical to Definition 4.3, hence we only treat the five new term formers here.

$$\begin{aligned}
&\mathrm{FV}(\texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e_1) &&= \mathrm{FV}(e_1) \setminus \{x, y\} \\
&\mathrm{FV}(x_1 x_2) &&= \{x_1, x_2\} \\
&\mathrm{FV}(\texttt{(}e_1\ \texttt{\&}\ e_2\texttt{)}) &&= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
&\mathrm{FV}(\texttt{fst(}x\texttt{)}) &&= \{x\} \\
&\mathrm{FV}(\texttt{snd(}x\texttt{)}) &&= \{x\}
\end{aligned}$$

6.1.1 Recursive Let Definitions

We may often abbreviate $\texttt{let}\ y\ \texttt{=}\ \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e_1\ \texttt{in}\ e_2$ by the short form $\texttt{let rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e_1\ \texttt{in}\ e_2$, as is commonly done. Note that our prototype implementation of ARTHUR furthermore allows mutual recursive functions by terms of the form

$$\texttt{let rec}\ y\ \texttt{=}\ \texttt{fun}\ x_1 \to e_1\ \texttt{and}\ \cdots\ \texttt{and fun}\ x_k \to e_k\ \texttt{in}\ e$$

Mutually recursive functions were already treated in LF and while it would be straightforward to include them in the theoretical work on ARTHUR as well, we refrained from doing so for the sake of simplicity.

Lambda abstraction is bundled with the introduction of recursive definitions, again for the sake of simplicity. Treating general (mutual) recursive let-definitions, which additionally allow the creation of cyclic data structures (see Section 4.3.3), is possible and was demonstrated in [JLHH10]. However, a general recursive let-construct complicates the operational semantics. One must ensure that any possibly created entity is eventually stored at a predicted location, since this location is needed for recursive references. This is difficult, as there might not even be a fresh allocation in all cases. For example, `let rec` x `=` y `in` e just returns the location of a pre-existing non-recursive object, so we cannot simply enforce that the location is mentioned in the result of the defining clause. A solution is usually to employ indirect pointers or to alter locations stored throughout the memory. All cases would require a much stronger invariant for our main soundness result, as the invariant must then deal with partially created data and its update. Bundling recursive definitions with lambda-abstraction side-steps these technical issues, since the recursive closure can always be allocated right away, since all the required information for the closure is readily available at that point. This allows us to keep our focus on the treatment of recursive high-order functions.

6.1.2 Currying, Uncurrying and Partial Application

An Arthur function always takes one single argument, as is standard for lambda abstraction. Of course, the single argument can be of any type, including a product type. One should note that the distinction between the type $(A \times B) \to C$ and $A \to B \to C$ cannot be neglected. A function of the latter type may either perform a computation upon application with an argument of type A, in which case the result must be stored, or it must store the store the argument of type A until the argument of type B is provided. Either case results in the creation of a new closure of $B \to C$, which requires heap space for storage. So currying and uncurrying is not for free in a resource-aware language, and the programmer must make a conscious choice which type is preferred.

Note that instead of lambda abstraction, one may also use *partial application* to the same effect of creating function closures. An amortised analysis for partial application was documented and implemented by the author for the EmBounded research project [EmB09]. The notable difference to lambda abstraction is the added difficulty that the size of a closure generated by partial application is not statically known. Any partially applied function could already be a closure of an arbitrary size, whose size is extended by the newly provided arguments. This problem was circumvented by

using incremental closures (or *closure chains*): Only the newly provided arguments are stored in the closure, plus a pointer to the previous closure. While this saves heap memory at the expense of a small runtime overhead (which can also be dealt with by the amortised analysis for worst-case execution time), it would also make the deallocation of closures more difficult. However, note that neither system allows for the deallocation of closures. Including a primitive for closure deallocation, similar to the one for data types, would be possible, if the sizes of closures were tracked within their type. Similarly, the safety of such deallocations should be guaranteed by a separate analysis method. We will not consider partial application in the remainder of this thesis, and focus on lambda abstraction instead.

6.1.3 Additive Pairs

Another novelty in ARTHUR is the addition of additive pairs, written $(e_1 \,\&\, e_2)$, which allows the explicit suspension of evaluation. Applying either one of the projections `fst` or `snd` causes e_1 or respectively e_2 to be evaluated within its original context. Thus additive pairs store their evaluation context in the heap at their point of creation, in the same manner as function definitions.

One may argue that additive pairs might be simulated through lambda abstraction and sum types, in the sense that an additive pair takes an implicit boolean argument, and returns its result as sum type again. This is not quite equivalent, since the type system cannot guarantee to which branch of the sum type the returned result value belongs to. In contrast, choosing the projection for an additive pair also determines its result type, thereby avoiding unnecessary case distinctions.

The more important difference to function closures is that an additive pair is destroyed by projection, allowing the reuse of the memory that stored the pair and its context. Moreover, the fact that a context of an additive pair is only used once naturally allows the closures of additive pairs to carry potential, while the contexts of a function closure is restricted to zero potential.

If we choose to allow a closure context to carry potential, then we must ensure that there is no unlawful duplication of potential by accessing the context with the same annotated multiple times. This is easily enforced for closures of additive pairs, since their evaluation destroys the closure according to our operational semantics, clearly preventing them to be evaluated twice. For function closures we solved this problem by *choosing* to restrict their potential to zero in all cases. The big benefit of this

restriction is that function closures can then be reused arbitrarily, without any risk of unlawful duplication of potential. The consequence is that functions must gain all their required potential in association with their input, except for a constant amount; This is a relatively minor restriction, which causes the analysis to fail for functions that only recurse over a captured variable alone. We have not yet found an interesting program example where this would be an issue.

We therefore showcase two extreme versions of dealing with closures in an amortised analysis in this thesis. In general, one could deal with use-n-times closures, which can then be shared and applied up to n-times. However, a "sized-type" analysis, such as [VH04], or similar would then be required to bound the number of uses of a closure. Such a bound would then also imply a constant upper bound on the overall potential carried by a closure, provided that all the entities within can be prescribed with a size bound as well. In this case, we do not need to associate a potential with the closure any more, since any constant potential can always be requested upon evaluation. We do not investigate the issue any further in this thesis, since by the above observations, use-n-times closures appear to be of limited use.

6.2 Evaluating Arthur

We now describe the evaluation of ARTHUR terms by giving the operational semantics. Most of the following is quite similar to Sections 4.3 and 4.5, we will mainly focus on the changes that are needed for the ARTHUR extension.

The notions of stack and heap remain unchanged. A *stack* is a finite partial mapping from variables to ARTHUR values; and a *heap* is a finite partial mapping from the set of locations Loc to ARTHUR values.

Definition 6.2 (ARTHUR Values)**.** The set of ARTHUR values Val is defined by the following grammar

$$v \quad ::= \quad () \mid \mathbf{tt} \mid \mathbf{ff} \mid (v_1, v_2) \mid \ell \mid \mathbf{Null} \mid \mathbf{Bad} \mid \langle e\,,\, \mathcal{S} \rangle$$

where e is an ARTHUR term and $\mathcal{S}$ is a stack.

So we still retain values for the unit and boolean constants; pairs; locations; the empty list; the special tag for deallocated locations, whose purpose had been extensively discussed in Section 4.3.1; and a new value for closures. Closures are always pairs,

storing an ARTHUR term together with an evaluation environment, a stack. Since terms for abstraction are commonly found in closures, we introduce a shorthand notation for brevity. Instead of the long $\langle \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e\,,\, \mathcal{S}\rangle$, we shortly write $\langle y{:}x.e\,,\, \mathcal{S}\rangle$.

We fix a concrete memory model again by defining a function $\textsc{Size} : \mathsf{Val} \to \mathbb{N}$. We set the size of a closure to be one plus the size of all values stored in its stack. The rationale being that the ARTHUR term is assumed to be stored in the form of a code pointer, and the size of a location being defined as one. Of course, if a different machine model is desired, this definition may be changed arbitrarily, as long as Definition 6.7 is adjusted accordingly, so that Lemma 6.17 is preserved.

Definition 6.3 (Sizes of ARTHUR Values)**.** We define $\textsc{Size} : \mathsf{Val} \to \mathbb{N}$ by

$$
\begin{array}{lcl}
\textsc{Size}(\texttt{()}) & = & 1 \\
\textsc{Size}(\mathbf{tt}) & = & 1 \\
\textsc{Size}(\mathbf{ff}) & = & 1 \\
\textsc{Size}\big((v_1, v_2)\big) & = & \textsc{Size}(v_1) + \textsc{Size}(v_2) \\
\textsc{Size}(\ell) & = & 1 \\
\textsc{Size}(\mathbf{Null}) & = & 1 \\
\textsc{Size}(\mathbf{Bad}) & = & 0 \\
\textsc{Size}\big(\langle e\,,\, \mathcal{S}\rangle\big) & = & 1 + \displaystyle\sum_{x \in \mathrm{dom}(\mathcal{S})} \textsc{Size}\big(\mathcal{S}(x)\big)
\end{array}
$$

Contrary to Chapter 4, we will only define an instrumented version of the operational semantics, which always measures cost. Thus we have a relation of the form

$$\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \downarrow v, \mathcal{H}'$$

which means that given an initial memory configuration consisting of stack $\mathcal{S}$ and heap $\mathcal{H}$, the ARTHUR expression e can be evaluated to ARTHUR value v and modified heap $\mathcal{H}'$. Moreover, during the entire evaluation, at most m additional heap cell units are occupied, and at least m' heap cell units are unused at the end of the evaluation. The unused heap cells may stem from deallocations within the initial heap $\mathcal{H}$ or from the initially unused m heap cell units. Please recall that the variables m and m' always denote natural numbers, unlike the annotations used in the type systems.

An evaluation statement without the cost annotations is understood in the sense of Lemma 4.20, i.e. the statement $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ is taken to be equivalent to saying that there exist $m, m' \in \mathbb{N}$ such that the judgement $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \downarrow v, \mathcal{H}'$ is derivable by the operational rules of ARTHUR.

Definition 6.4 (ARTHUR Operational Semantics)**.**

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} \ast \downarrow (), \mathcal{H}} \quad (\textsc{Arthur}\downarrow\textsc{Unit})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} \texttt{true} \downarrow \mathbf{tt}, \mathcal{H}} \quad (\textsc{Arthur}\downarrow\textsc{True})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} \texttt{false} \downarrow \mathbf{ff}, \mathcal{H}} \quad (\textsc{Arthur}\downarrow\textsc{False})$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} x \downarrow \mathcal{S}(x), \mathcal{H}} \quad (\textsc{Arthur}\downarrow\textsc{Var})$$

$$\frac{\mathcal{S}, \mathcal{H} \vdash^{m}_{m_0} e_1 \downarrow v_0, \mathcal{H}_0 \qquad \mathcal{S}[x \mapsto v_0], \mathcal{H}_0 \vdash^{m_0}_{m'} e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \downarrow v, \mathcal{H}'} \quad (\textsc{Arthur}\downarrow\textsc{Let})$$

$$\frac{\mathcal{S}^\star = \mathcal{S} \setminus \{x, y\} \cap \mathrm{FV}(e) \qquad w = \langle y{:}x.e\,,\, \mathcal{S}^\star \rangle \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} \texttt{rec}\ y = \texttt{fun}\ x \to e \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\textsc{Arthur}\downarrow\textsc{Rec})$$

$$\frac{\begin{array}{c}\mathcal{S}(y') = \ell \qquad \mathcal{H}(\ell) = \langle y{:}x.e\,,\, \mathcal{S}^\star \rangle \\ \mathcal{S}^\star[y \mapsto \ell, x \mapsto \mathcal{S}(x')], \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} y'\, x' \downarrow v, \mathcal{H}'} \quad (\textsc{Arthur}\downarrow\textsc{App})$$

$$\frac{\mathcal{S}(x) = \mathbf{tt} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_t \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\textsc{Arthur}\downarrow\textsc{If-T})$$

$$\frac{\mathcal{S}(x) = \mathbf{ff} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_f \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f \downarrow v, \mathcal{H}'} \quad (\textsc{Arthur}\downarrow\textsc{If-F})$$

$$\frac{\ell \notin \mathrm{dom}(\mathcal{H}) \qquad w = \left\langle (e_1 \,\&\, e_2)\,,\, \mathcal{S} \restriction \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \right\rangle}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} (e_1, e_2) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad (\textsc{Arthur}\downarrow\textsc{LinPair})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = \left\langle (e_1 \,\&\, e_2)\,,\, \mathcal{S}^\star \right\rangle \qquad \mathcal{S}^\star, \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash^{m + \textsc{Size}(\mathcal{H}(\ell))}_{m'} e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{fst}(x) \downarrow v, \mathcal{H}'} \quad (\textsc{Arthur}\downarrow\textsc{Fst})$$

$$\frac{\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = \big\langle (e_1 \,\&\, e_2)\ ,\ \mathcal{S}^\star \big\rangle \qquad \mathcal{S}^\star, \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash^{m + \textsc{Size}(\mathcal{H}(\ell))}_{m'} e_2 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{snd}(x) \downarrow v, \mathcal{H}'}$$

(ARTHUR↓SND)

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} (x_1\,,\, x_2) \downarrow \big(\mathcal{S}(x_1), \mathcal{S}(x_2)\big), \mathcal{H}} \quad \text{(ARTHUR}{\downarrow}\text{PAIR)}$$

$$\frac{\mathcal{S}(x) = (v_1, v_2) \qquad \mathcal{S}[x_1 {\mapsto} v_1, x_2 {\mapsto} v_2], \mathcal{H} \vdash^{m}_{m'} e \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\, x\, \texttt{with}\ (x_1\,,\, x_2) \to e \downarrow v, \mathcal{H}'} \quad \text{(ARTHUR}{\downarrow}\text{E-PAIR)}$$

$$\frac{w = \big(\mathbf{tt}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} \texttt{Inl}(x) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad \text{(ARTHUR}{\downarrow}\text{INL)}$$

$$\frac{w = \big(\mathbf{ff}, \mathcal{S}(x)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} \texttt{Inr}(x) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \quad \text{(ARTHUR}{\downarrow}\text{INR)}$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y {\mapsto} w'], \mathcal{H} \vdash^{m}_{m'} e_l \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ |\,\texttt{Inl}(y) \to e_l\,|\,\texttt{Inr}(y) \to e_r \downarrow v, \mathcal{H}'} \quad \text{(ARTHUR}{\downarrow}\text{E-INL)}$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{tt}, w'\big) \\ \mathcal{S}[y {\mapsto} w'], \mathcal{H}[\ell {\mapsto} \mathbf{Bad}] \vdash^{m + \textsc{Size}(w)}_{m'} e_l \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\, x\ \texttt{with}\ |\,\texttt{Inl}(y) \to e_l\,|\,\texttt{Inr}(y) \to e_r \downarrow v, \mathcal{H}'} \quad \text{(ARTHUR}{\downarrow}\text{E!-INL)}$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[y {\mapsto} w'], \mathcal{H} \vdash^{m}_{m'} e_r \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ |\,\texttt{Inl}(y) \to e_l\,|\,\texttt{Inr}(y) \to e_r \downarrow v, \mathcal{H}'} \quad \text{(ARTHUR}{\downarrow}\text{E-INR)}$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = \big(\mathbf{ff}, w'\big) \\ \mathcal{S}[y {\mapsto} w'], \mathcal{H}[\ell {\mapsto} \mathbf{Bad}] \vdash^{m + \textsc{Size}(w)}_{m'} e_r \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\, x\ \texttt{with}\ |\,\texttt{Inl}(y) \to e_l\,|\,\texttt{Inr}(y) \to e_r \downarrow v, \mathcal{H}'} \quad \text{(ARTHUR}{\downarrow}\text{E!-INR)}$$

$$\frac{}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m} \texttt{Nil} \downarrow \mathbf{Null}, \mathcal{H}} \quad \text{(ARTHUR}{\downarrow}\text{NIL)}$$

$$\frac{w = \big(\mathcal{S}(x_h), \mathcal{S}(x_t)\big) \qquad \ell \notin \mathrm{dom}(\mathcal{H})}{\mathcal{S}, \mathcal{H} \vdash^{m' + \textsc{Size}(w)}_{m'} \texttt{Cons}(x_h, x_t) \downarrow \ell, \mathcal{H}[\ell \mapsto w]} \ (\textsc{Arthur}{\downarrow}\textsc{Cons})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons}(x_h, x_t) \to e_2 \downarrow v, \mathcal{H}'} \ (\textsc{Arthur}{\downarrow}\textsc{E-Nil})$$

$$\frac{\mathcal{S}(x) = \mathbf{Null} \qquad \mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e_1 \downarrow v, \mathcal{H}'}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\, x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons}(x_h, x_t) \to e_2 \downarrow v, \mathcal{H}'} \ (\textsc{Arthur}{\downarrow}\textsc{E!-Nil})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \\ \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H} \vdash^{m}_{m'} e_2 \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons}(x_h, x_t) \to e_2 \downarrow v, \mathcal{H}'} \ (\textsc{Arthur}{\downarrow}\textsc{E-Cons})$$

$$\frac{\begin{array}{c}\mathcal{S}(x) = \ell \qquad \mathcal{H}(\ell) = w \qquad w = (w_h, w_t) \\ \mathcal{S}[x_h \mapsto w_h, x_t \mapsto w_t], \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash^{m + \textsc{Size}(w)}_{m'} e_2 \downarrow v, \mathcal{H}'\end{array}}{\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{match!}\, x\ \texttt{with}\ \texttt{|Nil} \to e_1 \texttt{|Cons}(x_h, x_t) \to e_2 \downarrow v, \mathcal{H}'} \ (\textsc{Arthur}{\downarrow}\textsc{E!-Cons})$$

The rules for function abstraction and the creation of additive pairs deserve some explanation, since both have the requirement that the stored stack is only allowed to store entities that are actually used in the type. One might usually expect this anyway as a regular optimisation, since there is no need to store a bloated environment. However, for Arthur this is even more important, since the type system has no chance to statically guess the amount of junk that is additionally stored, thereby leading to an under-estimation of heap space required to store the closure.

It is also important to note that the projections for linear pairs deallocate the closure right away. Following our discussion in Section 6.1.3, additive pairs are only allowed to be used once, hence it is always safe to deallocate their closures right away. No extra analysis to ensure the safety of this deallocation is needed, since our type system ensures that additive pair types are treated linearly.

The observations from Section 4.3.1 apply analogously for the operational semantics of Arthur as well, thus Lemma 4.8 can be reformulated:

Lemma 6.5 (Heap Stability)**.** *If* $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ *then for all* $\ell \in \mathrm{dom}(\mathcal{H})$ *we have either* $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ *or* $\mathcal{H}'(\ell) = \mathbf{Bad}$*. Furthermore we also have* $\mathrm{dom}(\mathcal{H}) \subseteq \mathrm{dom}(\mathcal{H}')$*.*

Proof. An inspection of the operational rules for ARTHUR reveals that any existing location can only be altered by overwriting with the special constant **Bad**, whereas newly created values are always bound to fresh locations. □

Note that unlike for LF, it would be useless to define memory consistency at this point. Memory consistency expresses that the values found in memory fit the type associated with them by the typing context and the typing environment (stack).

For example, we do not only expect to find a closure at a location that is associated with a function type, but also that argument and result types fit the defining body of the function. This connection is naturally expressed by a typing derivation for the term stored in the closure, so the definition of memory consistency for ARTHUR must necessarily involve typing judgements.

Given an unannotated version of the type system, like LF for $\mathrm{LF}_\diamond$, we could clearly define memory consistency. However, such a definition would then be useless as an invariant for our soundness theorem. An ARTHUR function type also describes the resource usage of a function. Hence memory consistency for ARTHUR must also express that the term stored in a closure can be evaluated with the limited resource as specified by the associated type. This in turn is expressed by the annotated type system. Therefore, we omit stating an unannotated version of the ARTHUR type system altogether, and define the annotated type system right away.

We remark that for the same reasons we cannot prove a separate preservation lemma similar to Lemma 4.13, since the strengthened memory consistency would require full soundness of the resource consumption for function types. However, such a statement is inherently a part of our main soundness theorem, in which preservation must necessarily be integrated.

6.3 Typing Arthur

The type system for ARTHUR shall be able to track the worst-case heap space costs of evaluating a typed term. Since ARTHUR is a higher-order language, the type

system must be strong enough to express the cost bounds of functions as well. In order to be useful, the type system must also allow function types to be *resource parametric*, meaning that a function that requires 5 heap cells to evaluate can also be used where a function that consumes 7 heap cells was expected. For another example, any function that requires an argument of type $\mathsf{list}(\mathsf{bool}, q)$ should clearly also accept an argument of the type $\mathsf{list}(\mathsf{bool}, r)$, provided that $r \geq q$ holds. Receiving more potential than required, or a reduction in resource demands should never be an obstacle, but at the same time, it should not lead to too much over-approximation either.

In order to satisfy these requirements, the Arthur type system must be able to directly manipulate constraints on the annotations. Unlike for $\mathrm{LF}_\diamond$, whose type system only dealt with rational numbers as annotations, the Arthur type system must therefore deal with so called *resource variables* instead. For example, in the $\mathrm{LF}_\diamond$ type rule for let-expressions `let` x `=` e_1 `in` e_2, the free resources available after evaluating the defining subexpression e_1 is required to be identical to the resource variable denoting the amount of resources that are available for evaluating the body of the let-expression e_2. In the Arthur type system we do not talk about numerical values, and must therefore make such connections explicit. The Arthur type rule for let-expression then uses different resource variables for each premise and then explicitly notes an equation that ensures the equality between the two resource variables.

Let RV be an infinite set of *resource variables*, being disjoint from the set of identifiers used in terms. Resource variables range over $\mathbb{Q}^+$. We denote sets of resource variable by the lower-case Greek letters α, β, γ. A *valuation* $\boldsymbol{v}$ is a map from resource variables to $\mathbb{Q}^+$.

Sets of linear inequalities over rational constants and resource variables are denoted by the lower-case Greek letters ξ, ϕ, ψ. We write $\boldsymbol{v} \vDash \phi$ if valuation $\boldsymbol{v}$ satisfies all constraints in ϕ. and $\psi \vDash \phi$ to denote that ψ entails ϕ, i.e. that all valuations that satisfy ψ also satisfy all constraints in ϕ. We extend valuations to annotated types, mapping the free resource variables to their associated value. It is trivial to see that the relation $\vDash$ between constraint sets is a preorder, i.e. it is reflexive and transitive. It is neither symmetric nor anti-symmetric.

The type grammar of Arthur is shown in Figure 6.2 on the following page, and contains all zero-order $\mathrm{LF}_\diamond$ types as well as two new types for additive pairs and functions.

$$A ::= \mathsf{unit} \mid \mathsf{bool} \mid A \times A \mid (A, q) + (A, r) \mid \mathsf{list}(A, q)$$
$$\mid A^{q \triangleright q'} \& B^{r \triangleright r'} \mid \forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} A$$

with $q, q', r, r' \in \mathsf{RV}$; $\alpha \subset \mathsf{RV}$; and ϕ a set of linear constraints over resource variables.

Figure 6.2: ARTHUR type grammar

The type $A^{q \triangleright q'} \& B^{r \triangleright r'}$ for additive pairs bears four additional resource variables as annotations, two for each part of the additive pair. As we have discussed in Section 6.1.3, an additive pair can be interpreted as a pair of use-once functions which take no arguments each, so the role of each annotation pair is the same as for the first-order type in $\mathrm{LF}_\diamond$. Projecting an additive pair causes the evaluation of the stored expression. The cost of this evaluation must be amortised against the potential associated with the values that were stored in the closure upon the construction of the additive pair, plus a constant amount, the petty potential for that evaluation. This petty potential must be paid when the projection is applied, and its maximal amount is denoted by the first annotation in each pair, q and r. The second annotation in each pair, q' and r', then stands for the least amount of petty potential that is left over after the evaluation of the projection is finished.

The pair of annotations in the function type $\forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} C$ serves the same purpose of noting the amount of petty potential spend and returned on calling a function of that type. However, functions can be used several times in different contexts, and each application might call for a different way of dealing with potential. The function type therefore allows the binding of a set α of resource variables. The linear constraints imposed by the expression stored in the closure are recorded in ψ. Each application of the function will rename all resource variables in α to a fresh name and insert the substituted constraints ψ. So each use of a function can have its own partial solution to the constraints on function body. Note that resource parametricity may thereby cause an increase of the generated LP that is exponential in the depth of the call graph, as noted later in Section 9.1. If both α and ψ are empty sets, i.e. the function type is not resource parametric, we may abbreviate it by simply writing $A \xrightarrow{q \triangleright q'} A$.

We remark that the observations in Section 5.2.1 also apply for ARTHUR. User-definable recursive data types, subsuming lists and trees, can be straightforwardly

$$
\begin{array}{lcl}
\mathrm{FV}_\diamond(\mathsf{unit}) & = & \emptyset \\
\mathrm{FV}_\diamond(\mathsf{bool}) & = & \emptyset \\
\mathrm{FV}_\diamond(A \times B) & = & \mathrm{FV}_\diamond(A) \cup \mathrm{FV}_\diamond(B) \\
\mathrm{FV}_\diamond\big((A,q) + (B,r)\big) & = & \{q,r\} \cup \mathrm{FV}_\diamond(A) \cup \mathrm{FV}_\diamond(B) \\
\mathrm{FV}_\diamond\big(\mathsf{list}(A,q)\big) & = & \{q\} \cup \mathrm{FV}_\diamond(A) \\
\mathrm{FV}_\diamond\big(A^{q \triangleright q'} \& B^{r \triangleright r'}\big) & = & \{q,q',r,r'\} \cup \mathrm{FV}_\diamond(A) \cup \mathrm{FV}_\diamond(B) \\
\mathrm{FV}_\diamond\Big(\forall \alpha \in \psi . A \xrightarrow{q \triangleright q'} B\Big) & = & \big(\mathrm{FV}_\diamond(\psi) \cup \mathrm{FV}_\diamond(A) \cup \{q,q'\} \cup \mathrm{FV}_\diamond(B)\big) \setminus \alpha
\end{array}
$$

Figure 6.3: Free resource variables of ARTHUR types

included. However, for clarity we restrict ourselves to lists and binary trees in this thesis, although our ARTHUR implementation actually know no built-in primitives for lists and trees, but only a single mechanism to deal with user-definable recursive data types.

Definition 6.6 (Free Resource Variables of ARTHUR Types)**.** The free resource variables of an ARTHUR type A are denoted by $\mathrm{FV}_\diamond(A)$, and are recursively defined as shown in Figure 6.3. We extend the definition naturally to contexts by summation over the domain $\mathrm{FV}_\diamond(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathrm{FV}_\diamond(\Gamma(x))$. In addition, if ψ is a set of linear constraints, then let $\mathrm{FV}_\diamond(\psi)$ also denote the set of resource variables occurring within all constraints.

We must also fix static sizes for operational values based on their associated ARTHUR type, as we did for $\mathrm{LF}_\diamond$ before. We prove the correspondence to the Definition 6.3 by Lemma 6.17, after memory consistency has been defined.

Definition 6.7 (Sizes of ARTHUR Types)**.** We refine Definition 4.15 from the pre-

vious chapter to cover annotated ARTHUR types in the following way

$$\begin{aligned}
\textsc{TypSize}\big(\mathsf{unit}\big) &= 1 \\
\textsc{TypSize}\big(\mathsf{bool}\big) &= 1 \\
\textsc{TypSize}\big(A \times B\big) &= \textsc{TypSize}\big(A\big) + \textsc{TypSize}\big(B\big) \\
\textsc{TypSize}\big((A,q)+(B,r)\big) &= 1 \\
\textsc{TypSize}\big(\mathsf{list}(A,q)\big) &= 1 \\
\textsc{TypSize}\big(A^{q \rhd q'} \& B^{r \rhd r'}\big) &= 1 \\
\textsc{TypSize}\big(\forall \alpha \in \psi. A \xrightarrow{q \rhd q'} A\big) &= 1
\end{aligned}$$

We extend the definition as usual to contexts by summation, i.e.

$$\textsc{TypSize}\big(\Gamma\big) = \sum_{x \in \mathrm{dom}(\Gamma)} \textsc{TypSize}\big(\Gamma(x)\big)$$

Closures for linear pairs and functions are never stored directly in a container, similarly to sums and lists they are boxed, i.e. only a pointer is held in such a case. The size of a pointer was arbitrarily chosen to be one, so we set $\textsc{TypSize}\big(A^{q \rhd q'} \& B^{r \rhd r'}\big)$ and $\textsc{TypSize}\big(\forall \alpha \in \psi. A \xrightarrow{q \rhd q'} A\big)$ to one accordingly.

Let Γ be a ARTHUR typing context mapping a finite set of identifiers to ARTHUR types, and let p, p' be resource variables ranging over $\mathbb{Q}^+$. An ARTHUR typing judgement

$$\Gamma \vdash^{p}_{p'} e{:}A \mid \phi$$

then reads "the expression e has ARTHUR type A under context Γ; furthermore, for all valuations $\boldsymbol{v}$ that satisfy all constraints in ϕ and all initial memory configurations in accordance with Γ, consisting of a stack $\mathcal{S}$ and a heap $\mathcal{H}$, evaluating e requires at most $\boldsymbol{v}(p) + \Phi_{\mathcal{H}}(\mathcal{S} : \boldsymbol{v}(\Gamma))$ heap cell units and leaves at least $\boldsymbol{v}(p') + \Phi_{\mathcal{H}'}(v : \boldsymbol{v}(A))$ heap cell units available afterwards, where v is the result value and $\mathcal{H}'$ the post-heap". We formalise this statement in Section 6.6 as our main soundness result, Theorem 3, which requires memory consistency as a precondition, of course.

Definition 6.8 (ARTHUR Type Rules)**.**

$$\frac{}{\{\,\} \vdash^{p}_{p'} * : \mathsf{unit} \mid \{p = 0,\, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Unit})$$

$$\frac{c \in \{\mathtt{true}, \mathtt{false}\}}{\{\,\} \vdash^{p}_{p'} c : \mathsf{bool} \mid \{p = 0,\, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Bool})$$

$$\frac{}{x{:}C \vdash^{p}_{p'} x : C \mid \{p = 0,\, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Var})$$

$$\frac{\Gamma_1 \vdash^{p}_{p'_1} e_1 : A \mid \phi \qquad \Gamma_2,\, x{:}A \vdash^{p_2}_{p'} e_2 : C \mid \psi}{\Gamma_1, \Gamma_2 \vdash^{p}_{p'} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : C \mid \phi \cup \psi \cup \{p'_1 = p_2\}} \quad (\textsc{Arthur}\vdash\textsc{Let})$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Gamma) = \mathrm{FV}(e) \setminus \{x, y\} \qquad \alpha \cap \big(\mathrm{FV}_\diamond(\Gamma) \cup \mathrm{FV}_\diamond(\phi)\big) = \emptyset \\ \Gamma, x{:}A, y{:}\forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B \vdash^{q}_{q'} e : B \mid \xi \qquad \phi \cup \psi \vDash \xi \\ \phi \vDash \{p = 1 + \textsc{TypSize}(\Gamma),\ p' = 0\} \cup \bigcup_{D \in \mathrm{img}(\Gamma)} \curlyvee(D \,|D, D)\end{array}}{\Gamma \vdash^{p}_{p'} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : \forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B \mid \phi} \quad (\textsc{Arthur}\vdash\textsc{Rec})$$

$$\frac{}{x{:}A,\, y{:}\forall\alpha \in \psi.A \xrightarrow{r \triangleright r'} B \vdash^{q}_{q'} yx : B \mid \psi \cup \{r = q, r' = q'\}} \quad (\textsc{Arthur}\vdash\textsc{App})$$

$$\frac{\Gamma \vdash^{p_1}_{p'_1} e_t : A \mid \phi \qquad \Gamma \vdash^{p_2}_{p'_2} e_f : A \mid \psi}{\Gamma,\, x{:}\texttt{bool} \vdash^{p}_{p'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi \cup \psi \cup \{p = p_1, p = p_2, p' = p'_1.p' = p'_2\}} \quad (\textsc{Arthur}\vdash\textsc{If})$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Gamma) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \qquad \Gamma \vdash^{p_1}_{p'_1} e_1 : A \mid \phi \qquad \Gamma \vdash^{p_2}_{p'_2} e_2 : B \mid \psi \\ \xi = \{\, p = 1 + \textsc{TypSize}(\Gamma),\ \ p' = 0,\ \ p_1 = q + p,\ \ p_2 = r + p,\ \ p'_1 = q',\ \ p'_2 = r' \ \ \}\end{array}}{\Gamma \vdash^{p}_{p'} (e_1\,\&\,e_2) : A^{q \triangleright q'} \& B^{r \triangleright r'} \mid \phi \cup \psi \cup \xi} \quad (\textsc{Arthur}\vdash\textsc{LinPair})$$

$$\frac{}{x{:}A^{q \triangleright q'} \& B^{r \triangleright r'} \vdash^{p}_{p'} \texttt{fst}(x) : A \mid \{p = q, p' = q'\}} \quad (\textsc{Arthur}\vdash\textsc{Fst})$$

$$\frac{}{x{:}A^{q \triangleright q'} \& B^{r \triangleright r'} \vdash^{p}_{p'} \texttt{snd}(x) : B \mid \{p = r, p' = r'\}} \quad (\textsc{Arthur}\vdash\textsc{Snd})$$

$$\frac{}{x_1{:}A,\, x_2{:}B \vdash^{p}_{p'} (x_1\texttt{,}\ x_2) : A \times B \mid \{p = 0, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Pair})$$

$$\frac{\Gamma,\, x_1{:}A,\, x_2{:}B \vdash^{p}_{p'} e : C \mid \phi}{\Gamma,\, x{:}A \times B \vdash^{p}_{p'} \texttt{match}\, x\, \texttt{with}\ (x_1\texttt{,}\ x_2) \to e : C \mid \phi} \quad (\textsc{Arthur}\vdash\textsc{E-Pair})$$

$$\frac{}{x{:}A \vdash^{p}_{p'} \texttt{Inl}(x) : (A, q_l) + (B, q_r) \mid \{p = q_l + \textsc{TypSize}(\texttt{bool} \times A),\, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Inl})$$

$$\frac{}{x{:}B \vdash^{p}_{p'} \texttt{Inr}(x) : (A, q_l)+(B, q_r) \mid \{p = q_r + \textsc{TypSize}(\mathsf{bool}\times B),\ p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Inr})$$

$$\frac{\Gamma,\ y{:}A \vdash^{p_l}_{p'} e_l : C \mid \phi \qquad \Gamma,\ z{:}B \vdash^{p_r}_{p'} e_r : C \mid \psi \qquad \xi = \Big\{p_l = p + q_l,\ p_r = p + q_r\Big\}}{\Gamma,\ x{:}(A, q_l)+(B, q_r) \vdash^{p}_{p'} \begin{array}{r}\texttt{match}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \\ \texttt{|Inr}(z) \to e_r\end{array} : C \mid \phi \cup \psi \cup \xi} \quad (\textsc{Arthur}\vdash\textsc{E-Sum})$$

$$\frac{\begin{array}{c}\Gamma,\ y{:}A \vdash^{p_l}_{p'} e_l : C \mid \phi \qquad \Gamma,\ z{:}B \vdash^{p_r}_{p'} e_r : C \mid \psi \\ \xi = \Big\{p_l = p + q_l + \textsc{TypSize}\big(\mathsf{bool}\times A\big),\ p_r = p + q_r + \textsc{TypSize}\big(\mathsf{bool}\times B\big)\Big\}\end{array}}{\Gamma,\ x{:}(A, q_l)+(B, q_r) \vdash^{p}_{p'} \begin{array}{r}\texttt{match!}\ x\ \texttt{with}\ \texttt{|Inl}(y) \to e_l \\ \texttt{|Inr}(z) \to e_r\end{array} : C \mid \phi \cup \psi \cup \xi} \quad (\textsc{Arthur}\vdash\textsc{E!-Sum})$$

$$\frac{}{\{\,\} \vdash^{p}_{p'} \texttt{Nil} : \mathsf{list}(A, q) \mid \{p = 0, p' = 0\}} \quad (\textsc{Arthur}\vdash\textsc{Nil})$$

$$\frac{\phi = \{p = q + \textsc{TypSize}\big(A\times\mathsf{list}(A, q)\big),\ p' = 0\}}{x_h{:}A,\ x_t{:}\mathsf{list}(A, q) \vdash^{p}_{p'} \texttt{Cons}(x_h\texttt{,}x_t) : \mathsf{list}(A, q) \mid \phi} \quad (\textsc{Arthur}\vdash\textsc{Cons})$$

$$\frac{\Gamma \vdash^{p}_{p'} e_1 : C \mid \phi \qquad \Gamma,\ x_h{:}A,\ x_t{:}\mathsf{list}(A, q) \vdash^{p_0}_{p'} e_2 : C \mid \psi \qquad \xi = \{p_0 = p + q\}}{\Gamma,\ x{:}\mathsf{list}(A, q) \vdash^{p}_{p'} \texttt{match}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1\texttt{|Cons}(x_h\texttt{,}x_t) \to e_2 : C \mid \phi \cup \psi \cup \xi} \quad (\textsc{Arthur}\vdash\textsc{E-List})$$

$$\frac{\begin{array}{c}\xi = \Big\{p_0 = p + q + \textsc{TypSize}\big(A\times\mathsf{list}(A, q)\big)\Big\} \\ \Gamma \vdash^{p}_{p'} e_1 : C \mid \phi \qquad \Gamma,\ x_h{:}A,\ x_t{:}\mathsf{list}(A, q) \vdash^{p_0}_{p'} e_2 : C \mid \psi\end{array}}{\Gamma,\ x{:}\mathsf{list}(A, q) \vdash^{p}_{p'} \texttt{match!}\ x\ \texttt{with}\ \texttt{|Nil} \to e_1\texttt{|Cons}(x_h\texttt{,}x_t) \to e_2 : C \mid \phi \cup \psi \cup \xi} \quad (\textsc{Arthur}\vdash\textsc{E!-List})$$

Structural rules

$$\frac{\Gamma \vdash^{p}_{p'} e : C \mid \phi}{\Gamma, x{:}A \vdash^{p}_{p'} e : C \mid \phi} \quad (\textsc{Arthur}\vdash\textsc{Weak})$$

$$\frac{\Gamma \vdash^{p}_{p'} e : C \mid \phi \qquad \psi \vDash \phi}{\Gamma \vdash^{p}_{p'} e : C \mid \psi} \quad (\textsc{Arthur}\vdash\textsc{Generalise})$$

$$\frac{\Gamma, x{:}B \vdash^{p}_{p'} e : C \mid \phi \qquad \psi \vDash A <: B}{\Gamma, x{:}A \vdash^{p}_{p'} e : C \mid \phi \cup \psi} \text{(ARTHUR}\vdash\text{SUPERTYPE)}$$

$$\frac{\Gamma \vdash^{p}_{p'} e : C \mid \phi \qquad \psi \vDash C <: D}{\Gamma \vdash^{p}_{p'} e : D \mid \phi \cup \psi} \text{(ARTHUR}\vdash\text{SUBTYPE)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : C \mid \phi}{\Gamma \vdash^{p}_{p'} e : C \mid \phi \cup \{p \geq q,\, p - q \geq p' - q'\}} \text{(ARTHUR}\vdash\text{RELAX)}$$

$$\frac{\Gamma,\, x{:}A_1,\, y{:}A_2 \vdash^{p}_{p'} e : C \mid \phi \qquad \psi \vDash \curlyvee(A \,|\, A_1, A_2)}{\Gamma,\, z{:}A \vdash^{p}_{p'} e[z/x, z/y] : C \mid \phi \cup \psi} \text{(ARTHUR}\vdash\text{SHARE)}$$

Similar to LF, all terminal rules require that the context does not include any unused typings. This is not a restriction in any way, since we can always get rid of unnecessary typing through the application of the structural weakening type rule. While the reason to use explicit weakening was largely cosmetic for LF, it is more important for ARTHUR. The type rules for function abstraction and the creation of additive pairs actually contain premises that dictate that the context only mentions variables that are actually mentioned in the term. This is required, since the context is also the basis to determine the size of the created closure through TYPSIZE.

Another peculiarity of type rule ARTHUR$\vdash$REC is the requirement that all types in the context for the function can be shared to themselves. The reason lies in an observation that will be formulated as Lemma 6.22, which essentially says that the potential of such types must be zero. Restricting the context stored in function closure to zero potential is important to allow the unlimited application of that closure, as outlined in Section 6.1.3.

One should also note that the type rule ARTHUR$\vdash$RELAX, in addition to the roles described for rule LF$\vdash_{\diamond}$RELAX in Section 5.1.1, takes on yet one more feature, namely the simple renaming of the resource variables for the petty potential.

Renaming of resource variables inside types is possible through subtyping and supertyping, which also add another way to introduce over-approximation of costs in ARTHUR, in addition to the application of ARTHUR$\vdash$RELAX and ARTHUR$\vdash$WEAK.

The type rules ARTHUR⊢SUPERTYPE and ARTHUR⊢SUBTYPE depend on the ternary subtyping relation, which we define next. We follow the standard notation $A <: B$, as used by Pierce in [Pie02], which reads as "every value of type A is also a value of type B", hence B is the "bigger" type in the sense that it encompasses more values. Note, however, that we will have $\mathsf{list}(\mathsf{unit}, 7) <: \mathsf{list}(\mathsf{unit}, 2)$, so the inequality is turned the other way around for resource annotations. This is indeed natural, since the type $\mathsf{list}(\mathsf{unit}, 7)$ holds a higher potential and is thus much more "precious or scarce", than the more "common" type $\mathsf{list}(\mathsf{unit}, 2)$.

Definition 6.9 (Subtyping)**.** For any two ARTHUR types A, B and constraint set ξ, we inductively define the ternary subtyping relation $\xi \vDash A <: B$ by the rules shown in Figure 6.4.

We write $A <: B$ as a shorthand for $\emptyset \vDash A <: B$.

Our implementation uses a simple heuristic for choosing σ when applying SUBFUN, which unifies the bound variables of the involved types as far as possible

For any fixed ξ, the relation is both *reflexive* and *transitive*, but not necessarily anti-symmetric, e.g. two different, but α-equivalent ARTHUR types are subtyps of each other under the constraints that express α-equivalence.

Lemma 6.10 (Transitivity of Subtyping)**.** *Fix a set ξ of linear constraints. For three arbitrary* ARTHUR *types A, B, C, if we have both $\xi \vDash A <: B$ and $\xi \vDash B <: C$, then also $\xi \vDash A <: C$ holds.*

Proof. The proof is by induction of the length of the derivation for subtyping, and trivial for all cases, except for SUBFUN, which follows by the composition of the two substitutions obtained from the premises. This is harmless thanks to the Barendregt convention [Bar85] that has been adopted here, which assumes that all bound variables are always suitably fresh. □

The sharing relation from $\mathrm{LF}_\diamond$ turns into a function for ARTHUR, since instead of verifying constraints, it just returns the constraints.

Definition 6.11 (ARTHUR Sharing)**.** The partial function $\curlyvee$, which maps three ARTHUR types to a set of constraints, is recursively defined as shown in Figure 6.5, and is undefined otherwise.

$$\frac{}{\xi \vDash A <: A} \quad \text{(SubRef)}$$

$$\frac{\phi \vDash D <: A \qquad \psi \vDash C <: B \qquad \xi \vDash \phi \cup \psi}{\xi \vDash D \times C \ <: \ A \times B} \quad \text{(SubPair)}$$

$$\frac{\phi \vDash D <: A \qquad \psi \vDash C <: B \qquad \xi \vDash \phi \cup \psi \cup \{d \geq a, c \geq b\}}{\xi \vDash (D, d) + (C, c) <: (A, a) + (B, b)} \quad \text{(SubSum)}$$

$$\frac{\psi \vDash B <: A \qquad \xi \vDash \psi \cup \{b \geq a\}}{\xi \vDash \mathsf{list}(B, b) <: \mathsf{list}(A, a)} \quad \text{(SubList)}$$

$$\frac{\begin{array}{c}\phi \vDash C <: A \qquad \psi \vDash D <: B \\ \xi \vDash \phi \cup \psi \cup \{a \geq c,\ c' \geq a',\ b \geq d,\ d' \geq b'\}\end{array}}{\xi \vDash C^{c \triangleright c'} \& D^{d \triangleright d'} \ <: \ A^{a \triangleright a'} \& B^{b \triangleright b'}} \quad \text{(SubLin)}$$

$$\frac{\begin{array}{c}\sigma : \beta \to \mathsf{RV} \text{ a substitution} \\ \xi \cup \phi \vDash \sigma(\psi) \qquad \xi \cup \phi \vDash \{q \geq \sigma(p),\ \sigma(p') \geq q'\} \\ \xi \cup \phi \vDash D <: \sigma(C) \qquad \xi \cup \phi \vDash \sigma(A) <: B\end{array}}{\xi \vDash \forall \beta \in \psi. C \xrightarrow{p \triangleright p'} A <: \forall \alpha \in \phi. D \xrightarrow{q \triangleright q'} B} \quad \text{(SubFun)}$$

Figure 6.4: Ternary subtyping relation for ARTHUR

$$
\begin{aligned}
\curlyvee(\mathsf{unit} \,|\, \mathsf{unit}, \mathsf{unit}) &= \emptyset \\
\curlyvee(\mathsf{bool} \,|\, \mathsf{bool}, \mathsf{bool}) &= \emptyset \\
\curlyvee\left(A \times B \,\middle|\, \begin{matrix} A_1 \times B_1, \\ A_2 \times B_2 \end{matrix}\right) &= \curlyvee(A \,|\, A_1, A_2) \;\cup\; \curlyvee(B \,|\, B_1, B_2) \\
\curlyvee\left((A, a) + (B, b) \,\middle|\, \begin{matrix} ((A_1, a_1) + (B_1, b_1)), \\ ((A_2, a_2) + (B_2, b_2)) \end{matrix}\right) &= \begin{matrix} \{a = a_1 + a_2,\, b = b_1 + b_2\} \cup \\ \curlyvee(A \,|\, A_1, A_2) \;\cup\; \curlyvee(B \,|\, B_1, B_2) \end{matrix} \\
\curlyvee\big(\mathsf{list}(A, a) \,|\, \mathsf{list}(A_1, a_1), \mathsf{list}(A_2, a_2)\big) &= \{a = a_1 + a_2\} \;\cup\; \curlyvee(A \,|\, A_1, A_2) \\
\curlyvee\left(\forall \alpha \in \psi. B \xrightarrow{q \triangleright q'} C \,\middle|\, \begin{matrix} \forall \alpha \in \psi. B \xrightarrow{q \triangleright q'} C, \\ \forall \alpha \in \psi. B \xrightarrow{q \triangleright q'} C \end{matrix}\right) &= \emptyset
\end{aligned}
$$

Figure 6.5: Sharing relation for ARTHUR

Note that functions may be freely shared and reused; while additive pairs cannot be duplicated at all, as one would expect from our observations in Section 6.1.3. The potential of boxed types (lists and sums) is split between the copies, thus preserving the overall potential, which we will formalise by Lemma 6.22.

Any type can only be shared to subtypes of itself. This is intuitively clear, since sharing only affect the resource variable annotations of a type, and generates the constraints that ensures that the resource variables of the shared types are sum up to the one in the original type.

Lemma 6.12 (Shared Subtyping)**.** *If* $\curlyvee(A \,|\, B, C) = \phi$ *holds then also* $\phi \vDash A <: B$ *and* $\phi \vDash A <: C$ *as well.*

Proof. The proof is by induction over the size of the type.

The claim is trivial for base types and function types, since these are identical under sharing and subtyping is reflexive regardless of ϕ.

It is also trivial for additive pairs, since these cannot be shared at all.

For pairs, we simply apply the induction hypothesis twice and apply the subtyping rule SUBPAIR.

For sums and lists we observe that for each resource variable a in A we have the equality $a = b + c$ in ϕ, where b, c are the corresponding resource variables at the same position in B and C, respectively. Since resource variables are always non-negative, we deduce both $a \geq b$ and $a \geq c$ from the equation as expected. This and the induction hypothesis then allows us to apply subtyping rule SUBSUM or SUBLIST, depending on the case, to obtain the required result. □

Due to the presence of structural rules in ARTHUR, we cannot know whether the typing judgement for an abstraction term was derived by the application of type rule ARTHUR⊢REC. Therefore we cannot know whether the type given for an abstraction is indeed a function type or whether all the premises of ARTHUR⊢REC hold. We therefore require an inversion lemma (or sometimes called generation lemma), which essentially shows that this indeed the case. A similar problem arises for all ARTHUR types, but we only require this property for additive pairs and function types.

Lemma 6.13 (Inversion Lemma)**.** *We state standard inversion for functions and additive pairs.*

Functions. *If the* ARTHUR *typing judgement* $\Gamma \vdash^{p}_{p'} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : C \mid \phi$ *is derivable, then there exists* $\Gamma_0, \xi_0, \forall \alpha_0 \in \psi_0. A_0 \xrightarrow{q_0 \triangleright q_0'} B_0$ *such that all of the following holds*

$$\phi \vDash \big(\forall \alpha_0 \in \psi_0. A_0 \xrightarrow{q_0 \triangleright q_0'} B_0\big) <: C \tag{6.1}$$

$$\Gamma_0, x{:}A_0, y{:}\forall \alpha_0 \in \psi_0. A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \vdash^{q_0}_{q_0'} e : B_0 \mid \xi_0 \tag{6.2}$$

$$\Gamma_0 \subseteq \Gamma \quad \wedge \quad \mathrm{dom}(\Gamma_0) = \mathrm{FV}(e) \setminus \{x, y\} \tag{6.3}$$

$$\alpha_0 \cap \big(\mathrm{FV}_\diamond(\Gamma_0) \cup \mathrm{FV}_\diamond(\phi)\big) = \emptyset \tag{6.4}$$

$$\phi \cup \psi_0 \vDash \xi_0 \tag{6.5}$$

$$\phi \vDash \bigcup_{z \in \Gamma_0} \curlyvee\big(\Gamma_0(z) | \Gamma_0(z), \Gamma_0(z)\big) \tag{6.6}$$

Note the premise $\phi \vDash p = 1 + \mathrm{TYPSIZE}(\Gamma_0),\ p' = 0$ *of rule* ARTHUR⊢REC *is missing among the conclusions of the lemma. In fact, both* p *and* p' *are not mentioned at all. The reason is that we need not care for how the creation of the closure had been paid for at the point of closure application, which is where this generation lemma will be needed.*

Linear Pairs. *If the* ARTHUR *typing judgement* $\Gamma \vdash^{p}_{p'} (e_1 \,\&\, e_2) : C \mid \xi$ *is derivable, then there exists* $\Gamma_0, \phi_0, \psi_0, A^{q_0 \triangleright q_0'} \& B^{r_0 \triangleright r_0'}$ *such that all of the following holds*

$$\phi \vDash \left(A^{q_0 \triangleright q_0'} \& B^{r_0 \triangleright r_0'}\right) <: C \tag{6.7}$$

$$\Gamma_0 \vdash^{q_0}_{q_0'} e_1 : A \mid \phi_0 \tag{6.8}$$

$$\Gamma_0 \vdash^{r_0}_{r_0'} e_2 : B \mid \psi_0 \tag{6.9}$$

$$\Gamma_0 \subseteq \Gamma \quad \wedge \quad \mathrm{dom}(\Gamma_0) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \tag{6.10}$$

$$\xi \vDash \phi_0 \cup \psi_0 \tag{6.11}$$

Proof. The proof is by induction on the length of the type derivation of the premise. Since the term is specified, only a few type rules apply in the last step towards the given judgement, namely any of the structural rules or either ARTHUR⊢REC for a function definition or ARTHUR⊢LINPAIR an additive pair.The proof is standard and hence we only consider the case for function abstraction here.

ARTHUR⊢REC This case is trivial, since we have $\Gamma_0 = \Gamma$ and $\xi_0 = \xi$ as well as $\forall \alpha_0 \in \psi_0 . A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 = C$. Conclusion (6.1) holds trivially since subtyping is reflexive; the remaining conclusions are all verbatim among the premises of type rule ARTHUR⊢REC, with (6.6) being a weakened version.

ARTHUR⊢WEAK This case follows trivially, since the conclusions from the induction hypothesis remain unchanged, since by the (6.3), Γ_0 was already a minimal subset of Γ, so any element thrown away by weakening could not appear in Γ_0 anyway.

ARTHUR⊢GENERALISE This case is also trivial, since the induction hypothesis yields all the desired results for a weaker constraint set, that is entailed by ϕ according to the premise of ARTHUR⊢GENERALISE.

ARTHUR⊢SUPERTYPE Let $z{:}D \in \Gamma$ be the variable focused on by the type rule, with $\phi_2 \vDash D <: D'$ and $\phi = \phi_1 \cup \phi_2$. We apply the induction hypothesis for $\Gamma' \vdash^{p}_{p'}$ `rec` y `=` `fun` $x \to e : C \mid \phi_1$ where $\Gamma' = \Gamma \backslash z, z{:}D'$ and obtain $\Gamma_0', x{:}A_0, y{:}\forall \alpha_0 \in \psi_0 . A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \vdash^{q_0}_{q_0'} e : B_0 \mid \xi_0'$.

Assume $z \in \Gamma_0'$, since otherwise the claim is trivial. We apply the type rule ARTHUR⊢SUPERTYPE to derive

$$\Gamma_0, x{:}A_0, y{:}\forall \alpha_0 \in \psi_0 . A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \vdash^{q_0}_{q_0'} e : B_0 \mid \xi_0' \cup \phi_2$$

where $\Gamma_0 = \Gamma_0' \backslash z, z{:}D$ as required. Almost all claims then follow easily by $\phi \vDash \phi_1$. Except $\phi \cup \psi_0 \vDash \xi_0' \cup \phi_2$, which follows from $\phi_1 \cup \psi_0 \vDash \xi_0'$ by additionally using $\phi_2 \subseteq \phi$, concluding this case.

ARTHUR⊢SUBTYPE According to the type rule ARTHUR⊢SUBTYPE, the constraint set ϕ can be partitioned into ϕ_1 and ϕ_2 such that both $\phi_2 \vDash D <: C$ and $\Gamma \vdash^{p}_{p'} \texttt{rec}\ y = \texttt{fun}\ x \to e : D \mid \phi_1$ for some type D hold. We apply the induction hypothesis and observe that (6.1) holds by the transitivity of the subtype relation, Lemma 6.10. The conclusions (6.5) and (6.6) hold since ϕ_1 is a subset of ϕ. The other conclusions follow unchanged from the induction hypothesis.

ARTHUR⊢RELAX According to the typing rule ARTHUR⊢RELAX, we have $\phi = \psi \cup \{p \geq q, p - q \ \geq p' - q'\}$ and the typing $\Gamma \vdash^{q}_{q'} \texttt{rec}\ y = \texttt{fun}\ x \to e : C \mid \psi$. Applying the induction hypothesis directly yields the desired conclusions, with the exception of using the weaker constraint set ψ. Since $\phi \vDash \psi$, we can replace ψ throughout by ϕ, concluding the proof. Intuitively, the case is trivial since relaxation only deals with the petty potential for defining a function, which is unrelated to the cost of executing a function.

ARTHUR⊢SHARE This case follows trivially from the induction hypothesis, since the conclusions remain unchanged, except for contracting two variables in Γ'_0.

□

6.4 Arthur's Memory Consistency

We now formulate the memory consistency invariant, similar to the one previously stated for LF in Section 4.3.2. Memory consistency expresses that values found in memory fit their expected types. For functions and additive pairs, it is not just enough to find a closure in the memory, but it must also entail that the term within the closure is well-typed, too.

A problem arises in dealing with closures of recursive functions, since the consistency of a closure with a certain type depends on the consistency of the values that are stored in the closure. So for a recursive function, the consistency of the closure naturally depends on itself, thereby leading to infinite derivations. We are following the approach of Milner and Tofte [MT91], who define their memory consistency relation coinductively.

We remark that it might be possible to avoid the coinductive definition, since we expect that any infinite derivation for memory consistency would only mention a finite

number of related triples. Therefore one could obtain finite derivations by pruning all repetitions, similarly to the technique of undefining previously encountered locations in order to avoid circular structures, as used in Definition 4.9, the memory consistency for LF. However, adding a set of previously encountered triples appears to require a cumbersome notation. Furthermore, we gain little from an inductive definition, since the memory consistency only serves as an invariant in our soundness theorem, but does not play any role in any implementation of the automated amortised analysis technique. Quite the opposite is true, since the coinductive definition easily allows us to accept the existence of circular data structures. Although circular data cannot be created in ARTHUR, it can processed by ARTHUR programs, as we have previously discussed in Section 4.3.3. Another benefit of the coinductive definition is that throughout the whole derivation, all occurring heaps are identical, which is not true for Definition 4.9, where the premises of some derivation rules are concerned with partial heaps only.

A good tutorial to the concept of coinductive definitions has been published by Gordon [Gor94]. The basic idea of a coindutive definition is due to the Knaster-Tarski theorem, whose consequence is that the greatest fixpoint of a monotone operator F is equal to the union of all consistent sets, i.e. $X \subseteq F(X)$. In our context, the set contains triples, each consisting of a heap, a value and a type; while the monotone operator is the set of inference rules given below. This operator is obviously monotone, since adding more assumptions can never prevent the application of previously valid inference rules. We distinguish inference rules that are to be interpreted coinductively with double horizontal lines, following the convention used by Leroy and Grall [LG09].

Definition 6.14 (Memory Consistency)**.** Memory consistency is a coinductively defined family of ternary relations between a heap $\mathcal{H}$, a value v and an ARTHUR type A. Each memory consistency relation is indexed by a valuation $\boldsymbol{v}$. There are no interdependencies among the family, i.e. any derivation only mentions a single valuation.

We write $\mathcal{H} \vDash_{\boldsymbol{v}} v :: A$ to denote that the value v is consistent with ARTHUR type A in the heap $\mathcal{H}$ under valuation $\boldsymbol{v}$. The relation is coinductively defined to be the largest relation, such that each related triple is justified by one of the following rules.

$$\frac{}{\mathcal{H} \vDash_{\boldsymbol{v}} () :: \mathsf{unit}} \quad \text{(CMUNIT)}$$

$$\frac{}{\mathcal{H} \vDash_{\boldsymbol{v}} \mathbf{tt} :: \mathsf{bool}} \quad \text{(CMTRUE)}$$

$$\frac{}{\mathcal{H} \vDash_v \mathbf{ff} :: \mathsf{bool}} \quad (\textsc{CMfalse})$$

$$\frac{\mathcal{H} \vDash_v v :: A \qquad \mathcal{H} \vDash_v w :: B}{\mathcal{H} \vDash_v (v, w) :: A \times B} \quad (\textsc{CMtuple})$$

$$\frac{\mathcal{H}(\ell) = (\mathbf{tt}, v) \qquad \mathcal{H} \vDash_v v :: A}{\mathcal{H} \vDash_v \ell :: (A, q) + (B, q)} \quad (\textsc{CMinl})$$

$$\frac{\mathcal{H}(\ell) = (\mathbf{ff}, v) \qquad \mathcal{H} \vDash_v v :: B}{\mathcal{H} \vDash_v \ell :: (A, q) + (B, q)} \quad (\textsc{CMinr})$$

$$\frac{}{\mathcal{H} \vDash_v \mathbf{Null} :: \mathsf{list}(A, q)} \quad (\textsc{CMnull})$$

$$\frac{\mathcal{H} \vDash_v \mathcal{H}(\ell) :: A \times \mathsf{list}(A, q)}{\mathcal{H} \vDash_v \ell :: \mathsf{list}(A, q)} \quad (\textsc{CMcons})$$

$$\frac{\begin{array}{c}\mathcal{H}(\ell) = \mathbf{Bad} \\ \text{Type } C \text{ has boxed values, i.e. } C \notin \{\mathsf{unit}, \mathsf{bool}, A \times B\} \text{ for all } A, B.\end{array}}{\mathcal{H} \vDash_v \ell :: C} \quad (\textsc{CMbad})$$

$$\frac{\begin{array}{c}\mathcal{H}(\ell) = \langle (e_1 \,\&\, e_2) \,,\, \mathcal{S}^\star \rangle \\ \text{There exists } \Gamma^\star, p^\star, p'^\star, \phi^\star \text{ such that all of the following are satisfied:} \\ \mathcal{H} \backslash \ell \vDash_v \mathcal{S}^\star :: \Gamma^\star \qquad \boldsymbol{v} \vDash \phi^\star \qquad \Gamma^\star \vdash^{p^\star}_{p'^\star} (e_1 \,\&\, e_2) : A^{a \triangleright a'} \& B^{b \triangleright b'} \mid \phi^\star\end{array}}{\mathcal{H} \vDash_v \ell :: A^{a \triangleright a'} \& B^{b \triangleright b'}} \quad (\textsc{CMadd})$$

$$\frac{\begin{array}{c}\mathcal{H}(\ell) = \langle y{:}x.e \,,\, \mathcal{S}^\star \rangle \\ \text{There exists } \Gamma^\star, p^\star, p'^\star, \phi^\star \text{ such that all of the following are satisfied:} \\ \mathcal{H} \vDash_v \mathcal{S}^\star :: \Gamma^\star \qquad \boldsymbol{v} \vDash \phi^\star \qquad \Gamma^\star \vdash^{p^\star}_{p'^\star} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : \forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} B \mid \phi^\star\end{array}}{\mathcal{H} \vDash_v \ell :: \forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} B} \quad (\textsc{CMfun})$$

We extend this definition pointwise to contexts as usual by writing $\mathcal{H} \vDash_v \mathcal{S} :: \Gamma$ if and only if for all $x \in \mathrm{dom}(\Gamma)$ holds $\mathcal{H} \vDash_v \mathcal{S}(x) :: \Gamma(x)$.

The notion of memory consistency must be strong enough to describe the resource usage of a closure, as previously outlined at the end of Section 6.2. Hence, the statement necessarily involves an annotated ARTHUR typing judgement for the terms that are stored in the heap memory. The memory consistency invariant for ARTHUR is thus significantly more complex than for LF. The occurring typing judgements have intentionally no restrictions on the petty potential $p^\star, p'^\star$. The petty potential only expresses the cost of constructing the closure, but since the closure already exists in memory, this value is of no concern anymore.

The two rules dealing with closures, rule CMADD and CMFUN, both ensure that the term stored within a closure is in accordance with the type for the closure by requiring the existence of an ARTHUR typing derivation for the stored term. Both rules also require that the stored environment is consistent with the existentially quantified context. For a recursive function, the environment contains a reference to the closure itself, leading to a non-wellfounded (infinite) derivation.

This is not the case for additive pairs, since the premise in rule CMADD requires that the location of the closure is not mentioned in the derivation of the consistency of the environment. However, this requirement is non-essential: the start of the evaluation of an additive pair immediately causes the deallocation of that additive pair. An additive pair can thus be evaluated at most once, so the use of a reference to itself would cause the evaluation to get stuck before any possible violation of the predicted resource bounds could occur. So requiring that closures of additive pairs are non-recursive has no technical reason, but simply satisfies our intuition about additive pairs.

The existential quantification of closure contexts is a convenient way to restore the typing that was necessarily present at the point of creation of the closure, without actually artificially storing it. The drawback is that we cannot ensure to reconstruct exactly the same typing. An alternative that avoids existential quantification over contexts for closures and which restores the original contexts could possibly be obtained by adopting *store specifications* as used by Abadi and Leino [AL97]. However, we did not investigate this possibility, since a variation in the context is harmless for us: the cost of evaluating a function body, the potential available through the input and the potential returned through the output are all fixed by the ARTHUR type already. The context only yields a justification for connecting the three, but it does not matter for the memory consistency invariant *how* the resource usage is justified, as long as it is justified.

We have already remarked earlier that we cannot hope to prove preservation for ARTHUR independently from the main soundness theorem, since ARTHUR types for

functions and additive pairs imply bounds on the resources required for the evaluation of a term of the given type. It is only through Theorem 3 that memory consistency for ARTHUR becomes strong enough to tell us that a term stored within a closure respects the ascribed resource bounds.

However, while there is no corresponding result to LF Lemma 4.13, we can prove a weakened version, namely that the consistency of pre-existing data is preserved by evaluation. This is intuitively evident due to the stability of the heap as expressed by Lemma 6.5, and that deallocated locations are consistent by definition.

Lemma 6.15 (Consistent Memory Extension)**.** *Let $\mathcal{H}$ and $\mathcal{H}'$ be two heaps with* $\mathrm{dom}(\mathcal{H}) \subseteq \mathrm{dom}(\mathcal{H}')$ *and for all $\ell \in \mathrm{dom}(\mathcal{H})$ either $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ or $\mathcal{H}'(\ell) = \mathbf{Bad}$. If for any value w and any type A we have $\mathcal{H} \vDash_v w :: A$ then also $\mathcal{H}' \vDash_v w :: A$.*

Proof. The proof uses the principle of coinduction: Memory consistency is defined as the largest relation that is consistent with the rules of Definition 6.14. We thus define a new ternary relation that is implied by premises of the Lemma, and then show that this relation is also consistent with the rules of Definition 6.14. By the coinduction principle, memory consistency must then encompass this relation.

We define $R_v\big(\mathcal{H}', w, A\big)$ to hold if and only if there exists a heap $\mathcal{H}$ with $\mathcal{H} \vDash_v w :: A$ and such that for all locations $\ell \in \mathrm{dom}(\mathcal{H})$ either $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ or $\mathcal{H}'(\ell) = \mathbf{Bad}$ holds (which already implies $\mathrm{dom}(\mathcal{H}) \subseteq \mathrm{dom}(\mathcal{H}')$ as well).

Assume $R_v\big(\mathcal{H}', w, A\big)$ holds. We must now show that this statement is justifiable by one of the memory consistency rules, i.e. we must exhibit a memory consistency rule, wherein we may replace the memory consistency relation by $R_v\big(\cdot, \cdot, \cdot\big)$, such that all its premises are satisfied and which has $R_v\big(\mathcal{H}', w, A\big)$ as its conclusion.

If the relation $R_v\big(\mathcal{H}', w, A\big)$ holds, then by definition there exist a $\mathcal{H}$ of which $\mathcal{H}'$ is an extension with deallocations such that $\mathcal{H} \vDash_v w :: A$ holds. We proceed by case distinction on the last memory consistency rule that had been applied to derive $\mathcal{H} \vDash_v w :: A$.

CMUNIT We necessarily have $w = ()$ and $A = \mathsf{unit}$. By rule CMUNIT the value $()$ is consistent with type unit in any heap. Thus $R_v\big(\mathcal{H}', (), \mathsf{unit}\big)$ is justifiable through rule CMUNIT as well.

CMTRUE, CMFALSE These two cases follow analogously to the case for CMUNIT.

CMTUPLE We have $\mathcal{H} \vDash_v (t, u) :: B \times C$ by rule CMTUPLE, hence necessarily both $\mathcal{H} \vDash_v t :: B$ and $\mathcal{H} \vDash_v u :: C$ hold. Therefore, by definition $R_v\big(\mathcal{H}', t, B\big)$ and $R_v\big(\mathcal{H}', u, C\big)$ hold. Thus the premises of rule CMTUPLE for $R_v\big(\mathcal{H}', (t, u), B \times C\big)$ are satisfied as required.

CMINL, CMINR We have $R_v\big(\mathcal{H}', \ell, (C, q) + (B, q)\big)$, where the last rule to derive $\mathcal{H} \vDash_v \ell :: (C, q) + (B, q)$ was by rule CMINL. Thus necessarily $\mathcal{H}(\ell) = (\mathbf{tt}, u)$ and $\mathcal{H} \vDash_v u :: B$.

By assumption we either have $\mathcal{H}'(\ell) = (\mathbf{tt}, u)$ or $\mathcal{H}'(\ell) = \mathbf{Bad}$. In the former case $R_v\big(\mathcal{H}', \ell, (C, q) + (B, q)\big)$ is obviously justifiable by rule CMINL again, whereas in the latter case the all the premises of rule CMBAD are satisfied.

The case for rule CMINR follows correspondingly.

CMNULL This case follows analogously to the case for rule CMUNIT again: Only one unique memory consistency rule is applicable for the value **Null**, and this rule ignores the heap entirely.

CMCONS We have $\mathcal{H} \vDash_v \mathcal{H}(\ell) :: A \times \mathsf{list}(A, q)$, and thus $R_v\big(\mathcal{H}', \mathcal{H}(\ell), A \times \mathsf{list}(A, q)\big)$. We have either $\mathcal{H}(\ell) = \mathcal{H}'(\ell)$ or $\mathcal{H}(\ell) = \mathbf{Bad}$ by the assumptions on $\mathcal{H}$ and $\mathcal{H}'$. In the first case the claim is trivially justifiable by rule CMCONS, and in the second case justification follows through rule CMBAD, whose premises are all satisfied.

CMBAD We have $\mathcal{H}(\ell) = \mathbf{Bad}$ and thus by assumption $\mathcal{H}'(\ell) = \mathbf{Bad}$ as well. Therefore, $R_v\big(\mathcal{H}', \mathbf{Bad}, A\big)$ is directly justifiable through rule CMBAD again.

CMADD We have $\mathcal{H}(\ell) = \big\langle (e_1 \,\&\, e_2)\,,\, \mathcal{S}^\star \big\rangle$ and the existence of $\Gamma^\star, p^\star, p'^\star, \phi^\star$ with $\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma^\star$ and $\boldsymbol{v} \vDash \phi^\star$ and $\Gamma^\star \vdash^{p^\star}_{p'^\star} (e_1 \,\&\, e_2) : C^{c \triangleright c'} \& B^{b \triangleright b'} \mid \phi^\star$.

By assumption either $\mathcal{H}'(\ell) = \mathbf{Bad}$, in which case the premises of rule CMBAD are all satisfied, or $\mathcal{H}'(\ell) = \langle y{:}x.e\,,\, \mathcal{S}^\star \rangle$. Consider the latter. For each $z \in \Gamma^\star$ we have $\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star(z) :: \Gamma^\star(z)$ through the pointwise extension of memory consistency to stacks and contexts, and we can derive $R_v\big(\mathcal{H}'\backslash\ell, \mathcal{S}^\star(z), \Gamma^\star(z)\big)$ as well, since undefining the same location in both $\mathcal{H}$ and $\mathcal{H}'$ does not invalidate the requirement that $\mathcal{H}'\backslash\ell$ is an extension of $\mathcal{H}\backslash\ell$ with possibly some deallocations. Therefore $R_v\big(\mathcal{H}', \ell, C^{c \triangleright c'} \& B^{b \triangleright b'}\big)$ is justifiable through rule CMADD, whose premises are all satisfied with $\Gamma^\star, p^\star, p'^\star, \phi^\star$ unmodified from above being the witnesses for the existentially quantified entities.

CMFUN This case is largely similar to the previous one. We have $\mathcal{H}(\ell) = \langle y{:}x.e\,,\, \mathcal{S}^\star\rangle$ and the existence of $\Gamma^\star, p^\star, p'^\star, \phi^\star$ with $\mathcal{H} \vDash_v \mathcal{S}^\star :: \Gamma^\star$ and $\boldsymbol{v} \vDash \phi^\star$ and also $\Gamma^\star \vdash^{p^\star}_{p'^\star} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : \forall\alpha \in \psi.C \xrightarrow{q \triangleright q'} B \mid \phi^\star$.

There are two case by our assumption on $\mathcal{H}$ and $\mathcal{H}'$. In the case that $\mathcal{H}'(\ell) = \mathbf{Bad}$, the premises of rule CMBAD are satisfied. Otherwise, we have $\mathcal{H}'(\ell) = \langle y{:}x.e\,,\, \mathcal{S}^\star\rangle$. For each $z \in \Gamma^\star$ we obtain $R_v\big(\mathcal{H}', \mathcal{S}^\star(z), \Gamma^\star(z)\big)$ through the definition of pointwise extension of memory consistency to stacks and contexts. $R_v\big(\mathcal{H}', \ell, \forall\alpha \in \psi.C \xrightarrow{q \triangleright q'} B\big)$ is thus justifiable through rule CMFUN, whose premises are all satisfied, using $\Gamma^\star, p^\star, p'^\star, \phi^\star$ unmodified from above as the witnesses for the existentially quantified entities.

□

In combination with heap stability we immediately obtain the following consequence.

Lemma 6.16 (Memory Consistency Preservation)**.** *If $\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}'$ then for all values w, we do have that $\mathcal{H} \vDash_v w :: A$ implies $\mathcal{H}' \vDash_v w :: A$.*

Proof. By Lemma 6.5 we know that for all locations $\ell \in \mathrm{dom}(\mathcal{H})$ either $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$ or $\mathcal{H}'(\ell) = \mathbf{Bad}$ holds and can thus apply Lemma 6.15 to derive $\mathcal{H}' \vDash_v w :: A$ as required. □

Of course, one could have proven preservation for an unannotated version of both the ARTHUR type rules and memory consistency, like we did in Chapter 4 for LF, but this would be of little use for proving our main soundness result for ARTHUR. For proving soundness, we obviously need an invariant that is strong enough to tell us that all closures stored in the heap respect the annotated typing, which, through the induction hypothesis, then tells us that the terms in the closures can be evaluated with the limited resources as detailed by their types.

Other results from LF, which also depend on memory consistency, can be correspondingly formulated for ARTHUR, and carry over without any surprise.

Lemma 6.17 (Size Correspondence)**.** *If $\mathcal{H} \vDash_v v :: A$ then* $\textsc{Size}(v) = \textsc{TypSize}(A)$.

Proof. The proof is by induction on the structure of type A. We proceed by case distinction on the value v

`()` Only one rule of Definition 6.14 can justify the consistency of the unit value. This single rule requires $A = \mathsf{unit}$. We observe that $\textsc{Size}(\texttt{()}) = 1 = \textsc{TypSize}(\mathsf{unit})$ holds as required.

tt, **ff** Similarly to the previous case, only two memory consistency rules apply to boolean values, both requiring $A = \mathsf{bool}$. We observe that $\textsc{Size}(\mathbf{tt}) = \textsc{Size}(\mathbf{ff}) = 1 = \textsc{TypSize}(\mathsf{bool})$ holds as required.

(v_1, v_2) This is the only case that requires the invocation of the induction hypothesis. Only one memory consistency rule can justify a pair value. We necessarily have $A = B_1 \times B_2$ and $\mathcal{H} \vDash_v v_i :: B_i$. Applying the induction hypothesis yields $\textsc{Size}(v_i) = \textsc{TypSize}(B_i)$. Hence we observe $\textsc{Size}\big((v_1, v_2)\big) = \textsc{Size}(v_1) + \textsc{Size}(v_2) = \textsc{TypSize}\big(B_1 \times B_2\big) = \textsc{TypSize}(B_1) + \textsc{TypSize}(B_2)$, where the first equality follows by Definition 6.3 and the third equality follows by Definition 6.7.

Null Necessarily $A = \mathsf{list}(B, q)$, hence $\textsc{Size}(\mathbf{Null}) = 1 = \textsc{TypSize}(\mathsf{list}(B, q)) = 1$.

ℓ We have $\textsc{Size}(\ell) = 1$. Memory consistency for a location requires that type A has boxed values, i.e. it is either a sum, a list, an additive pair or a function type. In all of these four cases we have $\textsc{TypSize}(A) = 1$ as required.

□

A trivial, but important consequence of the previous Lemma is that the type fully determines the size.

Lemma 6.18 (Size Determination)**.** *If we have both* $\mathcal{H} \vDash_v v :: A$ *and* $\mathcal{H} \vDash_v v :: B$ *then* $\textsc{TypSize}(A) = \textsc{TypSize}(B)$ *follows.*

Proof. Applying Lemma 6.17 twice yields the equations $\textsc{TypSize}(A) = \textsc{Size}(v)$ and $\textsc{Size}(v) = \textsc{TypSize}(B)$. □

The subtyping relation must also respect memory consistency, which is expressed by the following lemma.

Lemma 6.19 (Consistent Subtyping)**.** *If the three statements* $\mathcal{H} \vDash_v v :: A$ *and* $\xi \vDash A <: B$ *and* $\boldsymbol{v} \vDash \xi$ *hold, then* $\mathcal{H} \vDash_v v :: B$ *holds as well.*

Proof. The proof is by induction over the length of the derivation for $\xi \vDash A <: B$.

The claim is trivial for SUBREF, since this implies $A = B$ and thus conclusion is equal to the premise.

Rule SUBPAIR only applies for pair types. We can applying the induction hypothesis once for each part of the pair and derive memory consistency by its rule for pairs.

For SUBSUM, we have $\xi \vDash (D, d) + (C, c) <: (A, a) + (B, b)$. We assume $\mathcal{H}(\ell) = (\mathbf{tt}, w)$ without loss of generality, since the proof for the only other case allowed by memory consistency follows by the symmetry of sum values and sum types. By the induction hypothesis, we obtain $\mathcal{H}\backslash\ell \vDash_v w :: A$ and hence also $\mathcal{H} \vDash_v \ell :: (A, a) + (B, b)$ by the definition of memory consistency.

The case for rule SUBLIST we have $\xi \vDash \mathsf{list}(B, b) <: \mathsf{list}(A, a).$ and $\xi \vDash B <: A$. The case is trivial for $v = \mathbf{Null}$, so we assume $v \neq \mathbf{Null}$ and hence by the first assumption $\mathcal{H}\backslash\ell \vDash_v \mathcal{H}(\ell) :: A \times \mathsf{list}(A, q)$, which allows us to use the induction hypothesis on pair types.

The cases of rule SUBLIN and rule SUBFUN follow the same pattern. Memory consistency yields the existence of $\Gamma^\star, p^\star, p'^\star, \phi^\star$ such that $\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma^\star$ and $\boldsymbol{v} \vDash \phi^\star$ and also $\Gamma^\star \vdash^{p^\star}_{p'^\star} e : A \mid \phi^\star$ hold. Using the structural type rule ARTHUR$\vdash$SUBTYPE yields $\Gamma^\star \vdash^{p^\star}_{p'^\star} e : B \mid \phi^\star \cup \xi$ which in turn allows us to deduce the required $\mathcal{H} \vDash_v v :: B$ by the definition of memory consistency, concluding the proof. □

6.5 Arthur's Potential

The definition of potential remains the same as for $\mathrm{LF}_\diamond$, except that a valuation is needed to determine the sum, and that we must extend the definition of potential for the types of additive pair and functions.

Definition 6.20 (Potential)**.** We define the potential of a value v of $\mathrm{LF}_\diamond$ type A within heap $\mathcal{H}$ under valuation $\boldsymbol{v}$, written $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(v : a)$ recursively as shown in Figure 6.6 on the next page.

We extend this definition to $\Phi : \mathrm{Heap} \times \mathrm{Stack} \times \mathrm{LF}_\diamond\text{-type-context} \to \mathbb{Q}^+$ by

$$\Phi_{\mathcal{H}}(\mathcal{S} : \Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x) : \Gamma(x))$$

$$
\begin{aligned}
&\Phi^{v}_{\mathcal{H}}(() : \mathsf{unit}) && = 0\\
&\Phi^{v}_{\mathcal{H}}(\mathbf{tt} : \mathsf{bool}) && = 0\\
&\Phi^{v}_{\mathcal{H}}(\mathbf{ff} : \mathsf{bool}) && = 0\\
&\Phi^{v}_{\mathcal{H}}\big((v, w) : A{\times}B\big) && = \Phi^{v}_{\mathcal{H}}(v : A) + \Phi^{v}_{\mathcal{H}}(w : B)\\
&\Phi^{v}_{\mathcal{H}}\big(\ell : A, q{+}B, p\big) && = \begin{cases} v(q) + \Phi^{v}_{\mathcal{H}}(v : A) & \text{If } \mathcal{H}(\ell) = (\mathbf{tt}, v)\\ v(p) + \Phi^{v}_{\mathcal{H}}(v : B) & \text{If } \mathcal{H}(\ell) = (\mathbf{ff}, v)\\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Bad} \end{cases}\\
&\Phi^{v}_{\mathcal{H}}\big(\ell : \mathsf{list}(A, q)\big) && = \begin{cases} v(q) + \Phi^{v}_{\mathcal{H}\setminus\ell}\big((v_h, v_t) : A{\times}\mathsf{list}(A, q)\big) & \text{If } \mathcal{H}(\ell) = (v_h, v_t)\\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Null}\\ 0 & \text{If } \mathcal{H}(\ell) = \mathbf{Bad} \end{cases}\\
&\Phi^{v}_{\mathcal{H}}\Big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\Big) && = \begin{cases} \min\limits_{\Gamma^\star \in \mathcal{G}} \Phi^{v}_{\mathcal{H}\setminus\ell}(\mathcal{S}^\star : \Gamma^\star) & \text{If } \mathcal{H}(\ell) = \big\langle (e_1 \,\&\, e_2)\,,\, \mathcal{S}^\star \big\rangle\\ 0 & \text{otherwise} \end{cases}\\
&\Phi^{v}_{\mathcal{H}}\Big(\ell : \forall\alpha \in \psi. A \xrightarrow{q \triangleright q'} B\Big) && = 0
\end{aligned}
$$

where $\mathcal{G}$ in the clause for additive pairs stands for the set of all admissible contexts:

$$
\mathcal{G} = \left\{ \Gamma^\star \;\middle|\; \begin{array}{l} \mathcal{H}\setminus\ell \vDash_{v} \mathcal{S}^\star :: \Gamma^\star\\ \Gamma^\star \vdash^{q}_{q'} \ (e_1 \,\&\, e_2) : A^{a \triangleright a'} \& B^{b \triangleright b'} \ \mid\ \psi\\ v \vDash \psi \end{array} \right\}
$$

Figure 6.6: Potential of ARTHUR types

The potential for function closures is always zero, thereby allowing the infinite reuse of function closures, as discussed in Section 6.1.3. The potential associated with a closure for an additive pair may hold some potential through free variables captures at its creation. The captured free variables may contain potential through their own typing, which requires us to recover their typing. Memory consistency guarantees that such a context exists, but there may be more than one. Therefore we choose the context with the minimal potential necessary, which is the safest approximation. If the originally used context at the creation of the closure held more potential, then this amount is lost. However, the type and value of the result fully determines the potential of the result, so any additional potential of the original context would have been lost anyway.

Our next result shows that subtyping is safe with respect to potential, i.e. subtyping can only lower the potential by throwing it away.

Lemma 6.21 (Subtyped Potential)**.** *From the three statements*

$$\mathcal{H} \vDash_v \ell :: A \tag{6.12}$$

$$\xi \vDash A <: B \tag{6.13}$$

$$v \vDash \xi \tag{6.14}$$

follows $\Phi^v_{\mathcal{H}}(\ell : A) \geq \Phi^v_{\mathcal{H}}(\ell : B)$ *as well.*

Proof. We use induction on the number of summands in $\Phi^v_{\mathcal{H}}(\ell : A)$. The proof is trivial for base types and function types, since neither of these types may contribute any potential under any circumstances.

For pairs, the proof follows trivially by applying the induction hypothesis twice.

For sums, we have $\xi \vDash \Big((C, q) + (D, p)\Big) <: \Big((C^\star, q^\star) + (D^\star, p^\star)\Big)$. Without loss of generality, assume $\mathcal{H}(\ell) = (\mathbf{tt}, w)$, since the other case then follows by symmetry. Thus we have $\Phi^v_{\mathcal{H}}(\ell : (C, q) + (D, p)) = v(q) + \Phi^v_{\mathcal{H}}(w : C)$. From (6.14) and the subtyping rule SUBSUM we have $v(q) \geq v(q^\star)$ and from the induction hypothesis follows $\Phi^v_{\mathcal{H}}(w : C) \geq \Phi^v_{\mathcal{H}}(w : C^\star)$, concluding this case since we have shown that each part of the sum is replaced by a value that is less then or equal to the original value.

For lists, we observe, similarly to the case for sums, that the subtyping rule requires ξ to enforce that the annotation of the subtype is less or equal to the supertype, while the required inequality for the other summand follows by the induction hypothesis.

For additive pairs, we have $\xi \vDash C^{c \triangleright c'} \& D^{d \triangleright d'} <: A^{a \triangleright a'} \& B^{b \triangleright b'}$, the most interesting case. We have $\Phi^v_{\mathcal{H}}\big(\ell : C^{c \triangleright c'} \& D^{d \triangleright d'}\big) = \min_{\Gamma^\star \in \mathcal{G}} \Phi^v_{\mathcal{H}\backslash\ell}(\mathcal{S}^\star : \Gamma^\star)$ where $\mathcal{G}$ is set of all admissible contexts for the additive pair, i.e.

$$\mathcal{G} = \left\{ \Gamma^\star \,\middle|\, \mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma^\star \ \wedge\ \Gamma^\star \vdash^{q^\star}_{q'^\star} (e_1 \,\&\, e_2) : C^{c \triangleright c'} \& D^{d \triangleright d'} \mid \psi \ \wedge\ v \vDash \psi \right\}$$

which is non-empty by (6.12). However, each context in $\Gamma_0 \in \mathcal{G}$ can also be used to derive $\Gamma_0 \vdash^{q^\star}_{q'^\star} (e_1 \,\&\, e_2) : A^{a \triangleright a'} \& B^{b \triangleright b'} \mid \psi$ through an application of type rule ARTHUR⊢SUBTYPE by premise (6.13). Therefore, the set of contexts ranged over to obtain the minimal potential for the type $A^{a \triangleright a'} \& B^{b \triangleright b'}$ contains $\mathcal{G}$, which implies that the minimum must be smaller or equal, as required. □

One would expect an even stronger result, namely that the potential that is obtained for two shared entities is equal to the potential of the original entity, i.e. that sharing does not increase potential. This is indeed the case, as the next lemma shows.

Lemma 6.22 (Shared Potential)**.** *If $\mathcal{H} \vDash_v \ell :: A$ and $\curlyvee(A \,|\, B, C) = \phi$ holds and $v \vDash \phi$ then $\Phi^v_{\mathcal{H}}(\ell : A) = \Phi^v_{\mathcal{H}}(\ell : B) + \Phi^v_{\mathcal{H}}(\ell : C)$. Moreover, for $A = B$ and $A = C$, it follows that $\Phi^v_{\mathcal{H}}(\ell : A) = 0$ also holds.*

Proof. The claim is trivial for base types and function types, since they never carry potential. It is also trivial for additive pairs, since those cannot be shared at all.

For the remaining cases, we have by Lemma 6.12 that $\phi \vDash A <: B$ and $\phi \vDash A <: C$ and obtain $\mathcal{H} \vDash_v \ell :: B$ and $\mathcal{H} \vDash_v \ell :: C$. The proof is then similar to that for Lemma 6.21. The final observation follows since $q = 0$ is the only solution to $q = q + q$. □

Note that this lemma almost entails Lemma 6.21 via the subtyping of shared types, Lemma 6.12, except for the interesting case for additive pairs, which cannot be deduced via Lemma 6.12, since additive pairs cannot be shared.

6.6 Soundness of the Higher-Order Analysis

The ARTHUR type for a term is intended to describe its worst-case heap space usage, through the definition of potential. We will now prove this intention to be sound. The soundness of the ARTHUR type system is the main result of this thesis.

Intuitively, the theorem says that for an ARTHUR typing judgement $\Gamma \vdash^{p}_{p'} e{:}A \mid \phi$ and given a valuation v that satisfies the constraints ϕ, the evaluation of term e in a consistent memory configuration requires at most $v(p) + \Phi_{\mathcal{H}}(\mathcal{S} : v(\Gamma))$ heap units.

Moreover, the theorem also tells us about the amount of heap units that are guaranteed to be unused after the evaluation is finished. These may or may not have been used during the evaluation. For technical reasons, the theorem also includes a value r, which represents additional free heap cell units, and shows that these are harmless and preserved, similar in the sense of Lemma 4.18.

Theorem 3 (Soundness ARTHUR)**.**

$$\Gamma \vdash^{p}_{p'} e{:}D \mid \phi \tag{3.A}$$
$$\mathcal{S}, \mathcal{H} \vdash e \downarrow v, \mathcal{H}' \tag{3.B}$$
$$v \vDash \phi \tag{3.C}$$
$$\mathcal{H} \vDash_{v} \mathcal{S} :: \Gamma \tag{3.D}$$

If the four statements above are all satisfied, then for all $r \in \mathbb{Q}^{+}$ and $m \in \mathbb{N}$ with

$$m \geq v(p) + \Phi_{\mathcal{H}}(\mathcal{S} : v(\Gamma)) + r$$

there exists an $m' \in \mathbb{N}$ satisfying

$$m' \geq v(p') + \Phi_{\mathcal{H}'}(v : v(D)) + r$$

such that $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} e \downarrow^{\circ} v, \mathcal{H}'$ holds and furthermore $\mathcal{H}' \vDash_{v} v :: D$ as well.

Note that the important property of memory consistency conservation during evaluation, i.e. that $\mathcal{H}' \vDash_{v} \mathcal{S} :: \Gamma$ holds, is missing from the conclusions, since this follows independently[C] by Lemma 6.16.

Proof. The proof is by induction on the derivation of premise (3.B) and a subordinate induction on the derivation of premise (3.A). An induction over the length of the evaluation derivation alone would be insufficient for the cases when (3.A) was derived by a structural rule, since evaluation remains the same in these cases. An induction over the length of the typing derivation would fail for the cases when (3.A) was derived by function application or the projection of additive pairs; these are terminal rules, i.e. they typing derivation has length 1, but we need to apply the theorem for

[C]The author thanks Hugo Simoes for pointing out this convenient simplification of the proof.

the term stored in the closure, for which we obtain a typing judgement from (3.D), which is a possibly, and most likely, longer.

We begin the examination of the cases for the structural rules, which leave the evaluation (3.B) unchanged and hence also the length of its derivation. For these cases induction is only possible if length of the typing derivation was shortened.

Afterwards we examined the remaining cases, distinguished by the rule that was last applied to derive the evaluation statement. Using the induction hypothesis is then possible if the length of the derivation for (3.B) has been shortened, while the length of the typing derivation becomes unimportant.

The most interesting cases are the ones for function application and the projection of additive pairs. We only treat a few selected cases of the language constructs that are identical in both ARTHUR and $\mathrm{LF}_{\diamond}$, since the proof for their cases follow analogously to the ones shown in the proof of Theorem 1.

ARTHUR⊢WEAK: The type rule ARTHUR⊢WEAK allows the introduction of a superfluous binding $x{:}A$ to the typing context Γ. We show that the validity of all five premises is preserved if we drop the additional binding from the typing context, in order to apply induction for the shorter typing judgement without mentioning the binding $x{:}A$. Since neither of the three conclusions mention the typing context, this already suffices to prove this particular case.

Fix any $r \in \mathbb{Q}^+$ and any m such that $m \geq \boldsymbol{v}(p)+\Phi_{\mathcal{H}}\big(\mathcal{S} : \boldsymbol{v}(\Gamma)\big)+\Phi_{\mathcal{H}}\big(\mathcal{S}(x) : A\big)+r$ holds. Since potential is always non-negative by Definition 6.20, we also have that $m \geq \boldsymbol{v}(p) + \Phi_{\mathcal{H}}\big(\mathcal{S} : \boldsymbol{v}(\Gamma)\big) + r$ holds. We now apply the induction hypothesis for the same premises, apart from dropping $x{:}A$ from the typing context. Applying induction is possible, since the length of the typing derivation (3.A) is exactly one step shorter, whereas the length of the derivation for the evaluation (3.B) remains unchanged. Furthermore observe that $\mathcal{H} \vDash_{\boldsymbol{v}} \mathcal{S} :: \Gamma, \{x{:}A\}$ trivially implies $\mathcal{H} \vDash_{\boldsymbol{v}} \mathcal{S} :: \Gamma$ by Definition 6.14. The induction directly yields all the desired results.

ARTHUR⊢GENERALISE: This type rule allows the strengthening of the linear constraint set. We observe that (3.C) together with the premise $\psi \vDash \phi$ from (3.A) readily implies $\boldsymbol{v} \vDash \phi$ by definition. We can therefore apply the induction hypothesis for the type derivation shortened by the omission of type rule ARTHUR⊢GENERALISE and directly obtain all required conclusions, since the constraint set does not affect any premise nor conclusion apart from (3.A) and (3.C).

ARTHUR⊢SUPERTYPE: Choose any m such that

$$\begin{aligned} m &\geq \boldsymbol{v}(p) + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi^{v}_{\mathcal{H}}(\mathcal{S}(x) : A) + r \\ &\geq \boldsymbol{v}(p) + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + \Phi^{v}_{\mathcal{H}}(\mathcal{S}(x) : B) + r \end{aligned}$$

where the latter inequality follows by Lemma 6.21 and $\boldsymbol{v} \vDash A <: B$ from (3.A) and (3.C).

The induction hypothesis then yields $m' \geq \boldsymbol{v}(p') + \Phi^{v}_{\mathcal{H}}(v : C) + r$ as required. The application of the induction hypothesis is justified, since the derivation for (3.A) was one step shorter, while the length of the derivation for (3.B) remained unchanged. Furthermore the required premise $\mathcal{H} \vDash_v \mathcal{S}(x) :: B$ can be derived by Lemma 6.19.

The memory consistency of the result v follows trivially from the application of the induction hypothesis, since both ts type C and (3.B) remain unchanged.

ARTHUR⊢SUBTYPE: We apply the induction hypothesis right away and receive for each $m \geq \boldsymbol{v}(p) + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r$ the existence of

$$\begin{aligned} m' &\geq \boldsymbol{v}(p') + \Phi^{val}_{\mathcal{H}}(\mathcal{S}(x) : C) + r \\ &\geq \boldsymbol{v}(p') + \Phi^{val}_{\mathcal{H}}(\mathcal{S}(x) : D) + r \end{aligned}$$

as required, where the last inequality follows by Lemma 6.21 from the typing statement's premise $\psi \vDash C <: D$ and $\boldsymbol{v} \vDash \phi \cup \psi$ (3.C).

Memory consistency for the result $\mathcal{H} \vDash_v v :: D$ follows from $\mathcal{H} \vDash_v v :: D$ by Lemma 6.19.

ARTHUR⊢RELAX: We observe

$$\begin{aligned} m &\geq \boldsymbol{v}(p) + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r \\ &= \boldsymbol{v}(q) + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r' \end{aligned}$$

if we choose $r' = r + \boldsymbol{v}(p) - \boldsymbol{v}(q)$ for applying the induction hypothesis, which we can do since $\boldsymbol{v}(p) - \boldsymbol{v}(q) \geq 0$ holds by the constraints added by rule ARTHUR⊢RELAX. We thus obtain the existence of m' with

$$\begin{aligned} m' &\geq \boldsymbol{v}(q') + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r' \\ &= \boldsymbol{v}(q') + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r + \boldsymbol{v}(p) - \boldsymbol{v}(q) \\ &\geq \boldsymbol{v}(q') + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r + \boldsymbol{v}(p') - \boldsymbol{v}(q') \\ &= \boldsymbol{v}(p') + \Phi^{v}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r \end{aligned}$$

which follows by $\boldsymbol{v}(p) - \boldsymbol{v}(q) \geq \boldsymbol{v}(p') - \boldsymbol{v}(q')$ from the other constraint added in rule ARTHUR⊢RELAX. The rule not only allows us to discard resource before and after the execution of the term e, but also allows us to carry over excess resources, up to $(p - q)$.

Since this is a structural rule concerned with potential only, the consequences concerning the memory consistency of the result v follows trivially form the application of the induction hypothesis.

Note that this case is the only one were we need to make use of r, which essentially allows to preserve superfluous resource. In previous work, this was also necessary for the case dealing with the let-construct, in order to thread the resource needed for the body through the defining clause. However, in ARTHUR the same effect can already be achieved by applying the (improved) ARTHUR⊢RELAX rule at the last step of the defining clause.

ARTHUR⊢SHARE: First observe that $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(z : A) = \Phi^{\boldsymbol{v}}_{\mathcal{H}}(x : A_1) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(x : A_1)$ follows directly by Lemma 6.22 from $\boldsymbol{v} \vDash \curlyvee(A \,|\, A_1, A_2)$ from (3.A) and (3.C). Hence m, m' are identical to what we will obtain from applying the induction hypothesis.

We assume that $z \in \mathrm{FV}(e)$, for otherwise there would be no effect of the application of rule ARTHUR⊢SHARE. Let $\mathcal{S}^\star = (\mathcal{S} \backslash z)[x \mapsto \mathcal{S}(z), y \mapsto \mathcal{S}(z)]$. Obviously $\mathcal{S}, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e[z/x, z/y] \downarrow v, \mathcal{H}'$ is equivalent to $\mathcal{S}^\star, \mathcal{H} \overset{m}{\underset{m'}{\vdash}} e \downarrow v, \mathcal{H}'$ both having a derivation of the same length and structure. The same is true for the uncosted operational semantics.

Before we can make use of the induction hypothesis, we must verify that $\mathcal{H} \vDash_v \mathcal{S}(z) :: A$ justifies both $\mathcal{H} \vDash_v \mathcal{S}^\star(x) :: A_2$ and $\mathcal{H} \vDash_v \mathcal{S}^\star(y) :: A_2$. This follows by the applications of Lemma 6.12 and Lemma 6.19, once in each case. Applying the induction hypothesis concludes the proof for this case, with $\mathcal{H}' \vDash v :: D$ following directly once again.

ARTHUR↓UNIT, ARTHUR↓TRUE & ARTHUR↓FALSE: We consider the case of $e = *$, the cases for $e = \texttt{true}$ and $e = \texttt{false}$ follow analogously. Observe that by the premises we have $\boldsymbol{v}(p) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r = r$, since $\phi \vDash \{p = 0\}$ and Γ is required to be the empty typing context by the typing rule. For any $m \geq r$ we must choose $m' = m$ by the requirements of the annotated evaluation rule for the unit constant. We can easily verify that $m' \geq \boldsymbol{v}(p') + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(() : \mathsf{unit}) + r$ holds as required, since the sum on the right hand side is in fact equal to r, for

the unit type (or boolean type) never carries any potential by Definition 6.20 and $v \vDash \{p' = 0\}$ by premises (3.A) and (3.C).

The final remaining claim, memory consistency for the result value, written $\mathcal{H} \vDash_v () :: \mathsf{unit}$, follows regardless of v and $\mathcal{H}$ directly from Definition 6.14 (in the same manner we have $\mathcal{H} \vDash_v \mathbf{tt} :: \mathsf{bool}$ and $\mathcal{H} \vDash_v \mathbf{ff} :: \mathsf{bool}$ too), concluding the proof for this case.

ARTHUR↓REC: We have $v = \ell$ for some fresh $\ell \notin \mathrm{dom}(\mathcal{H})$; $\mathcal{H}'(\ell) = w$ and $w = \langle y{:}x.e_1 \,,\, \mathcal{S}^\star \rangle$ with $\mathcal{S}^\star = \mathcal{S} \setminus \{x, y\} \cap \mathrm{FV}(e_1)$. Fix $r \in \mathbb{Q}^+$ and choose any $m \in \mathbb{N}$ such that $m \geq v(p) + \Phi^v_{\mathcal{H}}(\mathcal{S} : \Gamma) + r = \big(1 + \textsc{TypSize}(\Gamma)\big) + \Phi^v_{\mathcal{H}}(\mathcal{S} : \Gamma) + r$, since $\phi \vDash p = 1 + \textsc{TypSize}(\Gamma)$.

We observe that Γ has no potential, since $\Phi^v_{\mathcal{H}}(\mathcal{S} : \Gamma) = 0$ holds by Lemma 6.22, using $\phi \vDash \bigcup_{z \in \Gamma} \curlyvee\big(\Gamma(z) | \Gamma(z), \Gamma(z)\big)$ and $v \vDash \phi$ (3.C). By the premises (3.A) and (3.B) we have $\mathrm{dom}(\mathcal{S}^\star) = \mathrm{dom}(\Gamma)$. Hence for all $z \in \mathrm{dom}(\Gamma)$ we deduce that $\textsc{TypSize}\big(\Gamma(z)\big) = \textsc{Size}\big(\mathcal{S}^\star(z)\big)$ by Lemma 6.17 and premise (3.D). We therefore obtain $m \geq \textsc{Size}(w) + r$, since by Definition 6.3 we have $\textsc{Size}(w) = 1 + \sum_{z \in \mathrm{dom}(\mathcal{S}^\star)} \textsc{Size}\big(\mathcal{S}^\star(z)\big)$.

We choose $m' = m - \textsc{Size}(w) = r$ accordingly to rule ARTHUR↓REC and verify that $m' \geq v(p') + \Phi^v_{\mathcal{H}'}(v : \forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} B) + r$ holds as required, since $\phi \vDash p' = 0$ and because the potential of function types is defined to be zero in Definition 6.20.

It remains to show that $\mathcal{H}' \vDash_v v :: \forall \alpha \in \psi. A \xrightarrow{q \triangleright q'} B$ holds, which follows easily by choosing the existentially quantified entities exactly as given in the annotated typing, premise (3.A). Note that $\mathcal{H} \backslash \ell \vDash_v \mathcal{S}^\star :: \Gamma$ holds by premise (3.D) and the requirement that ℓ was fresh, $\ell \notin \mathrm{dom}(\mathcal{H})$, which is a premise of rule ARTHUR↓REC (3.B).

ARTHUR↓APP: In this case, premise (3.A) was derived through a terminal rule, ARTHUR⊢APP, while (3.B) is not. This is the reason for the induction over the lexicographically ordered lengths of both derivations. We will reconstruct the typing context for the function's body from the invariant (3.D), which had to be strengthened considerably for this particular purpose, in order to apply the induction hypothesis to glean information about the functions resource usage. This approach is also taken in the proof cases for the linear pair projections.

Fix $r \in \mathbb{Q}^+$ and choose $m \in \mathbb{N}$ such that $m \geq \boldsymbol{v}(q) + \Phi_{\mathcal{H}}(\mathcal{S} : \boldsymbol{v}(\Gamma)) + r$. We recall that the term at hand is $y'x'$ with the typing context being $\Gamma = \{x'{:}A, y'{:}\forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B\}$. By premise (3.B) we have $\mathcal{S}(y') = \ell$ and $\mathcal{H}(\ell) = \langle y{:}x.e\,,\, \mathcal{S}^\star\rangle$. Hence there is only one possible rule of Definition 6.14 that could have led to premise (3.D), which yields the existence of a typing context $\Gamma^\star$, variables $p^\star, p'^\star$ and a constraint set $\phi^\star$ having the following properties:

$$\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma^\star \tag{6.15}$$

$$\boldsymbol{v} \vDash \phi^\star \tag{6.16}$$

$$\Gamma^\star \vdash^{p^\star}_{p'^\star} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : \forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B \mid \phi^\star \tag{6.17}$$

We apply the inversion lemma 6.13 and obtain

$$\phi^\star \vDash \big(\forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0\big) <: \forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B \tag{6.18}$$

$$\Gamma_0, x{:}A_0, y{:}\forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \vdash^{q_0}_{q_0'} e : B_0 \mid \xi_0 \tag{6.19}$$

$$\Gamma_0 \subseteq \Gamma^\star \quad \wedge \quad \mathrm{dom}(\Gamma_0) = \mathrm{FV}(e) \setminus \{x, y\} \tag{6.20}$$

$$\alpha_0 \cap \big(\mathrm{FV}_\diamond(\Gamma_0) \cup \mathrm{FV}_\diamond(\phi)\big) = \emptyset \tag{6.21}$$

$$\phi^\star \cup \psi_0 \vDash \xi_0 \tag{6.22}$$

$$\phi^\star \vDash \bigcup_{z \in \Gamma_0} \curlyvee\big(\Gamma_0(z) | \Gamma_0(z), \Gamma_0(z)\big) \tag{6.23}$$

Using almost all of the above, we apply the type rule ARTHUR⊢REC to derive $\Gamma_0 \vdash^{1\,+\,\mathrm{TYPSIZE}(\Gamma_0)}_{0} \texttt{rec}\ y\ \texttt{=}\ \texttt{fun}\ x \to e : \forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \mid \phi^\star$ which we immediately use together with (6.16) and $\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma_0$ (from (6.15) and (6.20)) to derive

$$\mathcal{H} \vDash_v \ell :: \forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \tag{6.24}$$

by applying a rule of Definition 6.14.

There is a slight abuse of notation above by writing integer constants directly on the turnstile again, as we did in Chapter 4, instead of introducing two fresh resource variables and adding two corresponding constraints. It is clear that adding a constraint of the form *"variable equals non-negative constant"* is harmless if that variable is fresh and occurs nowhere else. The actual values of these two resource variables, apart from their existence, does not matter either, since these values do not occur anymore in the statement (6.24). The reason that we do not need to care about these two values is that they represent the cost incurred at the creation of the closure that is now applied to, a concern that was dealt with in the previous case.

We apply the induction hypothesis for

$$\Gamma_0, x{:}A_0, y{:}\forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0 \vdash^{q_0}_{q_0'} e : B_0 \mid \xi_0 \tag{6.25}$$

$$\mathcal{S}^\star[y \mapsto \ell, x \mapsto \mathcal{S}(x')], \mathcal{H} \vdash^{m}_{m'} y'x' \downarrow v, \mathcal{H}' \tag{6.26}$$

$$\boldsymbol{v} \vDash \xi_0 \tag{6.27}$$

$$\mathcal{H} \vDash_v \mathcal{S}^\star[y \mapsto \ell, x \mapsto \mathcal{S}(x')] :: \Gamma_0, y{:}\forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0, x{:}A_0 \tag{6.28}$$

(6.25) is identical to (6.19).

(6.26) is a premise of (3.B).

(6.27) follows from $\boldsymbol{v} \vDash \phi^\star$ (6.16), $\phi^\star \cup \psi_0 \vDash \xi_0$ (6.22) and $\boldsymbol{v} \vDash \psi_0$. The latter itself being directly derived from $\boldsymbol{v} \vDash \psi$(3.C), again $\boldsymbol{v} \vDash \phi^\star$ (6.16) and (6.18) via subtyping rule SubFun.

(6.28) follows in three steps: first, from (6.15) and (6.20) we obtain $\mathcal{H}\backslash\ell \vDash_v \mathcal{S}^\star :: \Gamma_0$; second, from (6.24); and third, from (3.D) we have $\mathcal{H} \vDash_v \mathcal{S}(x') :: A$ to which we apply Lemma 6.19 using $\phi^\star \cup \psi \vDash A <: A_0$ which in turn is a consequence of (6.18).

For the application of the induction hypothesis We choose m such that it satisfies our original requirements, which is possible since

$$\begin{aligned}
m &\geq \boldsymbol{v}(q) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}\Big(\ell : \forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} B\Big) + r \\
&= \boldsymbol{v}(q) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A) + r \\
&\geq \boldsymbol{v}(q_0) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A_0) + r \\
&= \boldsymbol{v}(q_0) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A_0) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}\Big(\ell : \forall\alpha_0 \in \psi_0.A_0 \xrightarrow{q_0 \triangleright q_0'} B_0\Big) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}^\star : \Gamma_0) + r
\end{aligned}$$

The first equality follows directly by Definition 6.20. The next inequality follows from (6.18), which yields both $\boldsymbol{v}(q) \geq \boldsymbol{v}(q_0)$ and $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A) \geq \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}(x') : A_0)$ via Lemma 6.21 from $\phi^\star \vDash A <: A_0$. The last equality follows since $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S}^\star : \Gamma_0) = 0$ by (6.23) and Lemma 6.22.

Thus we obtain m' such that

$$m' \geq \boldsymbol{v}(q_0') + \Phi^{\boldsymbol{v}}_{\mathcal{H}'}(v : B_0) + r \geq \boldsymbol{v}(q') + \Phi^{\boldsymbol{v}}_{\mathcal{H}'}(v : B) + r$$

as required, where the latter inequality follows again from (6.18) and Lemma 6.21.

Furthermore, the induction hypothesis also yields $\mathcal{H}' \vDash_v v :: B_0$ as well, from which the required $\mathcal{H}' \vDash_v v :: B$ can be derived by (6.18) and Lemma 6.19, concluding this case.

ARTHUR↓LINPAIR: For the term in consideration we have $e = (e_1 \,\&\, e_2)$ of type $A^{a \triangleright a'} \& B^{b \triangleright b'}$ under the constraint set $\phi \cup \psi \cup \xi$. The result of the evaluation being $v = \ell$ with the post heap $\mathcal{H}' = \mathcal{H}[\ell \mapsto w]$. The newly added value is a closure $w = \langle (e_1 \,\&\, e_2) \,,\, \mathcal{S}^\star \rangle$ having the contextual stack $\mathcal{S}^\star = \mathcal{S}\restriction \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)$.

The annotated evaluation rule ARTHUR↓LINPAIR requires that $m = m' + \textsc{Size}(w)$. Hence it already suffices to show that $\boldsymbol{v}(p) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S} : \Gamma) + r - \textsc{Size}(w) \geq \boldsymbol{v}(p') + \Phi^{\boldsymbol{v}}_{\mathcal{H}'}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big) + r$. We prove this inequality by separately showing $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S} : \Gamma) \geq \Phi^{\boldsymbol{v}}_{\mathcal{H}'}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big)$ and $\boldsymbol{v}(p) = \textsc{Size}(w)$. Note that we already have $\boldsymbol{v}(p') = 0$ by the annotated typing rule ARTHUR⊢LINPAIR (3.A).

By Definition 6.3 we have $\textsc{Size}(w) = 1 + \sum_{x \in \mathrm{dom}(\mathcal{S}^\star)} \textsc{Size}\big(\mathcal{S}^\star(x)\big)$. From premise (3.D) we know that there exists a typing context $\Gamma^\star$ with $\mathcal{H}\backslash\ell \vDash_{\boldsymbol{v}} \mathcal{S}^\star :: \Gamma^\star$ and $\mathrm{dom}(\Gamma^\star) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)$. By Lemma 6.17 the value- and type-sizes agree, therefore $\sum_{x \in \mathrm{dom}(\mathcal{S}^\star)} \textsc{Size}\big(\mathcal{S}^\star(x)\big) = \sum_{x \in \mathrm{dom}(\Gamma^\star)} \textsc{TypSize}\big(\Gamma^\star(x)\big)$. Assumption (3.D) furthermore allows us to apply Lemma 6.18, eventually yielding $\textsc{Size}(w) = 1 + \sum_{x \in \mathrm{dom}(\Gamma)} \textsc{TypSize}\big(\Gamma(x)\big)$. Since we have $\boldsymbol{v}(p) = 1 + \textsc{TypSize}(\Gamma)$ from $\boldsymbol{v} \vDash \xi$ we have thus established $\boldsymbol{v}(p) = \textsc{Size}(w)$.

In order to prove the inequality $\Phi^{\boldsymbol{v}}_{\mathcal{H}}(\mathcal{S} : \Gamma) \geq \Phi^{\boldsymbol{v}}_{\mathcal{H}'}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big)$ we observe that by Definition 6.20 we have $\Phi^{\boldsymbol{v}}_{\mathcal{H}'}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big) = \min_{\Gamma^\star \in \mathcal{G}} \Phi^{\boldsymbol{v}}_{\mathcal{H}\backslash\ell}(\mathcal{S} : \Gamma)$, where $\mathcal{G}$ is the set of all suitable typing contexts as in the definition. The inequality then follows by $\Gamma \in \mathcal{G}$, which is a trivial consequence from the premises (3.A), (3.C) and (3.D). Note that the removal of location ℓ from heap $\mathcal{H}'$ is not problematic, since $\mathcal{H}'\backslash\ell = \mathcal{H}$ as ℓ was ascertained to be fresh by the premises of rule ARTHUR↓LINPAIR (3.A).

For the remaining claim we observe that $\mathcal{H}' \vDash_{\boldsymbol{v}} \ell :: A^{a \triangleright a'} \& B^{b \triangleright b'}$ follows by the same premises that we have already checked to compute $\Phi^{\boldsymbol{v}}_{\mathcal{H}'}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big)$.

ARTHUR↓FST & ARTHUR↓SND: We only consider ARTHUR↓FST, since the case for ARTHUR↓SND follows analogously. Let $e = \texttt{fst(}x\texttt{)}$, $\mathcal{S}(x) = \ell$, $\mathcal{H}(\ell) = w$ and $w = \langle (e_1 \,\&\, e_2) \,,\, \mathcal{S}^\star \rangle$. From premise (3.D) we know that there exists some annotated typing context $\Gamma^\star$, resource variable p, p' and a constraint set ψ such that $\mathcal{H}\backslash\ell \vDash_{\boldsymbol{v}} \mathcal{S}^\star :: \Gamma^\star$ and $\Gamma^\star \vdash^{q}_{q'} (e_1 \,\&\, e_2) : A^{a \triangleright a'} \& B^{b \triangleright b'} \mid \psi$ and $\boldsymbol{v} \vDash \psi$ holds. We choose $\Gamma^\star$ minimal with respect to its potential in accordance to Definition 6.20 such that $\Phi^{\boldsymbol{v}}_{\mathcal{H}}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big) = \Phi^{\boldsymbol{v}}_{\mathcal{H}\backslash\ell}(\mathcal{S}^\star : \Gamma^\star)$ holds. Note that our choice is guaranteed by premise (3.D).

The judgement $\Gamma^\star \vdash^{q}_{q'} (e_1 \,\&\, e_2) : A^{a \triangleright a'} \& B^{b \triangleright b'} \mid \psi$ must have been justified by rule ARTHUR⊢LINPAIR, having among its premises $\Gamma^\star \vdash^{p_1}_{a'} e_1 : A \mid \xi$ and

$\psi \vDash \{p_1 = a + q,\, q = 1 + \textsc{TypSize}(\Gamma^\star)\}$. Since the evaluation of e_1 is one step shorter than for e (3.B), we may apply the induction hypothesis to obtain that for any $m_0 \geq \boldsymbol{v}(a) + \boldsymbol{v}(q) + \Phi^{\boldsymbol{v}}_{\mathcal{H}\setminus\ell}(\mathcal{S}^\star : \Gamma^\star) + r$ there exists an $m' \geq \boldsymbol{v}(a') + \Phi^{\boldsymbol{v}}_{\mathcal{H}'}(v : A) + r$ with $\mathcal{S}^\star, \mathcal{H}[\ell \mapsto \mathbf{Bad}] \vdash^{m_0}_{m'} e_1 \downarrow v, \mathcal{H}'$.

According to rule Arthur↓Fst we choose $m = m_0 - \textsc{Size}(w)$. and obtain $\mathcal{S}, \mathcal{H} \vdash^{m}_{m'} \texttt{fst(}x\texttt{)} \downarrow v, \mathcal{H}'$. Note that by our choice of $\Gamma^\star$ we have $m \geq \boldsymbol{v}(a) + \Phi^{\boldsymbol{v}}_{\mathcal{H}}\big(\ell : A^{a \triangleright a'} \& B^{b \triangleright b'}\big) + r$ as required, provided that $\boldsymbol{v}(q) = \textsc{Size}(w)$, which in turn follows exactly as already described in the previous case for rule Arthur↓LinPair, using the Lemmas 6.17 and 6.18.

Memory consistency for the result follows directly from the application of the induction hypothesis, since the result value and type remains unchanged.

□

7 Implementation

Performing the automated amortised analysis technique proposed in Chapter 6 by hand is not only non-automatic, but it is also extremely tedious and error-prone. The in-depth manual $LF_\diamond$ analysis in Section 5.6 required already a lot of simplification in order to become digestible. We will therefore use results by an actual implementation of the ARTHUR system to evaluate its performance on several program examples. The amortised analysis technique presented in this thesis was actually implemented by the author three times.

First, the first-order version shown in [HJ03] was implemented in OCAML, comprising a total of 5220 lines of code. This implementation was used in the Mobile Resource Guarantees project [MRG05]. It notably included an interpreter to measure cost and already allowed *user-defined recursive data types*, including the ability, to assign *user-defined heap space costs* to each constructor.

Second, a prototype implementation of the early *higher-order* ARTHUR system, which was not yet fully defined at that point in time, was written. The implementation, although performed in OCAML again, was not based on the first, but rewritten from scratch. It consisted of 6440 lines of code overall. The implementation was rough, but fully usable and included lambda abstraction and additive pairs.

Third, a HASKELL implementation was required for the integration into the EmBounded project [EmB09], which is concerned with the domain-specific high-level language HUME [HM03]. Therefore, the second implementation was ported to HASKELL by Hans-Wolfgang Loidl, who also maintains the front-end of the analysis that generates an abstract syntax from a program file. The abstract syntax tree is delivered in ghost-let-normal form (see Section 7.2 below) and contains several useful annotations, such as framedepth, number of variables in patterns, feedback from compiler optimisations, and standard types for each node. The back-end of the analysis is maintained by the author. This implementation comprises about 16000 lines of well-commented HASKELL code, excluding the transformation to let-normal form, which was integrated into the HUME compiler.

HUME differs notably from ARTHUR by using *under- and over-application* instead of *lambda abstractions*. Support for the forms added, and the capability to treat the latter was kept, unlike support for *additive pairs*, which were dropped. The capability to account for destructive matches was also kept, despite HUME not supporting this feature. HUME uses a box-based[A] deallocation mechanism instead, which indiscriminately reclaims all the memory used during the execution of a box at the end

[A] The HUME language features a *coordination layer* that describes multiple, interacting, re-entrant processes; called "boxes". Each box itself is a functional program expression. The coordination layer links the boxes together and allows interaction with an external, imperative state through streams, which are delivered piece-wise to the system of concurrent boxes.

of the execution cycle for that box. Some additional experimental features, such as numeric potential, were also implemented.

Overall, adjusting the implementation of ARTHUR to HUME was a huge benefit for the analysis, since the tool gained a significant maturity. Notable features of the implementation, most of which being described more closely in [LJ09], include:

a) several comfortable language features required by proper languages, such as
   i) mutually recursive let-definitions,
   ii) under- and over-application of functions,
   iii) lambda abstraction,
   iv) built-in functions for arithmetic and optimised vector operations,
   v) pattern matches with multiple branches,
   vi) user-definable recursive data types, and
   vii) nested pattern matches;

b) good performance through the integration of the LP-solver lp_solve [BEN];

c) an easy-to use web-interface with several program examples (see `http://www.embounded.org/software/cost/cost.cgi`, [EmB09]);

d) the formulation of cost-bounds in a non-technical human-readable language, as opposed to pure annotated types (see Section 7.4);

e) the possibility to interactively explore the entire solution space of admissible cost-formulae (see Section 7.5); and

f) the availability of several interesting cost metrics (see Section 7.1), namely
   i) worst-case *heap space* usage for LFPL, ARTHUR, HAM-interpreter and HUME (all are available in both a boxed and an unboxed variant),
   ii) maximum *stack space* usage for ARTHUR and HUME (boxed and unboxed),
   iii) *call-count* metric to bound the number calls made to a user-defined set of function identifiers from above, and
   iv) *worst-case execution time* of HUME code on the Renesas M32C/85U and PowerPC MPC8280 microcontrollers.

7.1 Interchangeable cost metrics

The point about the use of interchangeable cost metrics is quite an important one. We will take an advantage of the availability of these resource metrics in the discussion of our program examples in Chapter 8.

Note that in order to define cost metrics that are not just concerned with heap memory, it is not enough to change the definitions of Size and TypSize. For example, any operation has a time cost, so a worst-case execution time (WCET) metric requires slight changes to all type rules. Fortunately, it is possible to do this in a generic way, which also subsumes the cost metrics for heap space, as shown in our work [JLH+09].

The basic idea is to insert fresh *resource constants*, which are subtracted from the petty potential at any possible step. For example, the constraints added through the type rule Arthur⊢Let for local definitions then changes to

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{p_1}_{p_1'} e_1 : A \mid \phi \qquad \Gamma_2,\, x{:}A \vdash^{p_2}_{p_2'} e_2 : C \mid \psi \\ \xi = \{p = \mathit{KLet1} + p_1,\ p_1' = \mathit{KLet2} + p_2,\ p_2' = \mathit{KLet3} + p_3\}\end{array}}{\Gamma_1, \Gamma_2 \vdash^{p}_{p'} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : C \mid \phi \cup \psi \cup \xi}$$

We see that the Arthur type rule can be easily recovered by simply setting the resource constants *KLet1*, *KLet2* and *KLet3* to zero. The operational semantics are augmented accordingly, which allows us to prove the soundness of the analysis independently from the actual values for the resource constants. Their actual values do not matter, as long as each constant always matches up with itself between operational semantics and type rules. Thereby we gain a very flexible analysis, that can bound the usage of any compositional quantifiable resource.

A WCET metric then requires one to obtain static bounds on each resource constant, so that the augmented annotated operational semantics match with reality. In practise, this requires the examination of the code fragments that are generated by the compiler that correspond to each resource constant. It is paramount that the WCET for the code fragments is compositional. If the target architecture suffers from timing anomalies due to the design of its cache and pipeline, a significant overhead may be incurred to obtain compositional bounds on WCET.

The **aiT** tool from AbsInt GmbH [FHL+01] was used during the EmBounded research project to obtain the resource constants for the WCET of Hume code on Renesas

M32C/85U and PowerPC MPC8280 microcontrollers. The WCET was determined in terms of processor clock cycles. For example, the worst-case clock cycle values obtained to execute a HUME `let` expression on a Renesas M32C/85U microcontroller are

$$KLet1 = 142 \qquad KLet2 = 0 \qquad KLet3 = 3 + RetLab$$

where RETLAB is an intermediate parameter obtained from the compilation process, specifically depends on the number of return labels contained in the intermediate C-code through the formula $RetLab = \max\big(116, 51 + 15 \cdot (\#\text{ReturnLabels})\big)$.

7.2 Ghost-let-normal form

Our implementation of the analysis transforms a program into a *ghost-let-normal form* first. The transformation to entirely harmless and straightforward, and was first discussed in our work [JLHH10]. It is also important to note that this technique is entirely independent of the amortised analysis technique, but simply facilitates the modelling of actual resource usage by our operational semantics. It allows us to apply our generic cost-aware operational semantics to a wider range of actual machine cost behaviour.

The translation to ghost-let-normal form adopts a much stricter policy than the ordinary let-normal form. The let-normal form just restricts most subexpressions to variables only, whereas the ghost-let-normal form demands that all variables used in subexpressions are unique *ghost variables*. A ghost-variable is a use-once variable, that was introduced by a so called ghost-let-expression, an addition to the syntax of the language. Only the defining clause of the ghost-let-construct must use normal program variables.

The ghost-let construct is otherwise semantically identical to the normal let-construct. We distinguish it from the normal let-construct by writing it in upper-case, i.e. `LET` $v = e_1$ `IN` e_2. The annotated type rule is also equivalent to the ordinary let, except for using a separate set of resource constants. Therefore the ghost-let construct can be assigned a different cost than the ordinary let-expression, which is important for the modelling of stack-space and worst-case execution time metrics as we shall see. The idea is that the automatic transformation into let-normal form only introduces these ghost-lets, in order to distinguish them from ordinary let-expressions that are used by the programmer.

For an example, the expression $\texttt{foo}(x, \texttt{bar}(x, 5))$ will be translated into

$$\texttt{LET } x_1 = x \texttt{ IN LET } z_1 = \texttt{(LET } x_2 = x \texttt{ IN LET } y_1 = 5 \texttt{ IN bar } x_2\ y_1\texttt{) in (foo } x_1\ z_1\texttt{)}$$

The identifiers with indices are ghost-variables, all others are not. We see that this translation also expresses the order in which entities are placed upon the stack, since the ghost-let defining x_1 is inserted before the one for z_1. Note that the above example is formulated in terms of the first-order system, where `foo` and `bar` are top-level function names. In the higher-order system, `foo` and `bar` must be wrapped through a ghost-let before their use, just like all other ordinary identifiers.

The advantage of adopting such a policy is that the cost metrics for stack space and worst-case execution time become much simpler. In these metrics, the cost of creating a constant on the top of the stack might differ from pushing a variable onto the top of the stack, etc. However, thanks to the ghost-let we do not need to distinguish this for all expressions that have subexpressions, since these can now expect that the cost for preparing their arguments on the stack has already been accounted for. The ghost-let itself is assigned a zero cost, but the defining clause will incur the cost for pushing a variable, or creating a constant through the ordinary rules.

7.3 Inference of Annotated Types

The automatice inference of the annotated types is performed in five stages:

a) In the first stage of the analysis process, a program is parsed and transformed into ghost-let-normal form, as described in the previous section. This intermediate form is branched out of the compilation process, after possible optimisations have been performed by the compiler. The analysis must be informed of any optimisations, since these might have an affect on whether a cost bound can be guaranteed or not.

b) The second stage then performs the analysis itself on all top-level functions. This is done for each block of mutual recursive functions within the call-graph at once, first starting the with block of functions that do not depend on any other functions. Analysing each block itself consists of three steps:

i) For each block of mutual recursive functions, the type signature is enriched with fresh resource variables. Each function type is non-parametric at first, i.e. for the generic function type $\forall\alpha \in \psi.A \xrightarrow{q \triangleright q'} C$ we have both α and ψ being empty initially. Together with the types obtained from analysing all previous blocks, this yields an annotated typing signature for all the functions that might be called within the current block.

ii) Afterwards, in the second step of stage two, the defining body of each function is analysed by constructing the ARTHUR type derivation. This is done by traversing the standard typing derivation *once* only. All occurring types are always given fresh resource variables as needed, which are then restricted by subtyping upon application to functions or constructors, or by merging the annotated result types of different conditional branches. This process is syntax-driven, without any choice. Structural rules such as subtyping, weakening and sharing are only used when required by the syntax, based on the free variables of subterms, and at the latest point possible. Suptyping simply generates the necessary inequalities, similar to sharing. Both always succeed in generating constraints, although the generated constraints might not be solvable in all cases. Rule ARTHUR$\vdash$RELAX is built-into every other rule, so it is effectively applied in each step.

iii) The third step of the second stage simply refines the annotated signature, by placing the whole generated constraint into the type of each function within the mutual block, setting α simply as the set of all free variables within that constraint set. Note that resource parametricity thus only applies to calls to functions which are strictly below the current function in the call graph. Recursive calls or calls between mutually recursive functions cannot make use of resource parametricity. The reason is that generation of the constraints for a block of mutual recursive functions depends on itself. The problem is similar to *polymorphic recursion* [Hen93, KTU93] and is likely to require a similarly complicated technique to allow *resource parametric recursion*.

Note that if there is a main expression given in the program besides function definitions, then it is analysed at the very end of stage two, just using step b.ii above.

c) For the third stage, an objective function is generated based on a heuristic, which generally prefers small values on the left-hand sides of function arrows

and large values for resource variables occurring on the right-hand side of functions. A unique slack variable is also introduced for each inequality, turning it into an equation. The slack variables are also minimised by the objective function, in order to minimise waste.

d) The fourth stage consists of simply feeding the constrains to an LP-solver, such as lp_solve [BEN]. In practise, we have found that the generated constraints can always be easily solved as they are by lp_solve. In fact, attempts to simplify the constraints both in their number and the number of variables used showed an increase in runtime and a loss of numeric stability.

The efficiency of solving the generated constraint sets is not an issue. The LPs generated for the program examples in Chapter 8, which usually involve less than 5000 constraints over less than 10000 variables, can all be solved within a single second on typical contemporary hardware (2.53GHz Intel Mobile Core 2 Duo, with 6MB cache and 4GB memory). Note that in general, even simple linear programs may require a long time to solve. The reason that the large linear programs generated by our amortised analysis technique are easy to solve lies likely in the observation that they always admit an integer solution, if they are solvable at all, as noted in our work [HJ03].

Actual runtime measurements for inferring the annotated types for a number of realistic program examples are given in [LJ09], and are repeated for convenience in Section 9.1, where we discuss the asymptotic complexity of the inference process.

It is noteworthy that the implementation can also print the generated constraints into a file, both solved and unsolved. Due to the constraint generation in stage two being syntax driven, each constraint can be prefixed with the precise line and column number of the syntactic construct that triggered its generation. Each constraint is furthermore labelled with three letters that identify the applied type rule. It is also possible to see symbolic sums using the names of the resource constants that led to the constant coefficient of a constraint. This yields a very high transparency to the amortised analysis process.

e) For the final stage, any solution that is found by the LP-solver in the previous stage is then presented to the user in the form of an annotated type and/or a human-readable closed cost formula for the analysed program.

7.4 Bounds in Natural Language

The implementation allowed us to gather informative feedback from programmers. A prominent complaint was that the presentation of resource bounds in the form of numeric annotations to the types is difficult to understand for the non-expert user. We therefore implemented an elaboration module, which translates the annotated types, produced by our analysis, into more easily digestible closed form expressions.

Communicating the cost bound through type annotations allows for a high precision, but listing all constructors within a type produces a long an unwieldy output. Furthermore we experienced that most annotations turn out be zero anyway for many program examples. Only a few annotations are crucial, and it is therefore desirable to focus the attention of the user on these. Therefore, elaboration module helps to interpret the annotated types by ignoring irrelevant information, summing up weights in equivalent positions and producing a commented cost-formula, parametrised over a program's input.

Recall the example of a function that inserts an integer into a balanced red-black tree from Section 1.2. Again, it is not important for this example what this function actually does. The result of the analysis for the function is the following annotated type, which according to our observation in Section 5.2.1, lists each type together with all its constructors for its annotation, as well as the annotated types of the arguments of each constructor. The symbol `#` represents the "self-type", i.e. the innermost enclosing type.

```
(int,tree[Leaf|Node<0>:colour[Red<10>|Black<28>],#,int,#]) -(20/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
```

This annotated type gives rise to the same simple arithmetic cost formula that we had stated in Section 1.2, through the definition of potential. However, the annotated type is certainly much harder to digest than the equivalent arithmetic cost formula. Therefore, the elaboration module of the implementation now automatically yields this simplification on demand and prints this cost formula after the annotated type.

```
Worst-case Heap-units required in relation to input:
20 + 10*X1 + 18*X2
    where X1 = number of "Node" nodes at 1. position
          X2 = number of "Black" nodes at 1. position
```

There is nothing deep in this transformation from annotated type to natural language. First, some simplifications are applied, such as shifting potential outwards as far as possible without worsening the bound, as discussed in Section 5.2.1. Second, it just applies the definition of potential, to generate a simple sum.

Note, however, that the first step looses some minor information. The cost formula tells us that we need at most 10 resource units per tree node, plus 18 for those nodes that are black, whereas the annotated type tells us that we do not need any resources per tree node, but 10 for red nodes and 28 for black nodes. The subtle difference lies in the timing: with the annotated type, we know that there is no resource consumption until after matching on the colour of a node, because the potential only becomes available at that point. It is conceivable that the programmer might make use of this information to optimise the code. However, the heuristic used to determine the objective function does not guarantee that we will always receive the above annotated type. LP-solver might just as well return also return the type

```
(int,tree[Leaf|Node<10>:colour[Red<0>|Black<18>],#,int,#]) -(20/0)->
                              tree[Leaf|Node:colour[Red|Black],#,int,#]
```

7.5 Interactive Solution Space Exploration

Programs often admit several possible resource bounds and it is in general not clear which bound is preferable. Recall the example for zipping two lists from Section 4.4.3, whose heap space consumption is proportional to the length of the shorter input list. The implementation of our analysis returns the following result.

```
zip: (list[Nil|Cons<0>:int,#],list[Nil|Cons<3>:int,#])
                                      -(0/0)->  list[Nil|Cons<0>:int,#]

Worst-case Heap-units required to call zip in relation to input:
  3*X1
     where  X1 = number of "Cons" nodes at 2. position
```

So the heuristic employed for the objective function cause the LP-solver to choose a bound in relation to the length of the second input list.

However, we might have further knowledge about our program that tells is that we usually call function `zip` with the first input list being shorter. The implementation allows the user to explore the entire space of admissible solutions for the linear program by increasing or decreasing the penalty attached to a resource variable in the type.

```
Enter variable for weight adjustment or "label","obj" for more info;
                                                leave blank to proceed: X1
Old objective weight: 8.0 Enter relative weight change: 10
Setting CVar '155' to weight '18.0' in objective function.
Attempting solving now...Done!
                          (4.999876022338867e-3s required for solving.)

zip: (list[Nil|Cons<0>:int,#],list[Nil|Cons<3>:int,#])
                                       -(0/0)->  list[Nil|Cons<0>:int,#]

Worst-case Heap-units required to call zip in relation to input:
  3*X1
     where  X1 = number of "Cons" nodes at 1. position
```

So we choose to increase the penalty for `X1` arbitrarily by 10. We learn that the name `X1` actually refers to resource variable 155, which we could have equally used, since it is also possible to view the type annotated with the names of its resource variables.

Note that the implementation keeps the linear program in memory, and only adjusts the objective function. The solver that has been used here, lp_solve [BEN], supports this through incremental solving, i.e. its starts the search from the previously found solution, instead of starting from scratch, which results in almost instantaneous solving. Concrete performance statistics about this particular improvement for solving the generated LPs can be found in [LJ09].

The new solution is printed on the screen again, and the user may then specify another cost variable to be altered, until the cost bound is satisfactory. Step-by-step, the user can thus explore the entire solution space for the analysed program.

The built-in heuristic for the objective function is usually able to guess a "good" result for many program examples right away. However, allowing the user to tune the solver's priorities is also a good way of understanding the overall resource behaviour of a program. So even in the many cases that are already properly resolved by the heuristic, this interaction may offer valuable insights.

8 Examples

In this chapter we will evaluate our analysis method by discussing the cost bounds inferred by our automated amortised analysis for various interesting program examples. The examples discussed in this chapter focus on the use of higher-order types. Several of these examples were first discussed in a limited form in our work [JLHH10]. All cost bounds presented in this chapter were inferred by our implementation, which was created during the EmBounded project [EmB09] as discussed in the previous chapter.

In order to match the implementation used, the programs in this chapter are presented in the syntax of expression-level Hume [HM03], which is quite similar to Haskell. However, recall from the previous chapter that lambda abstraction and destructive matches are extensions to Hume which are not officially supported. The notation that we use here is more compact than Arthur, and not in let-normal form, since a transformation to let-normal form is a part of our implementation, as described in Section 7.2. Functions are defined in a Haskell-style notation that uses multiple rules with pattern matching, which is equivalent to a top-level `match` expression. The basic type of integers is parametrised with its bit-size precision. We also use the familiar notation of `[]` for `Nil` and `_:_` for `Cons` in the pre-defined `list` type: `data [a] = Nil | Cons a [a]` for some of the program examples. This notation is automatically de-sugared before the analysis is performed.

We apply the amortised analysis technique to the following program examples:

List Folding. Widely-used list folding can be precisely bounded for all cost metrics.

Composition and Nesting. Our technique can easily deal with function composition and nested data structures.

Twice, Quad and Quad of Quad. Second and third order types mixed and mingled for the repeated application of an unknown function.

Tree Operations. Two higher-order tree traversal examples.

Sum-of-Squares. Compares first-order and higher-order version of a simple program and demonstrates the experimental feature of numeric potential.

Abstract Machine for Arithmetic Expressions. Revisits the example from Section 5.6, before defunctionalisation. Fails due to lack of resource parametric recursion.

Expression Evaluator. Higher-order evaluator for a loop-free programming language, that includes local definitions and conditionals.

Inverted Pendulum. Demonstrates a successful worst-case execution time analysis for a real-world embedded application.

```
data list = Cons int list | Nil;

foldl :: (int -> int -> int) -> int -> list -> int;
foldl f n l = match l with
                | Nil -> n
                | Cons x xs = foldl f (f n x) xs;

add :: int -> int -> int;
add x y = x + y;

sum :: list -> int;
sum xs = foldl add 0 xs;
```

Figure 8.1: Source code of list-sum

8.1 List Folding

Our first example demonstrates the probably most common use of higher-order types in functional languages: list folding. This is an important standard example for a higher-order function, and thus the analysis of this function is also discussed in our earlier publications [LJ09, JLHH10].

The code shown in Figure 8.1 defines the standard fold-left function, and then applies it to sum up a list of integers values. We see that `foldl` has the second order type $(\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{list}(\mathsf{int}) \to \mathsf{int}$ and successively reduces a list by applying a function, which is parametrised through the first-order argument to function `foldl`.

A bound on the heap usage for the `sum` function is given by an annotated type, as printed out by the implementation of the analysis. Recall from the discussion in Section 5.6 that we need to mention each constructor for its annotation, and that the symbol `#` represents the "self-type", i.e. the innermost enclosing type. For example, the print out "`list[Cons<2>:int,#|Nil]`" is to be interpreted as $\mu X.$`list[Cons<2>:int,`X`|Nil]` and thus represents the type of lists of integers, that also has a potential of two times the length of the list; the equivalent ARTHUR type notation would be $\mathsf{list}(\mathsf{int}, 2)$.

```
ARTHUR3 typing for ArthurHeapUnboxed:
foldl: ((int,int) -(0/0)-> int, int, list[Cons: int, #| Nil:])
                                                        -(0/0)-> int
ARTHUR3 typing for ArthurHeapUnboxed:
add: (int,int) -(0/0)-> int

ARTHUR3 typing for ArthurHeapUnboxed:
sum: (list [Cons: int, #| Nil:]) -(1/0)-> int
```

So we learn from the annotated type for `sum` that computing the sum of an integer list by folding only requires a single heap cell in the unboxed memory model. This bound is exact, since the heap cell is required to create an empty closure for `add` in the definition of function `sum`, in order to submit it to `foldl`, since only closures can be given as arguments.

A much more interesting cost bound for this example can be obtained if we exchange the cost metric from heap space usage to the worst-case clock cycles required to process these functions on a Renesas M32/85U.

```
ARTHUR3 typing for HumeTimeM32:
foldl: ((int,int) -(0/525)-> int,
                  int,list[Cons:<517> int, #| Nil:<225>])
                                                    -(330/0)-> int
ARTHUR3 typing for HumeTimeM32:
add: (int,int) -(637/0)-> int

ARTHUR3 typing for HumeTimeM32:
sum: (list[Cons:<1714> int, #| Nil:<394>]) -(826/0)-> int
```

The bound for function `foldl` on its own is rather meaningless, since the cost of `foldl` naturally depends on the resource behaviour of the supplied higher-order argument. This is what we refer to by resource parametricity: the specialisation of an annotated function type is delayed until all higher-order arguments are provided. Just for the purpose of printing the annotated type, some rather random annotations are picked for the higher-order arguments. The annotated type tells us which annotations were picked by the solver; in this instance the unlikely beneficial type $\mathsf{int} \times \mathsf{int} \xrightarrow{0 \triangleright 525} \mathsf{int}$ had been chosen in order to obtain a concrete type for the screen print.

The type for function `add` is inferred as $\mathsf{int}\times\mathsf{int}\xrightarrow{637\triangleright 0}\mathsf{int}$, so it takes at most 637 clock cycles to add two arbitrary integers in the given worst-case execution time cost metric. Inserting this annotated type of `add` for the function argument of `foldl` then leads to a useful bound on executing function `sum`. Recall that, thanks to resource parametricity, there is no need to re-analyse `foldl` again: the linear constraints that were inferred and used for the annotated typing of `foldl` are just solved again, with the added constraints that the annotated types of the argument and function `add` must agree.

In order to understand the annotated type of `sum`, we ask the tool to print the results in a more human-readable format, as described in Section 7.4. This then yields the following printout.

```
Worst-case Time-units required to call foldl in relation to its input:
  555 + 0*Y1 + 517*X2
    where Y1 = (overall bound depends on the 1. function argument)
          X2 = number of "Cons" nodes at 1. position

Worst-case Time-units required to call add in relation to its input:
  637

Worst-case Time-units required to call sum in relation to its input:
  1220 + 1714*X1 where X1 = number of "Cons" nodes at 1. position
```

So we learn that a call to function `sum` is linearly dependent on the length of the input list. For a list of length n we have the bound $1220+1714n$ on the worst-case execution time. The analysis uses the fact that any list has precisely one nil constructor, and hence adds its potential of 394 to the petty potential of 826, in order to arrive at the value 1220 for the constant term of the cost formula. The number 1714 is just the potential attached to each cons constructor, as seen in the annotated type. Also, note the warning attached to `Y1`, saying that the bound for `foldl` may vary with this input argument through resource parametricity, as we have just observed above.

A comparison between predicted worst-case and measured usage for calculating sums of integer lists by folding is shown in Table 8.1 on the next page. The measurements were obtained from an instrumented version of the underlying abstract machine, that counts resources used during the execution, and has been provided by Hans-Wolfgang Loidl for the EmBounded research project. The resource metric used is the *boxed*

Metric		$N = 1$	$N = 2$	$N = 3$	$N = 5$	$N = 10$	$N = 500$
Heap	Analysis	16	24	32	48	88	4008
	Measured	16	24	32	48	88	4008
	Ratio	1.00	1.00	1.00	1.00	1.00	1.00
Stack	Analysis	(35) 34	(46) 34	34	34	34	34
	Measured	34	34	34	34	34	34
	Ratio	1.00	1.00	1.00	1.00	1.00	1.00
Time	Analysis	3603	5615	7627	11651	21711	1007591
	Measured	3066	4606	6146	9266	16926	771526
	Ratio	1.18	1.22	1.24	1.26	1.29	1.31

Table 8.1: Analysis predictions and measurements for calculating sums of integer lists of length N by folding

Hume memory model, which implies that adding two integer values incurs a heap space cost of two units, in order to store the new integer value. Note that the table shows inferred bounds and measurements for the entire code, which also includes the costs for generating the test input. The absolute values given in the table are thus higher than the values calculated from the annotated type for function `sum` alone.

We see that the inferred bounds on heap and stack space usage are the best possible in all cases, since the execution demand meets the predicted worst-case. There is a small glitch for the stack usage bounds for lists of length 1 and 2, where the default heuristic chooses the loose bound shown in parenthesis, in order to maximise the return of unused resources at the end of the computation. However, the better bound of 34 is inferred if we disable the punishment of slack variables through command line option `--nsl`, which is recommended for stack space bounds. Finally, we see that our predictions on worst-case execution time are a pretty good match to actual execution times, with the estimated worst-case being between 18% and 31% higher than the observable cost, which is well below the stated target goal of 50% for the EmBounded project.

```
type num = int 32;

-- foldr :: (a -> b -> b) -> b -> [a] -> b;
foldr f n []     = n;
foldr f n (x:xs) = f x (foldr f n xs);
foldr1::( num -> [[num]] -> [[num]] )-> [[num]] -> [num] -> [[num]] ;
foldr2::([num]->[[[num]]]->[[[num]]])->[[[num]]]->[[num]]->[[[num]]];

-- compose :: (b -> (c -> d)) -> (a -> b) -> a -> (c -> d);
compose f g n l = f (g n) l;
compose1 :: ( [num]  ->  [[num]]  ->  [[num]] ) ->
               ( num  ->  [num] ) ->  num  ->  [[num]]  ->  [[num]] ;
compose2 :: ([[num]] -> [[[num]]] -> [[[num]]]) ->
               ([num] -> [[num]]) -> [num] -> [[[num]]] -> [[[num]]];

-- cons :: a -> [a] -> [a];
cons n l = (n:l);
cons1 ::  [num]  ->  [[num]]  ->  [[num]] ;
cons2 :: [[num]] -> [[[num]]] -> [[[num]]];

-- map :: (a -> b) -> [a] -> [b];
map f l = foldr (compose cons f) [] l;
map1 :: ( num  ->  [num] ) ->  [num]  ->  [[num]] ;
map2 :: ([num] -> [[num]]) -> [[num]] -> [[[num]]];

aomt :: num -> [num];
aomt x = let xp = x + 1 in [xp, 2*xp];

expression
  map2 (map1 aomt) (map1 aomt (aomt 0));
```

Figure 8.2: Source code for composite mapping

8.2 Nesting and Function Composition

Our second program example shows that our amortised analysis technique can easily deal with function composition and nested data structures. Function composition is very essential for higher-order functional programs, since higher-order types allow highly generalised library functions, which are then composed to build programs. Similarly, nested data structures allow the reuse of familiar operators on different levels of the structure.

The central function of this example is the well-known `map` function, which takes a function and a list and applies the function to each element of the list separately, returning the list of results. Instead of defining `map` directly, we make things more interesting by defining `map` through right-associative list folding and function composition. The composition is between the supplied function parameter and a simple `cons` function that just adds an element to a list.

The function to be mapped is an arbitrary demo function called `aomt`, which takes an integer, adds one and then multiplies the result by two. Function `aomt` returns an integer list of length two, which includes the intermediate result. Note that it does not matter for this example what `aomt` actually does, or that the length of its result list happens to be always two. Any mixture of functions of type $A \to \mathsf{list}(A)$ would serve equally well for this example, provided that their resource consumption can be linearly bounded by their input through our analysis.

For an example, we have the following computations:

```
                          aomt 0   = [1,2]
                map aomt (aomt 0)  = [[2,4],[3,6]]
map (map aomt) (map aomt (aomt 0)) = [[[3,6],[5,10]],[[4,8],[7,14]]]
```

This can be continued further by applying maps of `aomt` to the result without any problem for our analysis.

The analysed code is shown in Figure 8.2 on the preceding page. Since our implementation is monomorphic, the code requires some duplication to obtain all necessary typings for `map`, which were omitted in the figure. Note that function $\texttt{map}_n$ actually makes calls to $\texttt{foldr}_n$, $\texttt{compose}_n$ and $\texttt{cons}_n$. Of course, we could have chosen to avoid using the functions `cons` and `compose`, by directly using the Cons: constructor in the body of a lambda abstraction.

Applying our analysis yields the following results for the main expression

```
map2 (map1 aomt) (map1 aomt (aomt 0))
```

where the constructors Cons and Nil are abbreviated by their first letter:

```
ARTHUR3 typing for ArthurHeapUnboxed:
64, list [C: list [C: list [C: int, #| N:], #| N:], #| N:] ,0

ARTHUR3 typing for HumeHeapBoxed:
221, list [C: list [C: list [C: int, #| N:], #| N:], #| N:] ,0

ARTHUR3 typing for HumeStackBoxed:
128, list [C:<11> list [C:<6> list [C: int, #| N:], #| N:], #| N:] ,61

ARTHUR3 typing for HumeTimeM32:
55647, list [C: list [C: list [C: int, #| N:], #| N:], #| N:] ,0

ARTHUR3 typing for CallCount ["aomt"]:
7, list [C: list [C: list [C: int, #| N:], #| N:], #| N:] ,0
```

The bound for unboxed Arthur heap space usage is exact, as we shall see in the next paragraph. The instrumented Hume cost model execution confirms that the bound for the boxed Hume heap usage is exact, i.e. 221 heap units were indeed used, while the stack and WCET metrics have a slack of 20.8% and 59.3% respectively. The slack for both bounds drops below 9% if we look at the degenerated case `map2 (map1 aomt) (map1 aomt [])`, which must also be accounted for. The annotated type for stack usage also informs us that at least $11 \cdot 2 + 6 \cdot 4 + 61 = 107$ of the requested 128 stack units can be reclaimed after the execution, which is only one more than were actually used. The last resource metric `CallCount ["aomt"]` bounds the overall number of calls to function `aomt` by 7, which is exact as well: one initial call, two calls during the first mapping and two calls for each of the two subsequent mappings.

We now investigate how the resource usage of 64 heap units comes to pass for the unboxed Arthur heap metric. The expression clearly computes in three stages:

a) The first stage, the call `aomt 0` requires 4 heap units in order to construct a list of length two, since each list node requires two cells.

b) The second stage, which maps function `aomt` over the previous result, initially requires one heap cell for the creation of an empty closure for `aomt`, in order to pass it as a function argument. In the body of `map1` we similarly require one cell for turning `cons1` into an empty closure and three heap units for the closure that holds the composition of `cons1` and `aomt` (i.e. one plus two to remember the two pointers to the two supplied argument functions). So we see that the constant consumption of stage two is five.

   Executing function `foldr1` then requires another 12 heap units in total: 2 for each node of the two nodes of the outer list and again 2 each node of the two inner lists, which themselves are of length 2 each. Hence, the overall heap usage of stage two is 17.

   We can also apply the analysis on the expression `map1 aomt` to obtain the type

```
ARTHUR3 typing for ArthurHeapUnboxed:
map2 aomt: 3, (list [C:<6> int, #| N:]) -(4/0)->
                          list [C: list [C: int, #| N:], #| N:] ,0
```

   which confirms the computation. Each node of the argument list requires a potential of 6, since it triggers the generation of one outer and two inner list nodes. The call also requires 4 heap cells for the closures of `cons1` and the composition of this closure with function `aomt`.

   Note that the cost of the initial empty closure for `aomt` is *not* included in the annotated function type above, since it must have been created earlier in order to form the expression `map1 aomt` in the first place. This cost is included in the three that precedes the type, which shows the cost that is incurred for defining the expression. The other two heap units of these three are required to store the result of the expression `map1 aomt`, which is a closure due to the missing list argument for the map operation. The closure is not generated for the occurrence of `map1 aomt` in stage two of the evaluation of the large expression, since the call to `map1` is fully applied at this point.

c) The third stage requires again a heap unit to create an empty closure for function `aomt`, since our compiler does not recognise that it could reuse the

previously generated closure. Whether or not such an optimisation is implemented does not affect our analysis directly, since it would just lead to different code after ghost-let-normalisation (i.e. another push-variable instead of under-application).

Another two cells are now required to hold the closure for `map1 aomt` as an argument for `map2`. This leads to a fixed overhead of $3 + 4 = 7$, since another 4 heap cells are required for the closures of `cons2` and the composition of `cons2` with the closure for `map1 aomt`, for the same reason as observed above for stage two.

The inner list nodes of the argument to `map2 (map1 aomt)` must supply a potential of 6, again following our observation from stage two. The outer list nodes must coincidentally also carry a potential of 6. For each outer node of the input, we require four heap units to store the closures generated by executing `map1 aomt` on it, plus two to create the outermost list node that holds the result in the end.

Since the argument for `map2 (map1 aomt)` is a list of length two having inner lists of length two, we compute an overall heap space usage of $7+2\cdot6+2\cdot2\cdot6 = 43$ for stage three.

We can again use the analysis on the expression `map2 (map1 aomt)` to confirm these observations by inferring the function type

```
ARTHUR3 typing for ArthurHeapUnboxed:
5, (list [C:<4> list [C:<6> int, #| N:<2>], #| N:]) -(4/0)->
        list [C: list [C: list [C: int, #| N:], #| N:], #| N:] ,0
Worst-case Heap-units required in relation to input:
9 + 6*X1 + 6*X2
  where X1 = number of "C" nodes at 1. position
        x2 = number of "C" nodes at 2. position
```

which again includes a requirement of two additional heap cells for creating the resulting closure, that is not created during the computation of the large expression.

We must also remark that the heuristic for the objective function delivers a slightly inferior type for the partial expression, namely one that requires 4 additional heap cells, but that also assigns a potential of 4 to the output. Howeber, the bound is equivalent, since the code contains no destructive matches,

hence the four cells that are guaranteed to be left unused at the end of the computation must have been unused throughout. Using the interactive solving procedure to adjust the heuristic for the objective function, as described in Section 7.5, can also deliver the above printed type in a single step.

All the heap usage throughout the evaluation of the expression is cumulative, since the program has no destructive matches. Therefore, we see that the overall heap space usage is $64 = 4 + 17 + 43$, as automatically inferred by our automated analysis for that expression in the first place, without any manual intervention nor interactive solving.

We conclude this example by remarking that the amortised analysis technique is well suited for dealing with nested data structures. This example only dealt with sub-lists having an equal length, but losing this property would not diminish the quality of the inferred bounds, unlike for sized type based approaches. No over-approximation to accommodate the largest sub-list is required. The key point is that potential is always assigned *per node*, so sub-lists of differing lengths can be assigned precisely the potential they require individually.

8.3 Applying Twice, Quad and Quad of Quad

We now discuss an interesting example of two simple higher-order polymorphic functions, shown in Figure 8.3. The function `twice` simply applies a given (endo-)function twice on an argument. The function `quad` applies `twice` twice, effectively applying a given function four times on a given argument. While this example may appear very simple, one should note the difficulty of higher-order composition between `twice` and `quad`, which is certainly not trivial to analyse. The first three expressions already require second order types, while the fourth example expressions involves third order types.

We obtain the following polymorphic type as heap bound for `quad`:

```
ARTHUR3 typing for HeapBoxed:
 (a -(0/0)-> a) -(0/0)-> a -(5/0)-> a
```

```
succ :: int -> int;
succ x = x + 1;

doubl :: [int] -> [int];
doubl []    = [];
doubl (h:t) = h:h:(doubl t);

twice :: (A -> A) -> A -> A;
twice f x = f (f x);

quad :: (A -> A) -> A -> A;
quad f x = twice f (twice f x);


expr1 :: int;
expr1 = twice succ 0;

expr2 :: int;
expr2 = quad succ 0;

expr3 :: int;
expr3 = quad (quad succ) 0;

expr4 :: int;
expr4 = (quad quad) succ;
```

Figure 8.3: Twice and Quad

The resource consumption for `quad` is expressed by the annotation on the top level function type: five heap cells are required to build a closure for `twice f`, which contains one fixed argument. The zeros for the function argument are provisional, the LP-solver has simply chosen *a* possible solution. When applied to a concrete function, the merged constraints will need to be solved once again. Applying the successor function `succ`, which has a fixed cost of four heap units per call, then yields the correct typing of:

```
ARTHUR3 typing for HeapBoxed:   int -(21/0)-> int
```

Using a call-count metric for the number of calls to the function `succ` required to evaluate the second expression `quad succ 0`, we obtain the following bound. This accurately indicates that `succ` is called precisely four times.

```
ARTHUR3 typing for CallCount ["succ"]:   4, int ,0
```

Similarly, for the third expression, which applies `quad` once more we obtain:

```
ARTHUR3 typing for CallCount ["succ"]:   16, int ,0
```

We must note, though, that the current implementation still needs an experimental switch in order to obtain the above result. The treatment of polymorphic types was not originally included in the implementation, and therefore some minor tweaks are still necessary. The same is true for the next result.

An even more interesting result is the one for the fourth expression, where function `quad` is first applied to itself alone, before applying the result to function `succ` again:

```
ARTHUR3 typing for CallCount ["succ"]:0, int -<256>/<0>-> int ,0
```

It may surprise that the automatic amortised analysis presented in this thesis is capable of inferring the correct, but seemingly exponential bound, since the presented technique can only infer linear bounds. However, the bound is in fact constant, since the number of calls to `succ` made by the generated closure does not at all depend on the input to the closure.

```
data tree = Leaf num | Node tree tree;

dfsAcc ::(num -> [num] -> [num]) -> tree -> [num] -> [num];
dfsAcc g (Leaf x) acc = g x acc;
dfsAcc g (Node t1 t2) acc = let acc' = dfsAcc g t1 acc
                            in  dfsAcc g t2 acc';
cons :: num -> [num] -> [num];
cons x xs = x:xs;

revApp :: [num] -> [num] -> [num];
revApp [] acc = acc;
revApp (y:ys) acc =  revApp ys (y:acc);

flatten :: tree -> [num];
flatten t =  revApp (dfsAcc cons t []) [];
```

Figure 8.4: Source code of tree-flattening (`flatten`)

8.4 Tree Operations

The next two examples operate over trees. The first is a tree flattening function, using a higher-order depth-first-traversal of a tree structure that is parametrised by the operation that is applied at the leaves of the tree. The source code is given in Figure 8.4 and was written by Hans-Wolfgang Loidl, who also performed the measurements.

Again, the bounds for heap, stack, and time consumption are linear in the number of leaves (l) and nodes (n) in the input structure: the heap consumption is $8l + 8$, the stack consumption is $10l + 16n + 14$, and the time consumption is $2850l + 938n + 821$.

```
ARTHUR3 typing for HeapBoxed:
 (tree[Leaf<8>:int|Node:#,#]) -(8/0)-> list[Cons:int,#|Nil]
ARTHUR3 typing for StackBoxed:
 (tree[Leaf<10>:int|Node<16>:#,#]) -(14/23)-> list[Cons:int,#|Nil]
ARTHUR3 typing for TimeM32:
 (tree[Leaf<2850>:int|Node<938>:#,#]) -(821/0)-> list[Cons:int,#|Nil]
```

Metric		$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
Heap	Analysis	21	38	55	72	89
	Measured	21	38	55	72	89
	Ratio	1.00	1.00	1.00	1.00	1.00
Stack	Analysis	34	60	66	82	98
	Measured	34	50	50	66	66
	Ratio	1.00	1.20	1.32	1.24	1.49
Time	Analysis	4485	8732	12979	17226	21473
	Measured	4275	7970	11665	15360	19055
	Ratio	1.05	1.10	1.11	1.12	1.13

Table 8.2: Predictions and measurements for higher-order depth-first tree traversal

Table 8.2 compares analysis and measurement results for the tree-flattening example. The bounds for heap are exact. For stack, the analysis delivers a linear bound, whereas the measured costs are logarithmic in general. Here, we could usefully apply the extension of the amortised analysis technique proposed by Campbell [Cam09], as discussed in Section 3.1. Campbell describes methods for associating potential in relation to the depth of data structures, as well as the temporary "borrowing" of potential. Both techniques are generally more helpful for determining stack-space usage. Note that the benefits of tail-call optimisation are already accounted for in our implementation. The time bounds give very good predictions, with an over-estimate of at most 13% for the range of inputs shown here.

The second operation on trees considered is the `repmin` function, shown in Figure 8.5. The function `repmin` replaces all leaves in a tree with the minimal value. This function is implemented in two phases, both using higher-order functions: the first phase computes the minimal element using a tree-fold operation; the second phase fills in this minimal element using a tree-map function.

Table 8.3 on page 194 compares analysis and measurement results for the `repmin` example. Again the bounds for heap are exact. For stack, we observe a linear bound but with an increasing gap between the measured and analysed costs. This is due to the two tree traversals. The time bounds, however, show a good match against the measured costs, with an over-estimate of at most 22%.

```
type num = int 16;
type list = [num];
data tree = Leaf num | Node tree tree;

treefold :: (num -> num -> num) -> tree -> num;
treefold f (Leaf x) = x;
treefold f (Node t1 t2) = f (treefold f t1) (treefold f t2);

treemap :: (num -> num) -> tree -> tree;
treemap f (Leaf x) = Leaf (f x);
treemap f (Node t1 t2) = Node (treemap f t1) (treemap f t2);

mymin :: num -> num -> num;
mymin x y = if (x<y) then x else y;

k1 :: num -> num -> num;
k1 x _ = x;

getmin :: tree -> num;
getmin t = treefold mymin t;

fill :: num -> tree -> tree;
fill z t = treemap (k1 z) t;

repmin :: tree -> tree;
repmin t = let z = getmin t
           in  fill z t;
```

Figure 8.5: Source code for `repmin` program example

Metric		$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
Heap	Analysis	17	35	53	71	89
	Measured	17	35	53	71	89
	Ratio	1.00	1.00	1.00	1.00	1.00
Stack	Analysis	42	69	96	123	150
	Measured	42	52	61	62	71
	Ratio	1.00	1.33	1.57	1.98	2.11
Time	Analysis	5020	10991	16962	22933	28904
	Measured	4633	9395	14157	18919	23681
	Ratio	1.08	1.17	1.20	1.21	1.22

Table 8.3: Analysis predictions and measurements for higher-order tree replacement

8.5 Sum-of-Squares Variations

In this section, we study 3 variants of the classic sum-of-squares example (Figure 8.6 on page 196). This function takes an integer n as input and calculates the sum of all squares ranging from 1 to n. The first variant is a first-order program using direct recursion, which does not construct any intermediate list structures. The second variant uses the higher-order functions `map` and `fold` to compute the squares over all list elements and to sum the result, respectively. The third version additionally uses the higher-order function `unfold` to generate the initial list of numbers from 1 to n. The code was written and measurements were performed by Hans-Wolfgang Loidl for the EmBounded project.

Note that since all variants receive a integer as their sole input, this would imply that the analysis could only infer a constant cost bound, since all integers have the same fixed size. We therefore analyse this example with the experimental feature *numeric potential* enabled. Numeric potential, as described in Section 3.2, essentially determines the potential of a non-negative numeric value as if it would be a list of unit elements of the same length. This is only an analogy, so fractional values are allowed. Likewise, there is no runtime overhead, since the program remains unchanged and deals with numeric values as normal. For an example, the type $(\mathsf{int}, 3.3)$ simply means that the potential required is 3.3 times the actual value supplied.

The feature of numeric potential is still considered experimental, since it is not dealt with in our main soundness proof. Furthermore, the inferred bounds may indeed be

	N = 10			
	Calls	Heap	Stack	Time
sum-of-squares (variant 1: direct recursion)				
Analysis	22	114	30	18091
Measured	22	108	30	16874
Ratio	1.00	1.06	1.00	1.07
sum-of-squares (variant 2: with map and fold)				
Analysis	56	200	114	53612
Measured	56	200	112	42252
Ratio	1.00	1.00	1.02	1.27
sum-of-squares (variant 3: also unfold)				
Analysis	71	272	181	77437
Measured	71	272	174	59560
Ratio	1.00	1.00	1.04	1.30

Table 8.4: Results for three variants of the sum-of-squares function

invalid if a numeric value with a non-zero potential is actually negative at runtime. Our implementation contains no safety checks on the sign for numeric values. However, for this program example it is easy to see that all occurring numerical values will always be non-negative.

Table 8.4 summarises analysis and measurement results for all three variants. As expected, the higher-order versions of the code exhibit significantly higher resource consumption, notably for the second and third variants which generate two intermediate lists. These additional costs are accurately predicted by our analysis. In particular, the heap costs are exactly predicted and the stack costs are almost exact. The time results are within 30% of the measured costs. We consider this to be a very good worst-case estimate for higher-order code.

We exploit the capability to easily exchange the resource metric used by our analysis once more and determine bounds on the total number of function calls in a program expression. This call-count metric is of particular interest for higher-order programs,

```
-- common code
sq n = n*n;   add m n = m+n;
-- map, (left-) fold over lists are standard
-- enumFromTo m n generates [m..n] (tail-recursive)
-------------------------------------------------------
-- variant 1: direct recursion
sum_sqs' n m s = if (m>n)
                   then s
                   else sum_sqs' (n-1) m (s+(sq n));
sum_sqs n = sum_sqs' n 1 0;
-------------------------------------------------------
-- variant 2: uses h-o fcts fold and map
sum xs = fold add 0 xs;
sum_sqs n = sum (map sq (enumFromTo 1 n));
-------------------------------------------------------
-- variant 3: uses h-o fcts unfold, fold and map
data maybenum = Nothing | Just num;

unfoldr :: (num -> maybenum) -> num -> [num];
unfoldr f z = case f z of  Nothing -> []
                         | (Just z') -> z':(unfoldr f z');
countdown :: num -> maybenum;
countdown m = if (m<1) then Nothing else Just (m-1);

enum :: num -> [num];    -- this generates [n,n-1..1]
enum n = if (n<1) then [] else n:(unfoldr countdown n);

sum_sqs :: num -> num;
sum_sqs n = sum (map sq (enum n));
```

Figure 8.6: Source code of sum-of-squares (3 variants)

and discussed in more detail in [LJ09]. The results are also included as the first column in Table 8.4 on page 195. The first variant exhibits the lowest number of function calls, since all three phases of the computation are covered by one recursive function. Thus, we have one function call to the square function and one recursive call for each iteration. The number of iterations is identical to the initial integer value given as input. Additionally, we have one call to the top level function:

```
ARTHUR3 typing for CallCount: (int<2>) -(2/1)-> int
```

The second variant separates the phases of generating a list, computing the squares and summing them. The generation phase, implemented using direct recursion, needs one call per iteration. The other two phases each need two calls per iteration: one for the higher-order function, and one for the function being applied. In total we have $5n + 6$ calls, as encoded by the following type:

```
ARTHUR3 typing for CallCount: (int<5>) -(6/0)-> int
```

The third variant again has three phases. Now all three phases use higher-order functions, with the enumeration being implemented through a call to `unfold`. The number of calls therefore increases to $7n + 1$. This is encoded by the following type:

```
ARTHUR3 typing for CallCount: (int<7>) -(1/0)-> int
```

8.6 Revisiting an Abstract Machine for Arithmetic Expressions

In Section 5.6 we analysed an abstract machine for evaluating arithmetic expression, which was calculated by Graham and Hutton [HW06] from a specification essentially through adding continuations and defunctionalisation. It is now natural to ask whether we can perform the analysis directly on the high-order code before the defunctionalisation step, which is shown in Figure 8.7 on the next page.

Unfortunately, the linear program generated by our analysis is infeasible for any interesting resource metric, since the technique does not support resource parametric recursion.

```
data expr = Val(int) | Plus(expr,expr)

let rec eval = fun e →                          -- eval :: expr →(int → int) → int
  let rec ev_aux = fun c →
     match e with
      |Val(n) → c n
      |Plus(x,y) →
           let rec f = fun n →
             let rec g = fun m →
               let s = n + m in c s
             in eval y g
           in eval x f
  in ev_aux
in
let rec run = fun e →                           -- run :: expr → int
    let rec id = fun x → x
    in eval e id
in run
```

Figure 8.7: Continuation-passing arithmetic evaluator

In order to understand the problem, consider a simple cost metric where the binary +-operator on integers incurs a cost of one, while everything else has a cost of zero. The cost is clearly linear in the input, since precisely one integer addition will be executed per Plus:-constructor in the input.

Note that the problem persists similarly for other cost metrics, especially when the construction of function closures incurs some non-zero cost, as one would normally expect.

Consider the following general annotated type for function `eval`.

$$\texttt{eval} :: \mathsf{expr}\{\mathsf{Val}{:}v \mid \mathsf{Plus}{:}p\} \xrightarrow{x_0 \triangleright y_0} \left(\mathsf{int} \xrightarrow{x_2 \triangleright y_2} \mathsf{int}\right) \xrightarrow{x_1 \triangleright y_1} \mathsf{int}$$

Hence the annotated type of function `g` necessarily is $\mathsf{int} \xrightarrow{x_2+1 \triangleright y_2} \mathsf{int}$, since `g` performs an integer addition (for the cost of one in the considered metric) and then applies function `c` of type $\mathsf{int} \xrightarrow{x_2 \triangleright y_2} \mathsf{int}$. The use of function `g` as an argument in a recursive call to `eval` will then produce the unsolvable inequality $x_2 \geq x_2 + 1$.

The problem is that resource parametricity is not allowed for recursive calls. Given a valuation, any recursive call within the body of the function definition must use the same valuated type as the one obtained from that definition. It seems likely that using techniques similar to *polymorphic recursion* [Hen93, KTU93] may be able to achieve resource parametric recursion, but we leave this for future work.

8.7 Higher-Order Expression Evaluator with Local Definitions

We now examine another evaluator, which was written directly to make use of high-order functions by Hans-Wolfgang Loidl [JLHH10]. The evaluator not only encompasses arithmetic expressions, but also variable definitions and conditionals, but is loop-free. Its code for the evaluation function `eval` is shown in Figure 8.8 on the following page.

We see that the data type of expressions `expr` has 6 constructors: constant integer values; variables; conditionals; definition of local variables; unary operations; and binary operations. The evaluator uses a function `rho` to model the environment, hence variable lookup simply applies this environment function to variable names. The evaluation of a `let` expression then defines a new environment function `rho'`, which returns the newly computed value `x` for the freshly bound variable `v` and which acts like `rho` otherwise.

The code for this higher-order program evaluator is shown in regular HUME-style syntax, which can be directly processed by the HUME tools that implement our amortised analysis technique. The lambda abstraction added to HUME uses a standard notation combined with let-definitions, i.e. `let` fx `=` e `in` is equal to `let` f `=` `fun` $x \rightarrow e$ `in` .

```
type val = int 16;
type var = int 16;
type env = var -> val;

data exp = Const val
         | VarOp var
         | IfOp  var exp exp
         | LetOp var exp exp
         | UnOp  (val->val) exp
         | BinOp (val->val->val) exp exp;

_true = 1; _false = 0;

eval :: (var -> val) -> exp -> val;
eval rho (Const n)       = n;
eval rho (VarOp v)       = rho v;
eval rho (IfOp v e1 e2)  = if (rho v)==_false
                           then eval rho e2
                           else eval rho e1;
eval rho (LetOp v e1 e2) = let x = eval rho e1 ;
                               rho' v' = if v==v'
                                         then x
                                         else rho v'
                           in  eval rho' e2;
eval rho (UnOp  f e1)    = f (eval rho e1);
eval rho (BinOp f e1 e2) = f (eval rho e1) (eval rho e2);
```

Figure 8.8: Higher-order program evaluator

The analysis of the function `eval` produces the following heap bound for an unboxed Hume heap cost metric:

```
ARTHUR3 typing for HeapUnboxed:
eval: (int -(0/0)-> int) -(0/0)->
      exp[Const:int
          |VarOp:int
          |IfOp:int,#,#
          |LetOp<9>:int,#,#
          |UnOp:(int -(0/ANY)-> int<ANY>),#
          |BinOp:(int -> int -(0/ANY)-> int<ANY>),#,#]
                                                    -(0/0)->int
```

We see that the cost of executing function `eval` is at most 9 times the number of `let` expressions contained in its input, since the `LetOp` constructor is assigned a potential of 9, while all other constructors yield no potential.

The cost for evaluating `let` expressions is due to the update of the environment, which creates another closure for the update of the `rho` function. The cost is 9, since in the unboxed Hume memory model, a closure always occupies 4 cells plus the cells required to store the captured free variables, in this case two integers for 2 cells each (`v` and `x`) plus 1 cell for the pointer to the function representing the previous environment `rho`.

Similarly, for the unboxed Arthur model we obtain

```
ARTHUR3 typing for ArthurHeapUnboxed:
eval: (int -(0/0)-> int) -(0/0)->
      exp[Const: int
          |VarOp: int
          |IfOp: int, #, #
          |LetOp:<4> int, #, #
          |UnOp: ((int) -(0/ANY)-> int), #
          |BinOp: ((int,int) -(0/ANY)-> int), #, #]
                                                -(0/0)-> int
```

since an ARTHUR closure occupies 1 cell for the code pointer and uniformly 1 cell for each of the three captured entities.

Unfortunately, the `eval` function cannot be analysed under the boxed memory models, again due to the lack of *resource parametric recursion*. Executing the comparison operation in a boxed memory model causes the allocation of a boolean value within the heap memory. Therefore the cost of executing function `rho'` is increased as compared to function `rho`, hence a different annotated type is required for the second recursive call to `eval` in the branch dealing the the `LetOp` constructor.

The analysis succeeds if the we remove the branch dealing with the `LetOp` constructor from this example and we obtain the following result for the boxed ARTHUR heap cost metric.

```
ARTHUR3 typing for ArthurHeapBoxed:
eval: (int -(0/0)-> int,
       exp[Const: int
           |VarOp: int
           |IfOp:<2> int, #, #
           |LetOp: int, #, #
           |UnOp: ((int) -(0/0)-> int), #
           |BinOp: ((int,int) -(0/0)-> int), #, #]) -(0/0)-> int
```

Since the comparison and the creation of the false constant in the branch that processes the `IfOp` constructor both have a cost of 1 the boxed ARTHUR heap memory model, the analysis attaches a potential of two to this constructor. In other words:

```
Worst-case Heap-units required to call eval in relation to its input:
  2*d010 where d010 = number of "IfOp" nodes at 1. position
```

8.8 WCET of a Real-Time Controller for an Inverted Pendulum

Our last example is an inverted pendulum controller, which is a typical example for embedded controllers found in automotive systems. This example demonstrates that

our technique is able to deliver good bounds on worst-case execution time for real hardware.

The inverted pendulum is a simple, real-time control engineering problem. A pendulum is hinged upright at the end of a rotating arm. The controller receives as input the angles of the pendulum and the rotating arm and should produce as its output the electric potential for the motor that rotates the arm in such a way that the pendulum remains in an upright position.

The control program was executed on a Renesas M32C/85U development board, which was connected through voltage-scaling circuits to the sensors and motor of an actual inverted pendulum. It perpetually computes the motor output through differential equation reasoning, plus some routines handling button presses that are used for initialisation and calibration of sensors and motor. A more detailed description can be found in our publication [JLS+09]. The program comprises about 180 lines of Hume code, and it is one of the examples covered on the demonstration web page of our analysis, `http://www.embounded.org/software/cost/cost.cgi`. The hardware setup, implementation and measurements were performed by Norman Scaife for the EmBounded research project [EmB09].

The measurements conducted during the experiment showed a best-case of 36118, a worst-case of 47635, and average of 42222 clock cycles for processing an iteration of the main control loop. About 6000 iterations were performed during an actual run, where the Renesas M32C/85U actually controlled the inverted pendulum and kept it upright. Since we have used a chip with a clock frequency of 32Mhz, the execution time in milliseconds is obtained by multiplying the number of clock cycles with the factor 0.00003125. The worst-case measured loop time is thus 1.489ms. The pendulum controller can only be made stable with a reaction time of less than 10ms.

The automatic translation from Hume code into the ghost-let-normal form produces about eight hundred lines of code to be analysed. Our implementation of the amortised analysis then generated 1115 linear constraints over 2214 different variables. The overall runtime of the analysis itself was less than second (~ 0.67s) on a seasoned single core laptop (1.73Ghz Intel Pentium M processor with 2MB cache and 1GB of memory). This includes the time for solving the generated linear programming problem ($\sim$0.26s).

The bound on WCET that is inferred by our automated analysis is 63678 clock cycles, or up to 1.990ms on a Renesas M32C/85U. This proves that the controller

is guaranteed to be fast enough to successfully control the pendulum under all circumstances, since it is well below the 10ms limit on the sample rate. Compared to the worst measured runtime during experimentation, our inferred bound on WCET has an margin of 33.7%. This is quite a remarkable margin, since our inferred cost bound is based on the resource constants obtained by the **aiT** tool [FHL+01], which already, and necessarily, include a significant over-approximation on the WCET for each bytecode instruction.

We must also remark that the margins of the WCET on the supplementary branches of the code, that deal with the starting, calibrating and stopping of the experiment through button presses, were initially much worse. This is due to the fact that the execution cost of these branches is very small, and thus entirely dominated by the initial pattern matches that decides between those branches. For a WCET bound, we must necessarily assume that a match on a nested pattern can fail on the very last sub-pattern, whereas during the experimented, the match always failed on an early pattern. These over-approximations builds up, if there are several long patterns which are tested subsequently, especially for the very last branch, whose worst-case execution time must include the time for all previous patterns to fail. By manually rearranging the order of the pattern matches (moving the fail-safe patterns to the end), we could reduce the guaranteed WCET from 7702 down to a reasonable 2711 clock cycles for branch with worst margin. This branch had a measured execution time of up to 2669 clock cycles. We therefore reduced the margin from 188.6% to just 1.6%.

It should also be noted that our automated analysis technique was capable of inferring *exact* bounds on memory consumption for the inverted pendulum controller, namely 299 heap and 93 stack units, which were indeed required by the execution during the conducted experiments.

9 Conclusion

We have presented and proven sound a novel *automatic* compile-time analysis for determining upper-bound formulas on the use of quantitative resources for strict, higher-order, recursive programs. The bounds we infer depend on the sizes of the various inputs to a program. They thus expose the impact of specific inputs on the overall cost behaviour.

A noteworthy merit of our analysis technique lies in its versatility. Firstly, it is not only restricted to functional languages, but extends to imperative object-oriented languages, such as JAVA [HJ06, HR09].

Secondly, it is not only restricted to heap space usage, but can be used to bound other resource requirements of a program. In fact, any compositional quantifiable resource that admits a static worst-case bound for the execution of each elementary language construct can be accounted for. Adaptations of the technique to analyse the worst-case usage of other resources have already been published (e.g. worst-case execution time [JLH+09], stack-space usage [Cam09], and call-counts to selectable functions [LJ09]). The treatment of realistic program examples clearly demonstrates that the technique is truly capable to provide useful and precise bounds for all these resources.

The usefulness of the approach was also demonstrated through its adoption by three successful research projects [MRG05, EmB09, Isl11], and five peer-reviewed publications [HJ03, HJ06, JLH+09, LJ09, JLHH10] at respected conferences.

9.1 Complexity of the Inference

Our experiments based on our implementation show that we created a highly efficient automated resource usage analysis, that can be performed in mere seconds. Relying on linear constraint solving gives us a good intuition that our method is efficient, but we did not prove any statement concerned with the efficiency of our proposed method. We will therefore include some informal considerations about the complexity of the analysis in this conclusion.

The focus of this thesis is placed on proving the soundness of our newly devised automated program analysis. This is evident, for example, in type rules such as ARTHUR⊢GENERALISE, which allows to replace a set of linear constraints with any other constraint set that entails the original one, i.e. that is at least as restrictive. It

is obvious that such a natural rule is sound and how it *can* be used, but its general formulation leaves it unclear how this rule *should* be used. Even worse, rules such as this one allow things that may be sound, but are nonsensical. For example, one might add arbitrarily many superfluous trivial constraints to a constraint set through Arthur⊢Generalise. This means that the asymptotic complexity of our system, as stated in this thesis, is trivially unbounded.

Nevertheless, our implementation shows that the analysis can be performed efficiently. We believe that the analysis scales linearly with the size of the program, provided that one assumes the maximum type size and the depth of the call graph to be constant.

We will now informally underpin this belief, but the following considerations are not a rigorous proof, since we did not formulate an algorithmic type system. Note that the implementation, as discussed in Section 7, does not only differ from the formally stated type rules through language features, such as excluding the treatment of linear pairs, but also by directing the search for an annotated type. This direction was based on our intuition, but has not been formalised besides the code. The way structural rules are to be applied is left unspecified by the type rules themselves. We thus used common sense alone to decide upon their application. For example, the sharing rule is always applied as late as possible and only when absolutely needed (i.e. when a context must be split and a variable occurs free in either branch). The implementation therefore does not cover all valid Arthur type derivations, but we hope that only nonsensical types are excluded. The following considerations about the complexity therefore orient themselves on the design decisions taken during the prototype implementation.

The implementation examines each syntactic construct of the program precisely once to construct the annotated typing derivation. Most type rules introduce a small constant number of constraints, on average two. This agrees with a linear complexity, so we do not need examine these type rules here any further. The exceptional type rules that may introduce a varying number of constraints are:

Arthur⊢Rec	Arthur⊢Generalise	Arthur⊢Supertype
Arthur⊢App	Arthur⊢Share	Arthur⊢Subtype

Let us thus examine the complexity of these type rules more closely, and we will see that of these six rules, only the two type rules Arthur⊢Rec and Arthur⊢App affect the number of generated constraints significantly in practice.

ARTHUR⊢GENERALISE: Let us first consider the type rule ARTHUR⊢GENERALISE. Like we already observed above, this rule alone allows the introduction of arbitrarily many constraints, since it allows to replace the current constraint set by any constraint set that entails the old one. So the new constraint set might be arbitrarily larger, e.g. by adding superfluous constraints over fresh variables or trivial constraints such as $0 = 0$. The new constraint set might also be smaller, but in general there is no bound on the size of the newly introduced constraint set. Therefore, the complexity of our analysis is trivially unbounded.

However, our implementation only applies this rule to create the union of two constraint sets by concatenation, so it does not affect the overall complexity of the implemented algorithm at all. The rule is only invoked at the end of stage b.iii, as depicted in Section 7.3. For each block of mutually recursive functions, the constraint set for the body of each function is inferred separately first, but in the end all functions of the block share the same set of constraints within their resource parametric function type. This is achieved by simply taking the union of the constraint sets that had been inferred independently for each function body. This is justified in terms of the ARTHUR type rules by applying ARTHUR⊢GENERALISE once at the end of examining each function definition, with the newly introduced constraint set ψ being the union of all the individually inferred constraint sets.

We also experimented with optimising the generated constraint sets upon merging, in order to reduce their size. Again, this would be clearly permitted through ARTHUR⊢GENERALISE. However, it turned out that this only happened to increase the difficulty for our chosen LP-solver through numeric instabilities.

ARTHUR⊢SHARE: Inspecting the rules for sharing given in Figure 6.5 shows that sharing generates at most one linear constraint for each resource variable contained in the type to be shared. This leads to a quadratic complexity on its own, since the number of resource variables in a type can only be generously bounded by the overall program size. For example, consider a silly program that just creates an empty list of a highly nested list type, and then returns a pair of this type created by sharing the value, i.e. `let` x `=` $[\,]$ `in (`x`,` x`)` $: (A \times A)$ where $A = \mathsf{list}(\mathsf{list}(\cdots \mathsf{list}(\mathsf{unit}, q_n) \cdots, q_2), q_1)$. However, the maximum size of the types occurring within a realistic program is in practice independent from the overall size of the program. So it is reasonable to assume the number of annotations per type to be a small constant. Under this assumption sharing does not increase the complexity.

ARTHUR⊢SUPERTYPE & ARTHUR⊢SUBTYPE: The considerations for the number of constraints created due to super- and subtyping, governed by the rules shown in Figure 6.4 are similar to both previous observations. For example, rule SUBPAIR technically allows an arbitrary enlargement of constraints like described for ARTHUR⊢GENERALISE. However, again the implementation only concatenates constraint sets to obtain the (multiset) union.

Unlike for ARTHUR⊢GENERALISE, subtyping may also require certain new constraints, whose number is clearly bounded by the number of resource variables contained in the types that are involved in the subtyping. As discussed above, it is reasonable to assume the number of resource variables per type to be constant, so it does not raise the complexity either.

ARTHUR⊢REC & ARTHUR⊢APP: These are the only two rules that, in conjunction, may significantly increase the number of linear constraints that are inferred in practice.

The type rule ARTHUR⊢REC not only adds two constraints plus those obtained from sharing the context of the closure (i.e. sharing the type of each unbound implicit parameter of the function), but more importantly it allows to store a portion of previously generated constraints within the type, in order to allow for resource parametricity. Each subsequent function application through type rule ARTHUR⊢APP then copies the entire constraint set, that had previously been stored in the resource parametric function type.

Assume for a moment that ARTHUR⊢REC is only required once throughout typing the entire program. Since we can apply a function over and over again, we already end up with a quadratic complexity: the size of the constraint set in the parametric type is only bounded by the overall program size, and similarly the number of function applications is also only bounded by the overall program. For example, if half the code of a program of size n was used to define a function, while the other half just repeatedly applies the function, the number of constraints is already $O\left(\frac{n}{2}\right) \cdot O\left(\frac{n}{2}\right) = O\left(n^2\right)$.

A program that requires ARTHUR⊢REC for defining another independent function simply leads to a constraint set having twice the size. However, if we wrap the above program that leads to a quadratic constraint set size within the defining body of another function, and then repeatedly apply this function instead, we have already reached a complexity of $O\left(n^{2^2}\right)$, since each application already produces a constraint set of quadratic size. Of course, we can repeat

Program	Lines of	Constraints		Run-time		Constr.
	Code	Number	Variables	Total	LP-Solve	per LoC
pendulum	190	1115	2214	0.67s	0.260s	5.87
biquad	400	2956	5756	1.43s	0.418s	7.39
gravdragdemo	190	2692	5591	2.14s	1.065s	14.17
cycab	270	3043	6029	2.75s	1.385s	11.27
meanshift	350	8110	15005	11.01s	6.414s	23.17
matmult	100	21485	36638	84.17s	21.878s	214.85

Table 9.1: Run-time for Analysis and for LP-solving

this schema again and again, increasing the size of the constraint set stored in a resource parametric function type, by calling a previously defined resource parametric function in the definition of another. Therefore, if the depth of the call graph of a program is denoted by d, we obtain a complexity of at least $O\left(n^{2d}\right)$. However, we argue that the depth of the call graph of a program is again independent from its overall size.

We therefore conclude from the above observations that our inference algorithm, at least as we have implemented it (see Section 7), scales linearly with the program size, if one assumes that the maximum type size and the depth of the call graph are constant. It is otherwise exponential in the depth of the call graph and quadratic in the type size. This is confirmed by our experiments conducted primarily for the EmBounded research project [EmB09].

For convenience, we repeat a relevant fragment of the results presented in [LJ09] here. Table 9.1 summarises, for several realistic applications written in Hume, the sizes of the generated constraint sets, the number of variables used in the constraint sets, the observed overall runtime of the inference in seconds (i.e. both constraint generation and LP solving), the portion of the runtime spent on solving the constraints only, and the ratio between the number of generated constraints and the lines of code. Note that "lines of code" is quite a coarse measure. We counted the lines in the program files as they were supplied by the programmer. Some programs also incur some syntactic overhead due to using Hume boxes, a mechanism for coordination, which therefore also figures in the number of code lines required. LP-solving was done through calling the lp_solve library [BEN] through the foreign function interface of

Haskell, which also requires some marshalling of the data types. The measurements were performed on a somewhat older 1.73GHz Intel Pentium M with 2MB cache and 1GB main memory only.

The applications used in Table 9.1 to examine the performance of the analysis are as follows. The `pendulum` application balances an inverted pendulum on a robot arm with one degree of freedom, as previously described in Section 8.8. It is a first-order program with few function definitions and accordingly requires few constraints. The `biquad` application is a biquadratic filter application; it uses many Hume boxes. The `gravdragdemo` application is a simple, textbook satellite tracking program using a Kalman filter and vector types. The `cycab` application is the messaging component of an autonomous vehicle. The number of contained function definitions is about average. The `meanshift` computer vision application is a simple lane tracking program, using several functions and vectors. Finally, `matmult` is a function for matrix multiplication, which was automatically generated from low-level, C-like code, through an independent experiment conducted at Heriot-Watt university [GMH+09]. The process of automatic program generation makes heavy use of higher-order functions and of vectors for modelling the state space. The intensive use of higher-order functions leads to a complicated call-graph and in turn to a high number of constraints, requiring a compute-intensive analysis phase. Nevertheless, the time to solve the LP remains well manageable.

9.2 Limitations

Note that, by itself, our analysis is only partially correct, as must be expected for any such analysis dealing with a Turing-complete language. Any bounds reported by our analysis for supplied programs are proven to be upper bounds, however, our analysis may fail in finding such a bound for a specific program.

In particular, it will fail for all programs whose heap space usage is not *linearly* bounded by its input sizes. Nevertheless, the range of interesting programs and program fragments that can be successfully analysed is large, since we can expect that there exists an analysable program for every problem belonging to the complexity class $O(2^{cn})$ due to an earlier result of Hofmann [Hof02]. Indeed, the covered program examples in this thesis are far from being trivial, showing the usefulness of the approach.

Another important limitation is that our technique only *counts* resource usage, but cannot by itself guarantee a safe *use* of resource. So we rely on other independent work to ensure the safety of deallocation, or that no stale pointer is accidentally reanimated, for example. However, these issues were cleanly separated, hence we would not expect any interference when adopting such techniques for safety.

Further work in continuation of this thesis has already been extensively examined in Section 3.2 on page 26, so we only summarize the previously discussed possibilities here:

a) *super-linear bounds*, in order to increase the range of programs that can be successfully analysed;[A]

b) *lazy evaluation*, which is still an unsolved problem for program analysis in general, but appears to be manageable by our technique in ongoing research;

c) *negative potential*, allowing more precise bounds for worst-case execution time in dependency of also the output, rather than the input alone;

d) *lower bounds*, to obtain a complete picture on the resource usage of a program;

e) *ghost-data*, to expose the recursion structure of a program and to provide a fail-safe analysis, thereby enhancing its acceptance; and

f) *numeric potential*, to greatly increase precision for programs that perform intensive arithmetic operations.

In addition to these, one might further explore whether resource parametricity can be extended for (mutual) recursive calls as well, similarly to *polymorphic recursion*, as pointed out in Sections b.iii on page 171, 8.6 on page 199, and 8.7 on page 202; or whether *use-n-times* closures add anything essential, as mentioned in Section 6.1.3 on page 125. However, we believe that especially the latter is relatively minor with respect to the usefulness of the analysis.

Overall, we can safely conclude that none of these limitations would severely restrict the already impressive usefulness of the proposed amortised analysis technique as it is right now.

[A] *Note added in print:* in the meantime amortised analysis with super-linear bounds has been successfully studied in [HAH11, HH10a], albeit not with higher-order functions.

Bibliography

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands*, LNCS 5382, pages 113–132, Berlin, Heidelberg, October 2007. Springer-Verlag. 3.3, 3.3

[Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. 3.3

[ABH+07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, 2007. 5.4

[AGG09] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 129–138. ACM, June 2009. 3.3, 3.3

[AH02] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proceedings of the 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002. 4.3

[AL97] Martín Abadi and Rustan Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 1214, pages 682–696. Springer-Verlag, New York, N.Y., 1997. 6.4

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. In *Proceedings of the 19th European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010. 3.1, 4.3.1

[Bar85] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1985. 1.3, 6.3

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, June 2003. 3.3, 3.3

[BEN] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`. 5.5.1, b., d., 7.5, 9.1

[Ben01] Ralph Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal of Functional Programming*, 11(1):3–31, 2001. 3.3, 3.3

[BFGY08] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *Proceedings of the 7th international symposium on Memory management*, pages 141–150. ACM, 2008. 3.3, 3.3

[BIL03] Marius Bozga, Radu Iosif, and Yassine Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 55–65. ACM, 2003. 4.3.1

[Cam08] Brian Campbell. *Type-base amortized stack memory prediction*. PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, 2008. 3.1, 5.4

[Cam09] Brian Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, LNCS 5502, pages 190–204. Springer, 2009. 3.1, 4.1, 4.3, 8.4, 9

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001. 3.3, 3.3

[CNPQ08] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 151–160. ACM, June 2008. 3.3, 3.3

[CW00] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–198. ACM, 2000. 3.3, 3.3

[Dan08] Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 133–144, January 2008. 3.3

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994. 4.3.1

[Dow97] Mark Dowson. The ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84, 1997. A

[EmB09] Embounded, 2005–2009. EU Project No. IST-510255, see `http://www.embounded.org`. 1.1, 3.1, 6.1.2, 7, c., 8, 8.8, 9, 9.1

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the International Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 469–485. Springer, October 2001. 3.1, 3.3, 3.3, 7.1, 8.8

[FSDF93] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993. 4.1.1

[GL02] Gustavo Gómez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 75–86. ACM Press, 2002. 3.3, 3.3

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 127–139. ACM, Jan. 2009. 3.3, 3.3

[GMH+09] Gudmund Grov, Greg Michaelson, Christoph Herrmann, Hans-Wolfgang Loidl, Steffen Jost, and Kevin Hammond. Hume Cost Analysis for Imperative Pprograms. In *Proceedings of the International Conference on Software Engineering Theory and Practice (SETP)*, July 2009. 9.1

[Gor94] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *In Glasgow functional programming workshop*, pages 78–95. Springer, 1994. 6.4

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981. 2.2

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011. To appear. A

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *7th International Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings*, pages 13–18, 2007. 3.1

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, April 1993. b.iii, 8.6

[HH10a] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *8th Asian Symposium on Programming Languages (APLAS)*, 2010. To appear. A

[HH10b] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *Proceedings of the 19th European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010. 3.2

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003. a., 3.1, 3.1, 4.3.1, 4.3.1, 5.5.1, 5.5.4, 7, d., 9

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006. b., 3.1, 3.1, 4.3.3, 5.2.2, 9

[HM81] Robert Hood and Robert Melville. Real-time queue operations in Pure LISP. *Information Processing Letters*, 13(2):50–54, November 1981. 2.2

[HM00] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 2000. 3.2

[HM03] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003. 3.1, 7, 8

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. In *Nordic Journal of Computing 7(4)*, pages 258–289, 2000. 3.1, 3.1, 4, 5.2.1, 5.5.4

[Hof02] Martin Hofmann. The strength of non-size increasing computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 260–269, New York, NY, USA, 2002. ACM. 3.1, 9.2

[HP99] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 34, pages 70–81, September 1999. 3.2, 3.3, 3.3

[HPS96] R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (gg)*, pages 410–423. ACM, January 1996. 3.2, 3.3, 3.3

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *CSL: 18th EACSL Annual Conference on Computer Science Logic*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331, Heidelberg, 2009. Springer. 3.1, 9

[HS07] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42(9):251–264, 2007. 3.2

[HW06] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004. 5.6, 8.6

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001. 4.3.1

[Isl11] Islay, 2008–2011. EPSRC grant EP/F030657/1, Islay Project: Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds, see `http://www.eng.ed.ac.uk/~idcomislay`. 1.1, 3.1, 9

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369, Heidelberg, 2009. Springer. d., 3.1, 3.1, 4.1, 4.3, 4.3.2, 4.5, 5.2.1, 7.1, 9

[JLHH10] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 223–236, New York, NY, USA, January 2010. ACM. f., 3.1, 3.1, 3.3, 4.3.2, 4.3.3, 4.4, 4.5, 5.5.4, 6.1.1, 7.2, 8, 8.1, 8.7, 9

[JLS+09] Steffen Jost, Hans-Wolfgang Loidl, Norman Scaifen, Kevin Hammond, Greg Michaelson, and Martin Hofmann. Worst-Case Execution Time Analysis through Types. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–16. ACM, July 2009. Work-in-Progress Session. c., 3.1, 3.1, 8.8

[Jon81] H.B.M. Jonkers. Abstract storage structures. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981. 4.3.1

[Jos02] Steffen Jost. Static prediction of dynamic space usage of linear functional programs, 2002. Diploma thesis at Darmstadt University of Technology, Department of Mathematics. 3.1

[Joy96] Ian Joyner. C++??: a Critique of C++, 3rd Edition. Unisys - ACUS, Australia, `http://www.kcl.ac.uk/kis/support/cit//fortran/cpp/cppcritique.ps`, 1996. 1

[Kon03] Michal Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701. 4.3

[KTU93] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):290–311, April 1993. b.iii, 8.6

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. 6.4

[LJ09] Hans-Wolfgang Loidl and Steffen Jost. Improvements to a resource analysis for hume. In *Proceedings of the first International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, LNCS 6324, Heidelberg, November 2009. Springer. e., C, 3.1, 3.1, 7, d., 7.5, 8.1, 8.5, 9, 9.1

[LMJB09] Hans-Wolfgang Loidl, Kenneth MacKenzie, Steffen Jost, and Lennart Beringer. A proof-carrying-code infrastructure for resources. *Latin-American Symposium on Dependable Computing*, pages 127–134, September 2009. 3.1

[Mét88] Daniel Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2), April 1988. 3.3, 3.3

[MPJS09] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, 2009. 3.2

[MPS08] Manuel Montenegro, Ricardo Peña, and Clara Segura. A type system for safe memory management and its proof of correctness. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 152–162. ACM, 2008. 4.3.1

[MRG05] Mobile resource guarantees, 2002–2005. EU Project No. IST-2001-33149, see `http://www.dcs.ed.ac.uk/home/mrg/`. 1.1, 3.1, A, B, 7, 9

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. 6.4

[NAS99] NASA. Mars climate orbiter fact sheet, 1999. See `http://mars.jpl.nasa.gov/msp98/orbiter/fact.html`. B

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. 1.2, 2.1, 2.2, 2.4, 3.2

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. 6.3

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. 4.2

[PSM06] Ricardo Peña, Clara Segura, and Manuel Montenegro. A sharing analysis for safe. In *Proceedings of the Symposium on Trends in Functional Programming (TFP)*, pages 109–128, Nottingham, UK, April 2006. Intellect. 4.3.1

[RBR+05] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 296–309, New York, NY, USA, 2005. ACM Press. 4.3.1

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972. 5.6, 6

[San90] David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the third European Symposium on Programming (ESOP)*, pages 361–376, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 3.3, 3.3

[Sim10] Hugo Simões. *Lazy Amortised Analysis*. PhD thesis, School of Computer Science, University of St Andrews, 2010. Forthcoming, provisional title. 2.4, 3.2

[Svv07] Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications (TLCA 2007)*, LNCS 4583, pages 351–365. Springer, 2007. 3.2

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985. 2.1, 3.1

[TEX03] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, LNCS 2855, pages 340–355. Springer, 2003. 3.3, 3.3

[Vas08] Pedro Baltazar Vasconcelos. *Space cost analysis using sized types*. PhD thesis, School of Computer Science, University of St Andrews, November 2008. 3.2, 3.3, 3.3

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL)*, LNCS 3145, pages 86–101. Springer, 2004. 3.2, 3.3, 3.3, 6.1.3

[Vol07] Matt Volckmann, Venture Development Corp. (VDC). Embedded Systems Market Statistics, Research Report VDC7784-4. `http://www.electronics.ca/reports/embedded/systems_market.html`, December 2007. 1

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001. 4.3.1

Index

abstract interpretation, 32
additive pairs, 124
aliasing, 56, 66

benign sharing, 53

circular data structures, 56
closure, 123, 125
    chains, 124
    use-n-time, 125
complexity of inference, 132, 170, 208
currying, 123

deallocation, 47
dependent types, 31

exact cost formula, 85
example program
    Conventions for, 64
    Abstract Machine for Arithmetic Expressions, 110
    higher-order, 197
    Higher-Order Evaluator for Arithmetic Expressions, 199
    Higher-Order Folding and Composition, 184
    Insertion-Sort, 68, 108
    Inverted Pendulum Controller, 202
    List folding, 179
    List reversal, 66, 105
    List tails, 64, 104
    List zipping, 68, 106
    Sum-of-Squares, 194
    Tree flattening, 191
    Tree Leaf Replacing, 192
    Twice and Quad, 188

freelist, 51

ghost-let, 169

heap, 47, 125

let-normal form, 40
liveness, 57
location, 47
lp-solving
    efficiency, 172

objective function, 171
    heuristic, 107

partial application, 123
potential, 86–88, 151–153
    numeric, 28, 194
    petty, 16–17
        example, 114

recursive data type, 132
    generalisation to, 88
recursive let, 122
resource constant, 168
resource parametric recursion, 171, 214
    example, 199, 202
resource parametricity, 131

example, 180

sized types, 31
stack, 47, 125
storeless semantics, 51

type system
affine, 43
linear, 43
relevant, 43
typing context, 8

valuation, 131
value (operational)
Arthur, 125
LF, 47
boxed, 71, 140
unboxed, 71, 73
variable
free, 9
ghost-, 169
resource-, 113, 131